Chapter 5

AN INTEGRATED SYSTEM FOR INSIDER THREAT DETECTION

Daniel Ray and Phillip Bradford

Abstract This paper describes a proof-of-concept system for detecting insider threats. The system measures insider behavior by observing a user's processes and threads, information about user mode and kernel mode time, network interface statistics, etc. The system is built using Microsoft's Windows Management Instrumentation (WMI) implementation of the Web Based Enterprise Management (WBEM) standards. It facilitates the selection and storage of potential digital evidence based on anomalous user behavior with minimal administrative input.

Keywords: Insider threats, anomaly detection, proactive forensics

1. Introduction

Insider threats are "menaces to computer security as a result of unauthorized system misuses by stakeholders of an organization" [3]. A substantial percentage of reported computer crime incidents are perpetrated by insiders [6]. Insider attacks are problematic because they are difficult to detect, and because insiders are likely to have intimate knowledge about potential targets as well as physical and logical access to the targets. Insider threat detection systems seek to discover attacks perpetrated by organization insiders.

Our insider threat model [2] analyzes computer use from a behavioral perspective such as data about the programs that employees use on a daily basis and the programs' underlying processes. Our research suggests that illicit activities are indicated by variations from a statistical norm (anomaly detection). Moreover, to be effective, an insider threat detection system must remotely and unobtrusively gather aggregate information about user workstation activity for analysis.

The generalized statistical approach to detecting intrusions was first outlined by Denning [5]. Research on statistical intrusion detection is described in [7] and in several survey articles [1, 10, 13, 14]. Significant efforts include SRI's EMERALD [19], a distributed system for intrusion detection in large-scale networks, and the use of data mining for dynamic intrusion detection by Lee, *et al.* [12]. More recently, Kahai and colleagues [11] proposed a system for profiling external attacks while proactively storing forensic information in the event of compromise. Our insider threat detection approach draws from research in intrusion detection and the sequential hypothesis testing approach to proactive forensics [2, 3].

Our proof-of-concept insider threat system makes use of the Web Based Enterprise Management (WBEM) standards. The WBEM initiative was created to address the difficulty of dynamically managing distributed IT environments; its goal was to develop a set of standards that unified the management of diverse systems [4]. The idea was that companies would embrace the WBEM initiative and develop systems conforming with WBEM standards, making it possible for a single application to manage all the components of an IT infrastructure. Microsoft adopted the WBEM standards in its Windows Management Instrumentation (WMI) architecture. WMI inherits classes defined according to the WBEM standards, which makes management data available in structured form.

This paper discusses how development tools targeted for system administration (based on WBEM/WMI) can serve as the foundation for a real-time insider threat detection system. Our approach is fundamentally different from traditional anomaly-based intrusion detection systems. First, our system only requires that additional resources be focused on anomalous users; this is different from systems that require immediate action upon receiving threshold-based alerts (of course, our system can act on threshold-based alerts, if necessary). Second, our system requires security administrators to be trusted; these administrators are hard to monitor as they may be aware of the details that are being measured. Finally, our system is built using standard embedded functionalities.

2. Common Information Model

The WBEM initiative led to the creation of the Common Information Model (CIM) standard, which models management data in an object-oriented fashion [4]. The CIM standard provides a method for expressing useful management data for computer systems. It does not enforce a

scheme for naming classes or specific properties, methods, etc., that are to be included in classes. Rather, it is "a well-defined collection of class definitions that all (participating) companies agree to follow" [4].

The CIM schema provides a logical hierarchical structure of class definitions. For instance, the base class `CIM_LogicalElement` is a management class from which `CIM_Process` instances are derived. Each `CIM_Process` instance contains information about a particular process such as minimum and maximum working set size, process ID, execution path, etc. The class `CIM_LogicalFile` derives `CIM_Directory`, whose instances contain information about directories, and `CIM_DataFile`, whose instances contain information about operating system metadata files. CIM has hundreds of classes [15], which make it attractive for modeling information involved in managing computer systems.

The key to leveraging management information for insider threat detection is to dynamically select, aggregate and store all data that could be useful during an investigation (all of this is supported by WBEM). CIM and its implementations (e.g., WMI) also provide aggregated statistics such as detailed performance information about active processes [15] for responding to certain system events.

3. Windows Management Instrumentation

Windows Management Instrumentation (WMI) is a "technology built into Windows that enables organizations to manage servers and user PCs connected to their networks" [20]. The WMI architecture has three layers.

