Covert Channel Analysis and Detection with Reverse Proxy Servers using Microsoft Windows

WJ Buchanan [A] and D Llamas [B]
School of Computing, Napier University, EH10 5DT, Scotland, UK

Keywords: Reverse Proxy Server, Covert Channel, Data Hiding

Abstract

Data hiding methods can be used by intruders to communicate over open data channels (Wolf 1989; McHugh 1995; deVivo, deVivo et al. 1999), and can be used to overcome firewalls, and most other forms of network intrusion detection systems. In fact, most detection systems can detect hidden data in the payload, but struggle to cope with data hidden in the IP and TCP packet headers, or in the session layer protocol. This paper contains proposes a novel architecture for data hiding, and presents methods which can be used to detect the hidden data and prevent the use of covert channels for its transmission. It also presents the method used in creating a system for Microsoft Windows.

1 Introduction

A covert channel is a communication channel that allows two cooperating processes to transfer information in a manner that violates the system's security policy (Berg 1998). It is thus a way of communicating which is not part of the original design of the system, but can be used to transfer information to a process or user that, a priori, would not be authorised to access to that information. Covert channels only exist in systems with multilevel security (Proctor and Neumann 1992), which contain and manage information with different sensitivity levels. This it allows different users to access to the same information, at the same time, but from different points-of-view, depending on their requirements to know and their access privileges. The covert channel concept was introduced in 1973 (Lampson 1973), and are now generally, classified based on (Gligor 1993):

- **Scenarios**. In general, when building covert channels scenarios, there is a differentiation between storage and timing covert channels (Lipner 1975). Storage covert channels are where one process uses direct (or indirect) data writing, whilst another process reads the data. It generally uses a finite system resource that is shared between entities with different privileges. Covert timing channels use the modulation of certain resources, such as the CPU timing, in order to exchange information between processes.
- **Noise**. As with any other communication channel, covert channels can be noisy, and vary in their immunity to noise. Ideally, a channel immune to noise is one where the probability of the receiver receiving exactly what the sender has transmitted is unity, and there are no interferences in the transmission. Obviously, in real-life, it is very difficult to obtain these perfect channels, hence it is common to apply error correction codes, which can obviously reduce the bandwidth of the channel.
- **Information flows**. With conventional lines of transmission, different techniques are applied to increase the bandwidth. A similar method can be achieved in the covert channels. Channels where several information flows are transmitted between sender and receiver are denominated aggregated channels, and depending on how sent variables are initialized, read and reset, aggregations can be classified as serial, parallel, and so on. Channels with a unique information flow are denominated non-aggregated.

The concern for the presence of covert channels is common in high security systems (Figure 1), such as military ones, where typically two observed users know that someone wishes to listen to their conversations. Many of the studies done about attacks based on covert channels and its prevention have been done by US government and military bodies, such as the National Security Agency, US Air Force, National Computer Security Centre, and so on. However, in other environments it is also possible the existence of covert channels, especially in protocols like the TCP/IP protocol suite (Route 1996; Rowland 1996). The systems involved typically have to be fairly standard in the design, thus an ordinary proxy server could be used as an intermediate server that sits between the client and the origin server. In order to get content from the origin server, the client sends a request to the proxy naming the origin server as the target, and the proxy then requests the content from the origin server, and returns it to the client. The client must thus be specially configured to use the forward proxy to access other sites (The Apache Software Foundation 2001). These proxies can either be forward-looking devices, or work in reverse. A forward proxy typically provides Internet access to internal clients that are otherwise restricted by a firewall, and can use caching to reduce network usage. A reverse proxy, by contrast, appears to the client just like an ordinary WWW server, where no special configuration on the client is necessary. The client thus makes ordinary requests for content in the name-space of the reverse proxy. The proxy then decides where to send these requests, and returns the content as if it was itself the originator (The Apache Software Foundation 2001). A typical usage of a reverse proxy is to provide Internet users access to a server that is behind a firewall.

