INL/EXT-13-28224

Host Event Based Network Monitoring

Jonathan Chugg

January 2013

The INL is a U.S. Department of Energy National Laboratory
operated by Battelle Energy Alliance

INL/EXT-13-28224

Host Event Based Network Monitoring

Jonathan Chugg

January 2013

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Prepared for the
U.S. Department of Energy
Office of Electricity Delivery and Energy Reliability
Under DOE Idaho Operations Office
Contract DE-AC07-05ID14517

ABSTRACT

The purpose of INL's research on this project is to demonstrate the feasibility of a host event based network monitoring tool and the effects on host performance. Current host based network monitoring tools work on polling which can miss activity if it occurs between polls. Instead of polling, a tool could be developed that makes use of event APIs in the operating system to receive asynchronous notifications of network activity. Analysis and logging of these events will allow the tool to construct the complete real-time and historical network configuration of the host while the tool is running. This research focused on three major operating systems commonly used by SCADA systems: Linux, WindowsXP, and Windows7. Windows 7 offers two paths that have minimal impact on the system and should be seriously considered. First is the new Windows Event Logging API, and, second, Windows 7 offers the ALE API within WFP. Any future work should focus on these methods.

EXECUTIVE SUMMARY

Securing the country's energy sector infrastructure from cyber-attack is critical to the well-being of the American people and is a central focus to the Department of Energy's (DOE) Office of Electricity Delivery and Energy Reliability (OE) Cybersecurity for Energy Delivery Systems (CEDS) program. The DOE program aims to enhance the reliability and resilience of the nation's energy infrastructure by reducing the risk of energy disruptions due to cyber-attacks. The purpose of this research project is to demonstrate the feasibility of a host event based network monitoring tool and its effects on host performance. The research focus is how to apply such a tool and evaluate its practicality on a SCADA network.

Current host based network monitoring tools work on polling which can miss activity if it occurs between polls. Instead of polling a tool could be developed that makes use of event APIs in the operating system to receive asynchronous notifications of network activity. Analysis and logging of these events will allow the tool to construct the complete real-time and historical network configuration of the host while the tool is running. The challenge will be showing that the active tool is safe to run on SCADA networks in such a way the adoption of the tool will occur.

Project objectives include identifying host operating system and performing tool development. Requirements for the project specify that the tool should use operating system APIs in order to receive asynchronous notifications of network activity. The tool must be able to log information about routable (not loopback) TCP/UDP connections, including the following items: Time, Process Name, Source Port, Destination Port, and Destination Address.

SCADA systems run on many different operating systems; this research focused on three of the major operating systems used by SCADA systems: Linux, WindowsXP, and Windows7. The research team found that Windows 7 offers many paths to accomplish the task, with two that have minimal impact on the system and should be seriously considered. First is the new Windows Event Logging API. Second, Windows 7 offers the ALE API within WFP. The research team recommends that any future work focus on these methods.

CONTENTS

FIGURES

TABLES

ACRONYMS

ALE	Application Layer Enforcement
API	Application Programming Interface
CEDS	Cybersecurity for Energy Delivery Systems
DOE-OE	Department of Energy Office of Electricity Delivery and Energy Reliability
DLL	Dynamic-Link Library
ICS	Industrial Control Systems
INL	Idaho National Laboratory
IO	Input/Output
IOCTL	IO Control
IP	Internet Protocol
IRP	IO Request Packet
NDIS	Network Device Interface Specification
OS	Operating System
R&D	Research and Development
RPC	Remote Procedure Calls
SCADA	Supervisory Control and Data Acquisition
TCP	Transmission Control Protocol
TDI	Transport Driver Interface
UDP	User Datagram Protocol
WFP	Windows Filtering Platform
WMI	Windows Management Interface

Host Event Based Network Monitoring

1. Host Event Based Network Monitoring Project

The INL Host Event Based Network Monitoring Project is funded through the Department of Energy's (DOE) Office of Electricity Delivery and Energy Reliability (OE) Cybersecurity for Energy Delivery Systems (CEDS) Research and Development (R&D) Program. The DOE program aims to enhance the reliability and resilience of the nation's energy infrastructure by reducing the risk of energy disruptions due to cyber-attacks. The project is a one year effort, started in FY 2012, awarded as part of the INL NSTB Core Frontier R&D tasks.