- Layer 1: Managed Objects and Providers
- Layer 2: WMI Infrastructure
- Layer 3: Management Applications

In the WMI nomenclature, "producers" provide information that is used by "consumers" [17]. The information produced is stored in a set of data structures called the WMI repository. The managed objects and providers of Layer 1 represent the WMI producers. Managed objects are physical or logical components in the environment that are being managed. Providers are responsible for monitoring the managed objects and passing real-time information about the objects to the WMI infrastructure. Each managed object is represented by a WMI class. New classes may be derived to manage additional objects that are not managed by default.

Layer 2 represents the WMI infrastructure provided by Windows. The WMI infrastructure, which serves as a bridge between WMI producers

and consumers, has two main components, the WMI repository for storing information and the WMI service, which is responsible for providing data to management applications.

Layer 3 constitutes applications that consume WMI information from the WMI repository (e.g., our insider threat detection system). The `System.Management` namespace of the .NET framework provides the classes needed to interface with WMI in order to build such applications.

Our insider threat detection application employs WMI components called `PerformanceCounter` classes along with WMI event handlers. In the following, we briefly discuss the design and use of these components via the .NET framework.

WMI allows remote access to instrumentation on another computer via its namespace scheme. Code provided in [8] demonstrates the instantiation of a `ConnectionOptions` class, which is used to store the username and password of an administrator of a networked machine.

Class properties define the characteristics of the real-world object that the class represents. For example, consider the `Win32_Process` class derived from `CIM_Process` mentioned above. Each instance of the class represents a different process and each instance has 45 properties that describe the corresponding process. WMI properties may be queried using the WMI Query Language (WQL), a subset of SQL92 [9]. WQL provides a syntax for Select statements but not for updating or deleting WMI information [17].

Figure 1 (adapted from [8]) presents a sample WQL query that accesses class properties. First, the `ManagementScope` class is instantiated, the scope is set to the `CIMV2` namespace on the local machine, and the `ManagementObjectSearcher` class is instantiated with the appropriate arguments. Next, the `Get` method of class `ManagementObjectSearcher` is invoked; this returns the collection of objects that satisfies the WQL query. The collection of objects is stored in an instantiation of the `ManagementObjectCollection` class. The objects stored in the collection are searched serially, the `Description` and `ProcessId` properties of the returned objects are referenced by name, and their values are converted to strings.

4. WMI Techniques and Threat Detection

This section introduces advanced WMI techniques used to implement insider threat detection. We first focus on accessing performance counter classes, which provide statistical information about the state of computer systems (e.g., average processor load, average network traffic, etc.). Such statistics offer clues to uncovering insider attacks.

```
string queryString
   = ''SELECT Description, ProcessId FROM Win32_Process'';
string scopeStr = @''\\.\root\CIMV2'';
SelectQuery query = new SelectQuery(queryString);
ManagementScope scope
   = new System.Management.ManagementScope(scopeStr);
ManagementObjectSearcher searcher
   = new ManagementObjectSearcher(scope, query);
ManagementObjectCollection processes = searcher.Get();
foreach(ManagementObject mo in processes)
{
   // Handle getting data from management object here
   string teststring1 = mo[''Description''].ToString();
   string teststring2 = mo[''ProcessId''].ToString();
}
```

Figure 1. C# example of a WQL query.

Next, we discuss the management of WMI events. These events can be set to automatically alert a listening application (e.g., an insider threat detection system) when certain criteria are met. We also introduce methods for creating custom performance counters and event triggers.

4.1 Accessing Performance Counters

WMI's "Performance Counter Classes" reveal real-time, Windows native statistics about the state of computer systems. For example, `Win32_PerfFormattedData_PerfOS_System` provides statistical and aggregated information about the operating system, including file read and write operations per second, number of processes or threads, and number of system calls per second.

```
PerformanceCounter cpuCounter = new PerformanceCounter
   (''Processor'', ''% Processor Time'', ''_Total'');
int value = cpuCounter.NextValue();
```

Figure 2. Obtaining performance counter data.

`System.Diagnostics` is the principal .NET namespace for handling WMI performance counters. Figure 2 (from [16]) shows how an object of the `PerformanceCounter` class (member of the `System.Diagnostics` namespace) is used to obtain WMI performance counter information.