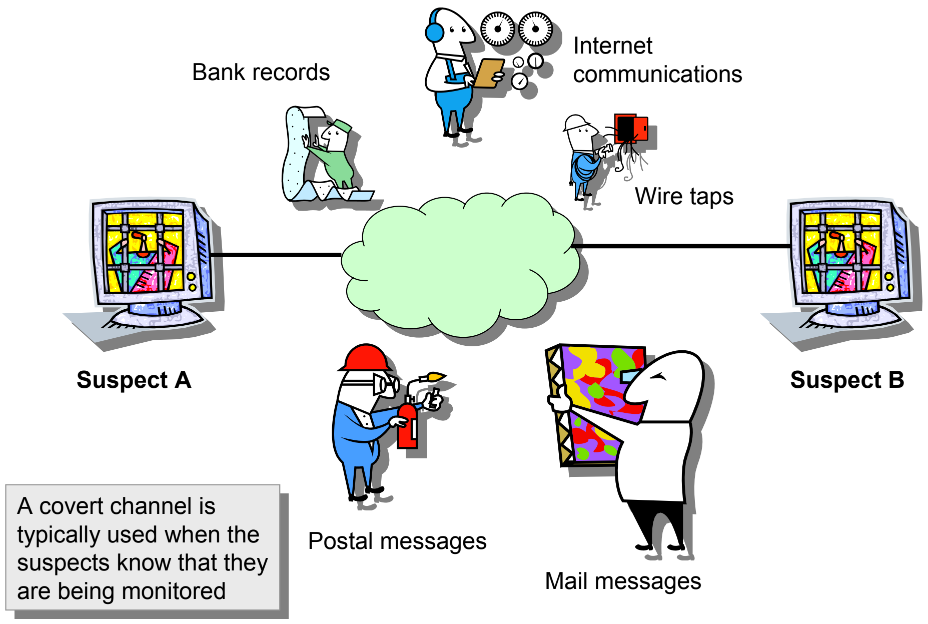


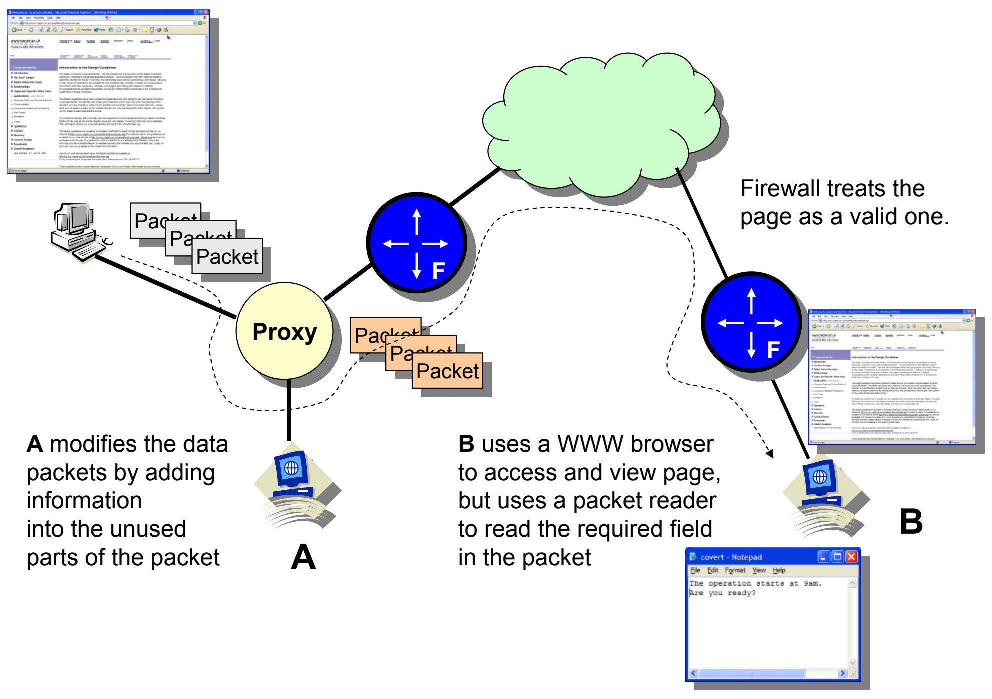
Figure 1: Convert channels

2 Reverse Proxy Server to hide data communications

With security, the control over the information transport mechanisms it is more important than ever in order to guarantee its correct operation under normal circumstances and also when attacks take place. It is also required in order to ensure that these transport mechanisms, which are typically the protocols used, are not used to hide information. An ordinary proxy server is thus useful for these purposes as it concentrates on the traffic associated to the user, both incoming and outgoing, and allows the creation of different kinds of restriction rules, authentication rules, and so on.