2. Background

Computer security focuses on network monitoring because a communication pathway must exist between the target machine and the "bad guys" for them to accomplish anything. Host based computer security tools such as "cports" and "netstat" monitor network connections on a Windows host by polling currently active network connections on the computer. This has several drawbacks. First, if malicious activity starts and stops between polls, the tools will fail to detect the activity entirely. Second, it is a wasteful use of resources to poll when no significant activity is occurring. It would be much better to process significant events as they occur and notify security software in real time. An event based network monitoring tool will not miss any activity because the OS will asynchronously notify the tool about network activity.

3. Scope and Technical Approach

Current host based network monitoring tools work on polling that can miss activity if it occurs between polls. Instead of polling a tool could be developed that makes use of event APIs in the operating system to receive asynchronous notifications of network activity. Analysis and logging of these events will allow the tool to construct the complete real-time and historical network configuration of the host while the tool is running.

The purpose of this research is to demonstrate the feasibility of a host event based network monitoring tool and the effects on host performance. The research focus is whether such a tool will be practical on a SCADA network and measuring industry reaction to an active tool.

The challenge is showing that the active tool is safe to run on SCADA networks in such a way that adoption of the tool could occur.

4. Objectives

The project's objectives included identification of a host operating system and tool development. The requirements for the project specified that the tool should use operating system APIs in order to receive asynchronous notifications of network activity and that the tool must be able to log information about routable (not loopback) TCP/UDP connections, including the following items: Time, Process Name, Source Port, Destination Port, and Destination Address.

5. Identifying Operating Systems

Based on the objectives for the project, a number of operating systems were considered in order to determine feasibility to accomplish the task.

5.1 Linux

Linux-based operating systems provide a low layer daemon that provides logging capabilities for a number of system calls called *auditd*. Auditd can be configured by editing the configuration files found in /etc/audit/ or by using the graphical interface *system-config-audit*. Once configured, the event

notifications are stored in a log file located at /var/log/audit/audit.log. Since digging through the log can be tedious, two tools were used to report on the findings: *aureport* and *ausearch*. During testing of linux operating systems, autitd was configured to report on any system call where a subsequent *socketcall* was issued. While the results from using this method were useful, this method did not fulfill all of the requirements. Basic port functionality was reported correctly, but some functions lacked in key information required, such as source and/or destination port. Additionally, auditd did not consistently report when ports closed.

Since auditd was unable to notify for all conditions identified in the requirements, it was determined that the auditd daemon would have to be modified in order to receive enough information concerning the required data items. Since linux is not widely used in SCADA systems and was only considered for proof-of-concept, linux was not chosen as the right target for this research.

5.2 Windows XP

The Windows XP operating system was first released on October 25, 2001. Since then it has become a common fixture in Industrial Control Systems (ICS). Even though Windows Vista and more recently Windows 7 have superseded it, XP still has a significant install-base on ICS networks, so it is logical to spend time developing software specific to this platform.

The XP operating system had three possible methods for monitoring network connections, and each will be considered in turn.

The first method is the Network Device Interface Specification (NDIS). A driver could be written that intercepts calls to network devices and generates events based on the observed packets. This has several drawbacks. It requires kernel code and would require filtering packets as they are sent/received by network interface cards. Generating events using this method would require parsing the raw packets (e.g. Ethernet frames) and performing the same work already done by the IP stack in the kernel. Essentially this would mean creating an additional, independent TCP/IP stack maintaining its own state about external connections.

Another possibility is to hook the Winsock.DLL and intercept calls to and from application programs using that DLL. This code would run in user space (a definite advantage), but could be bypassed by applications that create sockets without using Winsock.

The final possibility, and the one chosen by the project, was to take advantage of the Transport Driver Interface (TDI). The TDI is a generic API for interacting with various protocol transports (one of which is TCP/IP). Further, TDI allows the installation of new devices that become a part of the stack, e.g. calls coming from applications must go through any TDI filter devices before reaching their final destination. Figure 1 shows the normal relationship between user applications using Winsock.DLL and the drivers that provide socket services (AFD.SYS, TCPIP.SYS and ultimately the network interface card driver). AFD.SYS is the Ancillary Function Driver for Winsock (a TDI client) TCPIP.SYS implements the TCP/IP TDI provider. Also included in Figure 1 is NetBT.SYS, the NetBIOS over TCP/IP driver, which is another TDI client that calls TCPIP.SYS.