The use of proxy forward servers implies a specific configuration in the client side, which in some ways assumes that the user is aware of their connection with external networks will be subject to some rules and controls. In a reverse proxy server, the client views the accessed device like an ordinary service (for example, a WWW server). Thus no special configuration on the client is required. Depending on the kind of reverse proxy server, the returned content can be as if it was itself the origin. This paper proposes a novel architecture for data hiding and detection through a **Data Hiding Intelligent Agent** (DHIA) which is embedded on a reverse proxy server (Figure 2). This agent is responsible for hiding and detection activities, as well as the prevention management and the application of countermeasures to the use of protocols as a mechanism of transport of information. In the current security context, where almost anything can be considered as information, it is highly recommended the use of solutions based on stegano-components, which allows for hidden components that can work in a discrete mode.

Although it is not the focus of this paper, a novel technique known as Dynamic Reverse Proxy (DRP) is presented, where a dynamic connection is established between the user and the reverse proxy server, without requiring any configuration in the client side. This link is totally independent and with its own properties and methods, and is in charge of the control and analysis of the traffic, and is always operating in a hiding way, and in a discrete mode.


Packet
Packet
Packet
Proxy
F
Packet
Packet
Packet
Firewall treats the page as a valid one.
F
A modifies the data packets by adding information into the unused parts of the packet
A
B uses a WWW browser to access and view page, but uses a packet reader to read the required field in the packet
B
covert - Notepad
File Edit Format View Help

The operation starts at 9am.
Are you ready?

Figure 2: Convert channels with data hiding on the proxy

2.1 Data Hiding Intelligent Agent (DHIA)

The proposed architecture for the management of hidden information, as well as the mechanism for its monitoring and detection, prevention and countermeasures activation is the DHIA which is embedded in a Reverse Proxy Server. At present, the designed prototype manages requests at the HTTP level, but a full range of session/application layer protocols, such as FTP and TELNET need to be implemented to cover other protocol channels, and also at other levels such as the network (IP) and transport layer protocols (TCP). The user thus thinks that they are connecting to a WWW server and navigates on it, and is transparent for the navigation. Figure 3 outlines the mechanics of such a system, which are:

- **Reverse Proxy Server**.

- **Embedded DHIA**. The DHIA component must be tested in different location within the scenario where it should operate, in order to find its most convenient situation, taking also into account the Dynamic Reverse Proxy technique.
- **Data Hiding Viewer**. This is run on the client side and shows the hidden data sent by the DHIA embedded in the Reverse Proxy Server.

The system implements covert communication through the manipulation of the Identification field of the IP protocol header (Rowland 1996). In this case, it is implemented using the first byte as sequencer, and the second byte to host the character in ASCII code (multiple packets will thus contains the overall message). The Identification field of the IP protocol header helps with the re-assembly of packet data by remote routers and host systems. Its purpose is to give a unique value to packets, so, if fragmentation occurs along a route, they can be accurately re-assembled.

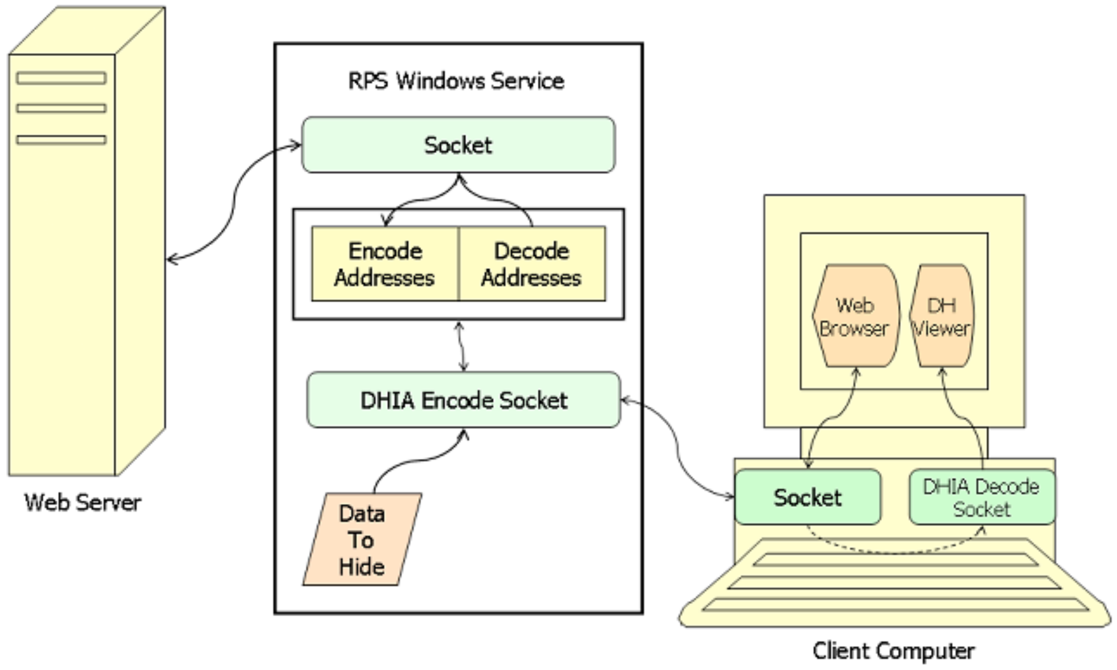


Figure 3: Components of the reverse proxy system

2.2 Monitoring and detection

The monitoring is performed using the common method of placing the network card in promiscuous mode and sniffing the network traffic. It is well-known that covert channels, as well as the steganographical techniques, have a high level of difficulty relating to detection issues. The methods designed by the DHIA are those related with the detection of sequences in the fields of the headings of the different packets that propagate in the network. Other techniques such as the extra traffic payload sensoring, and the analysis of encrypted content or the creation of packets ad-hoc has no effect.

2.3 Prevention and Countermeasures

The prevention and countermeasures for the communication based on covert channels will depend on the techniques from which protection is required. The use of the reverse proxy server as a middleware element implies a scenario based on a minimum of two connections. This is where one is done by the user from his computer to the reverse proxy server in a transparent way, and the other is between the reverse proxy server and the WWW server which the user wants to navigate in. This system allows for an easy control of the packets when they go from one segment to another, and involves the overwriting of the Identification field, which will automatically mean the elimination of the original content of the field. The Identification field will not be used in communications which do not have fragmented data packets. This is because the window size of the TCP connection is normally large enough on interconnection systems.

3 Implementation on Windows platforms

A novel aspect of this research is the implementation of the covert channels on a Microsoft Windows™ platform. In all the Windows operating systems, the TCP/IP protocol is proprietary, and its source code is not accessible which means that the manipulation of the packets in any of the TCP/IP protocol suite is **not** possible from levels above the TCP/IP driver layer. This makes the use of these techniques in a Windows platform more complex.

3.1 Windows NT/2000 Network subsystem architecture

The Microsoft Windows NT/2000 network architecture is composed of software components that provide networking abilities to the operating system. Network communication begins when an application program attempts to access resources on another computer, normally using a layered approach (such as using the network layer for network addressing, and the transport layer for data segmentation). Each layer is thus able to communicate with the layer immediately above and below itself (Microsoft Corporation 2000).

From operation point-of-view, this layered approach is typically seen from two levels:

- **Kernel mode**. This is where the processor executes all instructions, including those designated *privileged*, and can access all of the memory. The mode provides a set of services that the rest of system can use. In Windows, it calls the Hardware Abstraction Layer (HAL) to handle any necessary platform-specific operations.
- **User mode**. This is where an application can only access the memory to which the operating system has granted it permission. A user-mode program can ask the operating system to change the memory map, but, it is the Kernel mode which actually makes the change, if it decides the change is permissible.