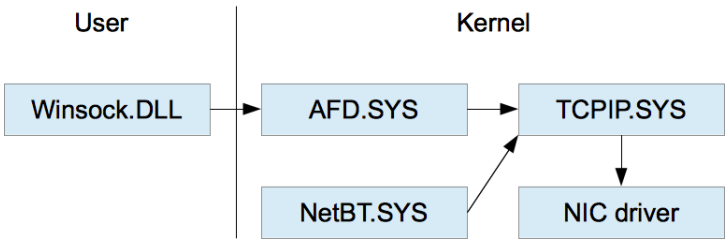

Figure 1. Normal relationship between user space (Winsock) and kernel drivers.

It is possible to install a TDI filter driver that intercepts all calls to a particular device and (optionally) forwards them along. The method for doing so is to attach to the device using the kernel routine: IoAttachDevice(). Once done, the logical configuration is that shown in Figure 2. Notice that all users of TCPIP.SYS will now route requests to the MyFilter driver.

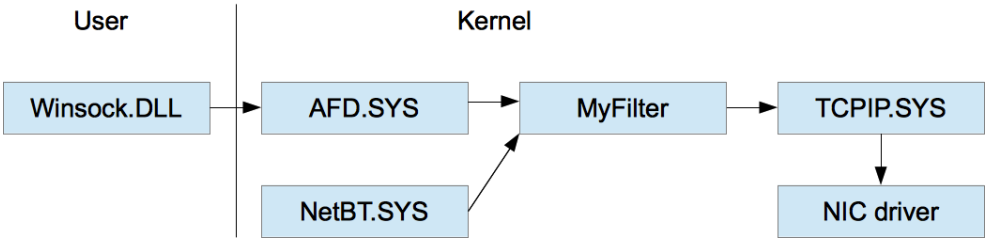

Figure 2. driver relationship with filter driver installed.

TDI layer is an abstraction of what transport drivers need to provide, so the problem then becomes translating the calls to it into TCP/IP events (socket creation, binding, connecting, closing, etc.). This process is poorly documented and required a significant amount of effort to figure out. Figure 3 shows the state transition diagram for TCP (raw sockets and UDP are greatly simplified state diagrams and, so, are omitted from this discussion).

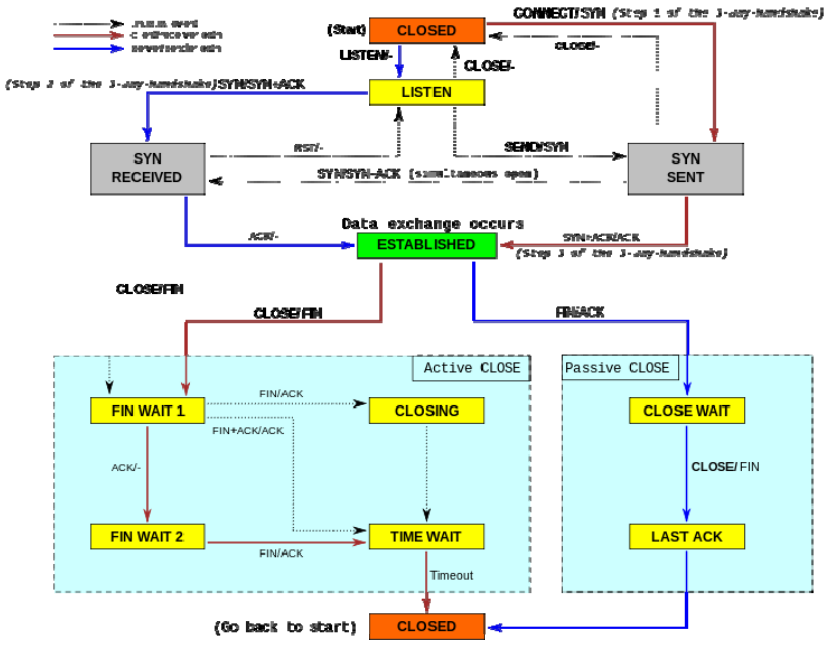

Figure 3. TCP state transition diagram.[a]