Figure 4 shows the view of these modes and the affected drivers and components.

3.2 Network traffic filtering technologies for Windows

The research has involved an extensive search for methods which could be used to implement a system which allowed other system to hook into the data transmissions. In Windows this is not an easy task. In general there are several ways to network traffic filtering (Divine 2002) on a Windows system. For user-mode traffic filtering the methods are:

- **Winsock Layered Service Provider (LSP)**. This method determines the process that called Windows Sockets, such as for QOS (Quality Of Service), encryption of data streams, and so on. Unfortunately, this approach cannot be used on routers, because packets are routed on the TCP/IP level (or even on MAC level).

- **Windows 2000 Packet Filtering Interface.** Windows 2000 provides an API which can can install a set of *filter descriptors*, which can be used by TCP/IP for packet filtering (PASS/DROP). However, rules for filtering are rather limited (pass/drop based on IP address and port information), and this approach can be only be used from Windows 2000 and onwards.

- **Substitution of Winsock DLL.** This approach is mentioned only for security reasons, and is thus not recommend.
- **Global hook of all dangerous functions.** These might include starting with Windows sockets, DeviceIoControl, and so on, and can be done, but it may have an impact on overall system stability and security.

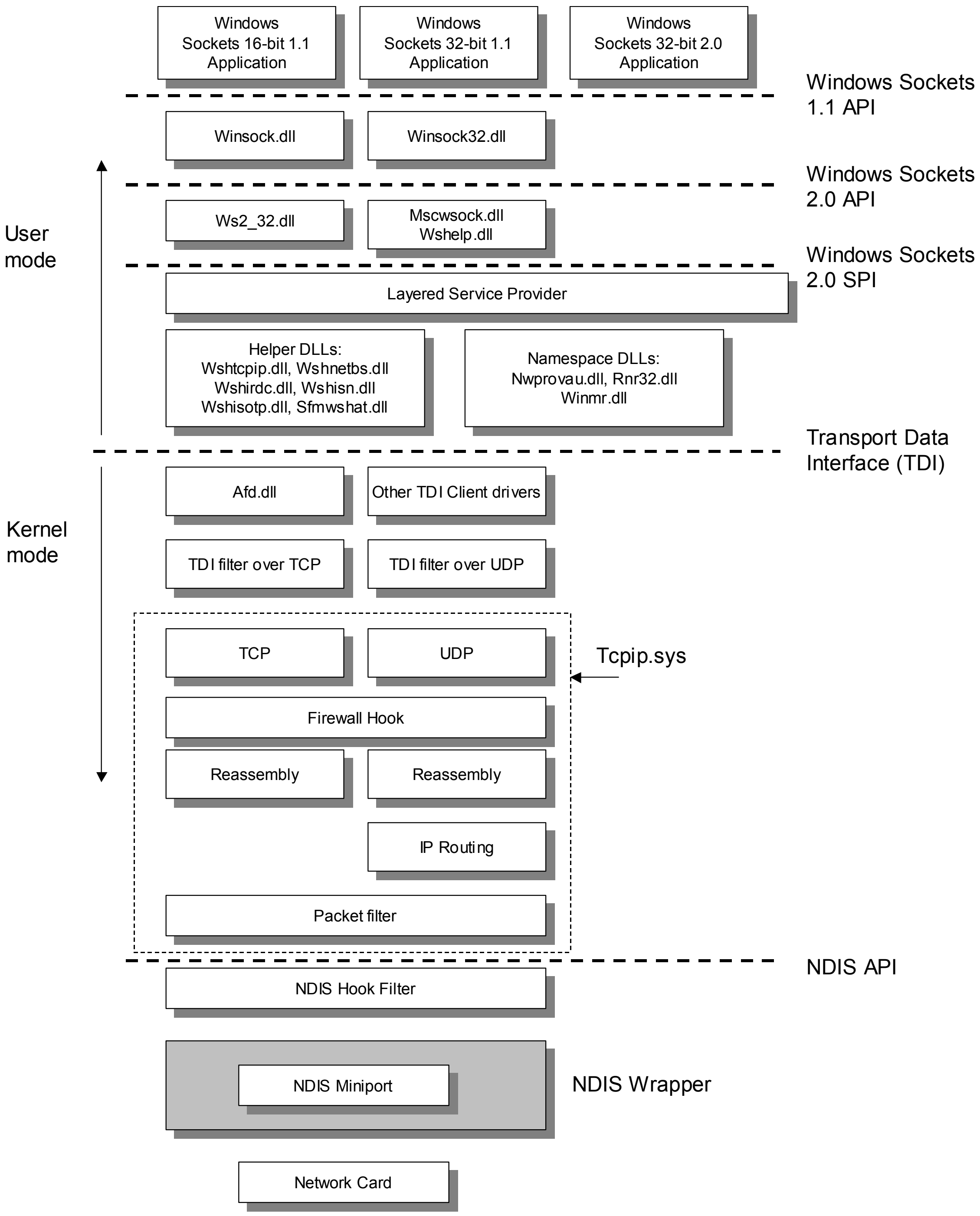


Figure 4: User and kernel modes

For a kernel-mode traffic filtering:

- **Kernel-mode sockets filter.** This technology is applicable for Windows NT/2000, and is based on the interception of all calls from msafd.dll (the lowest level user-mode Windows Sockets DLL) to the kernel-mode module afd.sys (the TDI [Transport Driver Interface]-client, which is a kernel-mode a part of Windows Sockets). This method is interesting, but its possibilities are no much wider, than LSP's. Unfortunately, it has limited portability.
- **TDI-filter driver.** This technology can be implemented on a wide range of Windows platforms, but they vary in their implementation method. As for Windows NT/2000, in the case of TCP/IP filtering, it is necessary to intercept (using IoAttachDevice or patching dispatch table in driver object) all calls directed to devices created by tcpip.sys driver (\Device\RawIp, \Device\Udp, \Device\Tcp, \Device\Ip, \Device\MULTICAST).
- **NDIS Intermediate Driver.** NDIS Intermediate drivers introduced in NT 4.0 to allow developers to write packet capture drivers. These drivers can see all the network traffic on the system as they are placed between protocol drivers and network drivers. Many developers use NDIS Intermediate drivers to provide fault-tolerant and load-balancing options for NICs. Unfortunately, the driver must be digitally signed at Microsoft.
- **Windows 2000 Filter-Hook Driver**. The Filter-Hook Driver was introduced by Microsoft in Windows 2000 DDK (Driver Development Kit). It is not a new network driver class and it is only a way to extend IP Filter Driver to Windows 2000 functionality.
- **NDIS Hooking Filter Driver.** The technique is based on the interception of a subset of NDIS functions which allows it to trace the registration of all protocols installed in the operating system, and opening of network interfaces by them. Among advantages of this is that it has an ease of installation and is transparent support of Dial-Up interfaces. This technique is the one that has been used to implement the Data Hiding Intelligent Agent (DHIA).

5 Conclusions and Future Work

This work has shown that applications that can operate on a discrete mode, such as the reverse proxy servers, which are oriented to the monitoring, detection and prevention of the information hiding techniques, such as covert channels, steganographical techniques, and so on. The use of Reverse Proxy servers as a transparent element and middleware component for highly security environments would be required immediately, as the attack techniques in the information world develop very quickly. It has also been shown the importance of the Reverse Proxy server in the high security environments, as well as the novel development of this kind of tools for windows platforms. The paper also defines the problems that the Windows 2000/NT architecture causes in covert channel development, and proposed the use of the NDIS Hooking Filter Driver.

Currently work is continuing in performing more extensive experiments that will involve the inclusion of new covert channel techniques associated to the manipulation of the heading fields of any of the TCP/IP protocol suite (or other techniques that we are currently investigating such as the use of the TTL field) and an extension to the Reverse Proxy Server to manage requests at different levels.

6 References

Berg, S. (1998). Glossary of Computer Security Terms, http://packetstormsecurity.org/docs/rainbow-books/NCSC-TG-004.txt.

deVivo, M., G. O. deVivo, et al. (1999). "Internet Vulnerabilities Related to TCP/IP." SIGCOMM Computer Communication Review 29.