In the Windows kernel, input/output (IO) requests are sent between drivers as IRPs (IO Request Packets). For instance, AFD.SYS would generate an IRP in response to a user program request and send it up the stack to MyFilter (Figure 2) that will forward it along to TCPIP.SYS. There are several well known IRP's: IRP_MJ_CREATE (create a file-like object, e.g. a socket), IRP_CLEANUP/IRP_CLOSE (equivalent to closing a file or socket). TDI defines several IO control codes (IOCTLs) including: TDI_CONNECT, TDI_DISCONNECT, TDI_LISTEN, TDI_ACCEPT, etc.). MyFilter gets a chance to

[a] Diagram is licensed under the creative commons attribution-share alike 3.0 license and was obtained from http://commons.wikimedia.org/wiki/File:Tcp_state_diagram_fixed.svg

examine requests from AFD.SYS and the responses back from TCPIP.SYS and can thus maintain state about what the TCPIP stack is doing as it happens.

5.2.1 Client State Transitions

Table 1 shows client state transitions.

State Transition	POSIX function	Windows Kernel Calls
CLOSED/SYN SENT	socket(), connect()	IRP_MJ_CREATE(*address*) IRP_MJ_CREATE(*connection*) TDI_SET_EVENT_HANDLER(*address*, TDI_EVENT_DISCONNECT, *disconnfun*) TDI_ASSOCIATE(*address*, *conn*) TDI_CONNECT(*conn*)
SYN SENT ESTABLISHED	return from connect()	return from TDI_CONNECT()
ESTABLISHED FIN WAIT 1	shutdown(SHUT_RD)	TDI_DISCONNECT(connection)
ESTABLISHED CLOSE WAIT	read() == EOF	Call to *disconnfun*
LAST ACK CLOSE		TDI_DISASSOCIATE(*address*, *connection*) IRP_MJ_CLEANUP(*address*, *connection*) IRP_MJ_CLOSE(*address*, *connection*)
TIME WAIT CLOSE		same as above

Table 1. Client state transitions.

5.2.2 Server State Transitions

Table 2 shows server state transitions.

State Transition	POSIX function	Windows Kernel Calls
CLOSED LISTEN	socket(), bind(), listen()	IRP_MJ_CREATE(*address*) IRP_MJ_CREATE(*connection*) TDI_SET_EVENT_HANDLER(*address*, TDI_EVENT_CONNECT, *connfun*) TDI_ASSOCIATE(*address*, *conn*)
LISTEN SYN RECEIVED	accept()	call to *connfun*
SYN RECEIVED ESTABLISHED		TDI_ACCEPT in response to call to *connfun* TDI_ASSOCIATE(*address, connx*) TDI_SET_EVENT_HANDLER(connx, TDI_EVENT_DISCONNECT, *disconnfun*)
ESTABLISHED FIN WAIT 1	shutdown(SHUT_RD)	TDI_DISCONNECT(*connx*)
ESTABLISHED CLOSE WAIT	read() == EOF	Call to *disconnfun*
LAST ACK CLOSE		TDI_DISASSOCIATE(*address*, *connx*) IRP_MJ_CLEANUP(*address*, *connx*) IRP_MJ_CLOSE(*address*, *connx*)
TIME WAIT CLOSE		same as above

Table 2. Server state transitions.

5.2.3 Results

The current version of the driver produces Windows kernel debug messages in response to state changes in the TCP/IP stack. It is possible to modify the driver to pass these events to user process(es) via

the Windows Management Interface (WMI) call WdfWmifireEvent() API, the Windows Event Log call IoWriteErrorLogEntry(), or via a custom interface.

The time available to device drivers has a resolution of 10ms. This is the "tick" rate of the Windows kernel, so it is not easy to resolve time differences of less than 10ms. The CPU performance counters, which run at the full clock speed of the CPU, could possibly be leveraged if higher time resolution is necessary; however, given that the timeouts in TCP are measured in seconds, this kind of granularity should not be necessary.