Divine, T. F. (2002). Windows Network Data and Packet Filtering. I. Printing Communications Assoc., http://www.pcausa.com/.

Gligor, V. D. (1993). A Guide to understanding Covert Channel Analysis of Trusted Systems. Technical Report NCSC-TG-030, National Computer Security Centre.

Lampson, W. (1973). "A note on the Confinement Problem.Communications of the ACM." (16(10)): 613-615.

Lipner, S. B. (1975). "A note on the Confinement Problem." Operating Systems Review, 9(5): 192-196.

McHugh, J. (1995). Covert Channel Analysis. Handbook for the Computer Security Certification of Trusted Systems. USA, Naval Research Laboratory.

Microsoft Corporation (2000). Windows 2000 Network Architecture, http://www.microsoft.com/resources/documentation/windows/2000/server/reskit/en-us/tcpip/part4/tcpappb.mspx.

Proctor, N. E. and P. G. Neumann (1992). Architectural implications of Covert Channels.15th National Computer Security Conference, 28-43.

Route (1996). "Project Loki: ICMP Tunnelling." Phrack Magazine 7(49).

Rowland, C. H. (1996). Covert Channels in the TCP/IP Protocol Suite, http://www.firstmonday.dk/issues/issue2_5/rowland/.

The Apache Software Foundation (2001). Apache HTTP Server. v1.3, http://httpd.apache.org/docs.

Wolf, M. (1989). "Covert channels in LAN protocols." LANSEC'89.

7. Authors Biography

[A] Dr. William Buchanan is a Reader in the School of Computing at Napier University, and leads the Distributed Systems and Mobile Agents (DSMA) research group. He has a PhD and has written more than 20 academic books, and published over 50 research papers.

WWW: **http://www.dcs.napier.ac.uk/~bill**
http://buchananweb.co.uk
Email: **w.buchanan@napier.ac.uk**

[B] David Llamas is a researcher in the School of Computing at Napier University. His research area of interest is Information Hiding, focused on network security by the use of covert channels, and focused on hiding communications by the use of steganographical techniques. He is the administrator of the specialised website: http://www.steganography.org.

WWW: **http://www.dcs.napier.ac.uk/~01009322**
http://www.inchcolm.org
Email: **david@inchcolm.org**

Appendix

The Network Driver Interface Specification allows a hook into the network layer as Ethernet data frames are being passed to and from the Network Interface Card at the Windows Kernel mode. Through API's, the interception of these packets can be finally done in the Windows User mode where most of the Windows software runs. A C DLL was written to intercept and modify outgoing packets and to allow information to be reported to a controlling application. The controlling application was written with Microsoft Visual C#.NET 1.1. Due to security in Windows, the .NET Framework raw sockets and the Berkeley specification are not fully supported. Thus modifying packets at this level is not possible, as the .NET Framework and the Windows operating system will correct any faults it perceives in the header or ignores the header completely and treats it as the payload. This is undesirable behaviour for a system which requires direct hooks into the data traffic flow. The code to initialise the network adaptor, and add the hook is:

```
// Initialise. Selects the adapter and inserts the hook
COVERTAGENT_API int __stdcall Initialise(int adapterIndex)
{

   ADAPTER_MODE Mode;

   Mode.dwFlags = MSTCP_FLAG_SENT_TUNNEL|MSTCP_FLAG_RECV_TUNNEL;
   Mode.hAdapterHandle = (HANDLE)AdList.m_nAdapterHandle[adapterIndex];
   g_adapterIndex = adapterIndex;
   // Create notification event
   hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
   if (!hEvent)
   {
     _DWL0("Failed to create the notification event.\n");
   }

   // Set event for helper driver
   if ((!hEvent)||(!SetPacketEvent(api, (HANDLE)AdList.m_nAdapterHandle[adapterIndex], hEvent)))
   {
     _DWL0("Failed to create notification event or set it for driver.\n");
     return 0;
   }

   // Initialize Request
   ZeroMemory ( &Request, sizeof(ETH_REQUEST) );
   ZeroMemory ( &PacketBuffer, sizeof(INTERMEDIATE_BUFFER) );
   Request.EthPacket.Buffer = &PacketBuffer;
   Request.hAdapterHandle = (HANDLE)AdList.m_nAdapterHandle[adapterIndex];

   SetAdapterMode(api, &Mode);

   return 1;
}
```