5.2.4 Client Example

What follows are debugging messages sent when using the telnet.exe program to connect from 192.168.137.144 to 192.168.137.139 port 445. Notice that the driver has access to the full path to the executable and the current time (accurate to the tenth of a millisecond, approximately). First, a connect is sent from .144 to .139 port 445. The operating system dynamically assigns the port number 1068 to the request. After some time, the connection is disassociated and closed.

```
2012/12/06 23:21:34.328 STATE: \Device\HarddiskVolume1\WINDOWS\system32\telnet.exe SYN_SENT
192.168.137.139:445

2012/12/06 23:21:34.359 STATE: \Device\HarddiskVolume1\WINDOWS\system32\telnet.exe
ESTABLISHED 192.168.137.144:1068 -> 192.168.137.139:445

2012/12/06 23:21:40.046 STATE: \Device\HarddiskVolume1\WINDOWS\system32\telnet.exe FIN_WAIT
192.168.137.144:1068 -> 192.168.137.139:445

2012/12/06 23:21:40.125 STATE: \Device\HarddiskVolume1\WINDOWS\system32\telnet.exe
DISASSOCIATE 192.168.137.144:1068 -> 192.168.137.139:445

2012/12/06 23:21:40.171 STATE: \Device\HarddiskVolume1\WINDOWS\system32\telnet.exe CLOSING
192.168.137.144:1068 -> 192.168.137.139:445
```

5.2.5 Server Example

Below is the log for a simple server application. The socket being created on port 1337 on all interfaces can be seen in the first line. Next, a client connects from 193.168.137.139 port 1050 and the connection is established. After some time, the connection is closed.

```
2012/12/06 23:13:51.010 STATE: \Device\HarddiskVolume1\cygwin\home\Administrator\a.exe LISTEN
0.0.0.0:1337

2012/12/06 23:13:57.015 STATE: \Device\HarddiskVolume1\cygwin\home\Administrator\a.exe
SYN_RCVD 192.168.137.139:1050

2012/12/06 23:13:57.046 STATE: \Device\HarddiskVolume1\cygwin\home\Administrator\a.exe
ESTABLISHED 0.0.0.0:1337 -> 192.168.137.139:1050

2012/12/06 23:13:57.468 STATE: \Device\HarddiskVolume1\cygwin\home\Administrator\a.exe
ASSOCIATE 0.0.0.0:1337 -> 192.168.137.139:1050

2012/12/06 23:13:59.937 STATE: \Device\HarddiskVolume1\cygwin\home\Administrator\a.exe
CLOSE_WAIT 0.0.0.0:1337 -> 192.168.137.139:1050
```

5.2.6 Future Work

The current version of the driver uses Windows debugging messages to report TCP/IP state changes. This is not ideal. Instead, one of three approaches should be used for conveying this information to a user-space application.

The first method for getting events from the Windows kernel to user-space is through the event log. It should be possible to use the IoAllocateErrorLogEntry() call to package up the state transition information and send it to the event log where an application can monitor the events in almost real time.

Another possibility is to take advantage of the WMI interface. In the case of WMI, programs subscribe to events generated by the driver. The IoWMIFireEvent() call can be used to send notifications of state changes to the subscribers.

Finally, a custom device driver interface could be written. In this case, a user-space program opens and reads the device. Events are passed as data from the kernel through this custom device driver. While this method has the greatest flexibility, it is more complex to implement. Both the event log and WMI are easier to implement.

5.2.7 Problems and Issues

The primary disadvantage of developing Windows kernel code is that mistakes made in this code (referencing memory out of bounds, incorrect forwarding of IRPs, etc.) result in the dreaded "blue screen of death." Given the number of configurations possible within the Windows networking stack (VPN, shared connections, IPv6, etc.), verification of correctly written code (and coping with bugs outside the control of the device driver) is difficult; however, there is no other way to gather all TCP/IP events on the Windows XP family of operating systems (Windows XP, Server 2000, Server 2003, etc.). So while this approach is not ideal there is really no other viable choice. Months of testing would need to be done in a safe ICS environment before it could be used in a live control system environment.

5.3 Windows 7

Windows 7 has several possible methods for monitoring network connections, but only a select few will be considered in turn.

5.3.1 Windows Event Log API

The first method is to utilize the new Windows 7 network audit events. Beginning with Windows Vista, a new Windows Event Log API is available that added thousands of new available events. There are a number of different ways to retrieve these events, but for time considerations the project focused on consuming events. When consuming events, it is possible to retrieve events from a live event channel or an event log file. From the Windows Event API, a call is made to the function EvtSubscribe() with the ChannelPath set to Security. This allows the program to receive all events under the Windows Security Log. Since this logs many non-essential alerts, such as when a user is logged on, a subscription can be made to specific events by using an XPath query. It was found that the event IDs 5154 and 5158 contained the information needed in the requirements.

A program was created to query for these events and process the events to retrieve the items described in the requirements. A log file is generated that provides these details (as well as a few additional details) that provide a real-time and historical network configuration of the host. The one feature lacking is the ability to notify when a port is closed on a host. According to Windows 7 documentation, there are no network audit events that signal when a port is closing or is being released. Additionally, using the network audit event, it appears not to capture some system ports and services that are started during boot time. As far as the research has determined, normally started processes and applications are logged using network audit events. Since some system ports and services aren't identified at boot time by the network audit events, it is also plausible that if the host were infected with malware, this type of tool would not be able to identify some ports and services started by rootkits.

It was also identified that further analysis could be done using network audit events to determine how much traffic is being sent per port, but this further analysis could not be done live. Because of the nature and type of event that is generated, live analysis is not possible, but a report could be generated on demand to determine port usage.

It was determined that using the Windows Event Log API could be a viable means to accomplishing a host based network event monitoring system. If the program were run on a host, the overhead generated would be significantly less than other means. Although control systems generate a log of network traffic and consume a fair amount of CPU resources, using the Windows Event Log API would not impact the host operating system in a negative way. If it were determined by a utility to be too much overhead, the possibility exists to offload all events generated on a host to another host on the network. This would minimize the CPU overhead used to consume network events to a central machine which would then be used as means to monitor all hosts on a control network. Paired with the excessive traffic generated by a control system, this may be too much traffic for a single network interface card, but another could be installed and all event traffic offloaded onto the secondary network card.

5.3.2 Windows Filtering Platform

During development using network audit event, it became apparent that the Windows Filtering Platform (WFP) may be a better alternative. The Windows Filtering Platform is a new architecture in Windows Vista and Window 7 that offers a means to filter and modify TCP/IP packets, monitor or authorize connections, filter IPsec-protected traffic, and filter remote procedure calls (RPCs). The filtering and modifying TCP/IP packets allows a program to examine or modify incoming and outgoing packets before any additional processing occurs. WFP was created to replace the old methods used in Windows XP, namely the TDI and NDIS APIs. WFP works similarly to the process described under Windows XP, but it is more robust, as shown in Figure 4.

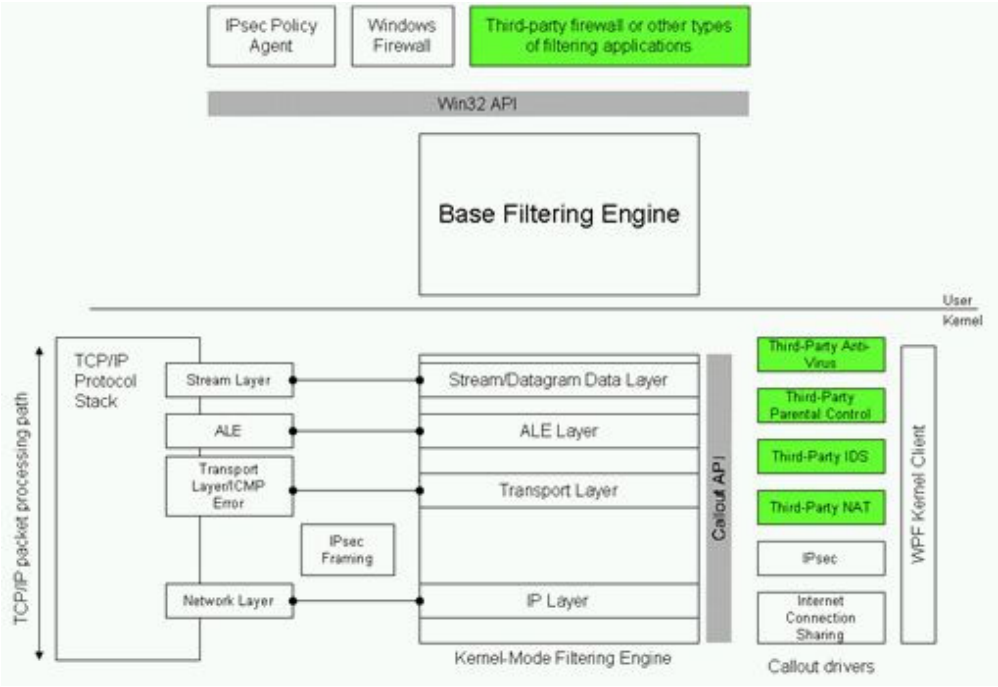

Figure 4. Achitecture of WFP and its extensibility.[b]