The covert channel information is added with:

```
// The return value is the index into to the array where the message was placed
COVERTAGENT_API  int  __stdcall  AddCovertInformation(unsigned  __int32  IPAddress,  unsigned  __int16  Port,
   LPCSTR Message)
{
   size_t len = strlen(Message);
   HANDLE hheap = GetProcessHeap();
   LPSTR p = (LPSTR)HeapAlloc(hheap, HEAP_ZERO_MEMORY, len+1);
   strcpy(p, Message);

   // Replace the existing message
   if (g_InfoArray[i].HiddenInformation)    HeapFree(hheap, 0, (LPVOID)g_InfoArray[i].HiddenInformation);
   g_InfoArray[i].HiddenInformation=p;
   return i;

   g_InfoArray[g_nMessages].IPAddress=IPAddress;
   g_InfoArray[g_nMessages].Port=Port;
   g_InfoArray[g_nMessages].HiddenInformation=p;
   g_nMessages++;
   return g_nMessages-1;
}
```

Finally the packet Identification field can be modified with the required character (`pPacket`):

```
void ModifyPacket(CovertInformation* pCovertInformation, char* pPacket)
{
   WORD* pIdentification = (WORD*)(pPacket+4);

   DWORD charOffset = pCovertInformation->CurrentOffset;
   char aChar = *(pCovertInformation->HiddenInformation + charOffset);
   *pIdentification = (charOffset & 0xFF) + (aChar<<8);

   // If the character just sent was the string terminator (zero) then
   // reset the current offset back to zero, otherwise update it for
   // the next packet to pass through.
   if (aChar == 0) pCovertInformation->CurrentOffset=0;
   else      pCovertInformation->CurrentOffset++;

   RecalcIPChecksum(pPacket);
}
```

The code to process incoming data frames is:

```
COVERTAGENT_API int __stdcall Cycle(int timeout)
{
   // Vars used during the filtering
   WORD wPort;
   DWORD dwIPAddress;
   ether_header*    pEthHeader = NULL;
   char* pIPHeader;
   CovertInformation* pCovertInformation;

   // Vars controlling the loop and time out situation.
   int counter = 0;
   DWORD wfso;
   int timeleft = timeout;
   DWORD starttime = GetTickCount();
   DWORD currenttime;
   while (timeleft>0)
   {
     // Determine if the timeout has been reached

     currenttime = GetTickCount();
     // If we've reached our timeout, or the system timer has wrapped around...
     timeleft = timeout - (currenttime-starttime);
     if ((timeleft <= 0) || (currenttime < starttime))
     {
      break;
     }
     wfso = WaitForSingleObject ( hEvent, timeleft );
     if (wfso == WAIT_TIMEOUT)
     {
      break;
     }
     ResetEvent(hEvent);

     while(ReadPacket(api, &Request)) {
      counter++;
      pEthHeader = (ether_header*)PacketBuffer.m_IBuffer;
      //counter++;
      if (PacketBuffer.m_dwDeviceFlags == PACKET_FLAG_ON_SEND) {
         // Place packet on the network interface
         if(ntohs(pEthHeader->h_proto) == ETH_P_IP)
         {
            pIPHeader = (char*)(pEthHeader)+14;
            ExtractIPandPort(pIPHeader, &dwIPAddress, &wPort);
            pCovertInformation = &(g_InfoArray[0]);
            for(int i=0; i<g_nMessages; i++)
            {
               if ((pCovertInformation->IPAddress == dwIPAddress) && (pCovertInformation->Port))
               {
                  ModifyPacket(pCovertInformation, pIPHeader);
                  break;
               }
               pCovertInformation++;
            }
         }
         SendPacketToAdapter(api, &Request);
      }
      Else {
         // Indicate packet to MSTCP
         SendPacketToMstcp(api, &Request);
      }
     }
   }
```

```
    return counter;
}
```