5.3.3 Application Layer Enforcement

During the research for using WFP, the research team found that the Microsoft documentation states that there are no layers within WFP that would alert on port termination. However, within WFP there are a set of kernel-mode layers that are used for stateful filtering, the Application Layer Enforcement (ALE) layers. By using ALE, WFP will keep track of the state of network connections and will notify on any of

[b] Graphic used from http://msdn.microsoft.com/en-us/library/windows/hardware/gg463267.aspx

the network states we choose. Using the ALE layers within WFP, a program was created that filtered on the following port states: RESOURCE_ASSIGNMENT, AUTH_LISTEN, AUTH_RECV_ACCEPT, AUTH_CONNECT, FLOW_ESTABLISHED, RESOURCE_RELEASE, ENDPOINT_CLOSURE, CONNECT_REDIRECT, and BIND_REDIRECT. Using this method, the program is directly inserted between the kernel and the traffic generated on the port. Everything sent via IPv4 and IPv6 would have to go through the ALE filters. By design, a program is able to reject or drop packets or connections when using WFP and ALE. Our program was written not to enable this feature, and instead allowed all traffic to flow though it, rejecting zero packets.

This method allowed us to log all items described in the requirements, including when ports or processes were closed and terminated. The overhead needed to run this task was minimal and could easily be used in a control system environment. Process and hard-drive utilization could even be further reduced by using the same logging methodologies discussed in the Windows XP section above instead of using the expensive debugger logging method. WFP with ALE could be used on control system hosts, and then logging and further processing could be offloaded to another machine that would monitor all hosts on the control system network.

5.3.4 Problems and Issues

A positive note for using this method is that even the system processes during boot time would have to go through these layers and would be logged. A downside is that in order to use the ALE filters within WFP, there are two requirements. First, since you are in between the kernel and the generated port traffic, you must either allow a packet to pass through the ALE layers or deny the packet delivery (some may say this is too risky. Second, in using ALE layers within WFP you are creating a Windows kernel mode driver that may need to be signed.

6. Conclusion

SCADA systems run on many different operating systems and this research focused on three major operating systems: Linux, WindowsXP, and Windows7. Linux is currently used in a minority of cases for control systems and already has kernel programs that would enable some capability for host event based network monitoring. Further modification of the auditd daemon would be needed in order to capture all events identified in the requirements.

Windows XP did not offer any network event notification, unlike Linux or Windows 7. Windows XP offered three other possibilities; two were undesirable, but the last was promising. Using TDI in Windows XP, a program was created that was able to capture all events and details identified in the requirements. While the creation of a host event based network monitoring tool for Windows XP was successful, it has too many problems and drawbacks. The TDI interface injects the program far within the Windows kernel, and any problems with code in the program will lead to the Blue Screen of Death. Months of testing on a particular SCADA system would be needed before it could possibly be used on a production system, and many asset owners would probably reject the tool in the end.

Windows 7 offers many paths to accomplish the task, with two that should be seriously considered. Both of these methods identified during the research were found to use minimal system processes, ranging from fractions of a percentage to 5 percent utilization of the CPU. First is the new Windows Event Logging API. This newer version of the API allows for a host to create an event, much like the Linux auditd daemon, for specific network events. Using this method it is possible to capture all of the required details and events, except for when a port is closed. This method also failed to capture the opening of certain system services during boot time, a critical time slice if a rootkit infection exists on the machine. Although these are drawbacks, a definite benefit is that offloading of the processing of events can be done. Each host within a SCADA system could be configured to send certain network events to a single computer used only for monitoring the hosts in the SCADA system. This single computer would

receive and process all network events from all SCADA hosts and could display the “network health” of each computer.

Second, Windows 7 also offers the ALE API within WFP. Using ALE it is possible to capture all network events and details indentified in the requirements. ALE allows for the capture of the following port events: RESOURCE_ASSIGNMENT, AUTH_LISTEN, AUTH_RECV_ACCEPT, AUTH_CONNECT, FLOW_ESTABLISHED, RESOURCE_RELEASE, ENDPOINT_CLOSURE, CONNECT_REDIRECT, and BIND_REDIRECT. The WFP API allows the creation of new events that are processed through the Windows Event Logging API. Using this method, it should be able to offload these new events to another host for processing, as described above. This would be the desired goal, and any future work should focus on these methods.