In Figure 2, the `PerformanceCounter` class takes as parameters the category of performance data, the particular performance category class, and the handle of a particular instance of this class (in this example, the keyword `_Total`) to aggregate all processes. A `PerformanceCounter` object is returned. The `NextValue` method can be called on this object to retrieve the current statistic represented by the object.

```
bool pleaseContinue = true
public void doit()
{
   int processID =
      System.Diagnostics.Process.GetCurrentProcess().Id;
   int workingSet = 30000000;
   string wqlQuery = String.Format(
      @‘‘Select * FROM __InstanceModificationEvent WITHIN 1
      WHERE TargetInstance ISA ‘Win32_Process’ AND
      TargetInstance.ProcessId = {0} AND
      TargetInstance.WorkingSetSize >= {1} AND
      PreviousInstance.WorkingSetSize < {2} ’’,
   processId, workingSet, workingSet);
   WqlEventQuery query = new WqlEventQuery(wqlQuery);
   ManagementEventWatcher watcher =
      new ManagementEventWatcher(query);’
   watcher.EventArrived +=
      new EventArrivedEventHandler(onEvent);
   watcher.Start();
   ArrayList array = new ArrayList();
   while(pleaseContinue){
      array.Add(1);
      if(i%1000 == 0) System.Threading.Thread.Sleep(1);
}

public void onEvent(object sender, EventArrivedEventArgs e)
{
   pleaseContinue = false;
   Console.WriteLine(‘‘Found a misbehaving process’’);
}
```

Figure 3. Dealing with WMI events.

4.2 Event Handlers

WMI event handlers define and consume events related to data handled by WMI. The code in Figure 3 (from [16]) shows how a .NET application can specify the events it is interested in capturing. The internal WMI event `__InstanceModificationEvent` fires whenever a value

in the namespace is updated. The WQL statement selects instances from the set of stored WMI events where a value in the namespace `Win32_Process` changes, the `ProcessId` is 1367, and the working set size is more than 30,000,000. The code then creates an event `watcher` object whose `EventArrived` method is invoked when such an event occurs. The code dictates that a .NET event handler should be initiated so that the specific code for handling an event can be managed in a different `onEvent` function according to .NET conventions. In this case, the event is handled by printing ``‘‘Found a misbehaving process’’`` and exiting the loop.

This technology is powerful and has "out of the box" functionality that provides useful statistics about system operations to any application interfacing with WMI. Numerous possibilities exist for leveraging these statistics to detect insider threats.

4.3 Custom Performance Counters and Events

WMI enables programmers to create custom performance counters and to expose custom events to WMI. Thus, WMI can generate and store highly specialized data at individual workstations rather than relying on complicated WQL statements and statistical calculations at a centralized server.

Creating custom performance counters involves defining the metadata about the new performance counter, giving it proper types, etc. The `System.Diagnostics.PerfomanceCounter` class that was used to read in performance counter information [16] may be used to create custom performance counters for WMI. To prevent naming collisions, the `Exists` method must be invoked to check if a performance counter with the same name already exists. If not, a new performance counter is created using the `Create` method.

The code in Figure 4 (from [16]) implements this functionality. It creates a `CounterCreationDataCollection` object and populates it with `CounterCreationData` instances. The performance counter merely indicates the number of 7's in an integer parameter to a web service. A `PerformanceCounterType` is assigned to each performance counter that is created in order to determine how the `NextValue` method works. Several statistics are supported (average, difference, instantaneous, percentage and rate), but the type chosen (`NumberOfItems32`) simply returns the number of items [16]. `PerformanceCounterCategory.Create` then adds the counter to the WMI CIM repository.

```
// Check if the category already exists or not.
if (!PerformanceCounterCategory.Exists(categoryName))
{
  CounterCreationDataCollection creationData =
     new CounterCreationDataCollection();

  // Create two custom counter objects
  creationData.Add(new CounterCreationData(``Number of 7s - Last'',
     ``Number of occurences of the number 7 in the last WS call'',
     PerformanceCounterType.NumberOfItems32));

  // Bind the counters to a PerformanceCounterCategory
  PerformanceCounterCategory myCategory =
     PerformanceCounterCategory.Create(``TestWS Monitor'',
     helpInfo, creationData);
}
```

Figure 4. Creating custom performance counters.

The code in Figure 5 (from [16]) may be used to read from a custom performance counter.

```
PerformanceCounter counter = new PerformanceCounter
   (``TestWS Monitor'', ``String Length - Last'', false);
counter.RawValue = count;
```

Figure 5. Reading from a custom performance counter.

The code in Figure 6 (from [16]) shows how to make extensions to WMI CIM events that can be consumed by event consumer applications. It uses several .NET namespaces described in [16].

The `CustomEvent` class, which is created in Figure 6, extends the `System.Management.Instrumentation.BaseEvent` class. Such an extended class could contain any properties and methods deemed appropriate to its task [16]. Calling the event is as easy as creating a `CustomEvent` instance and invoking the `Fire` method inherited from `BaseEvent`.

5. Insider Threat Detection System

Our insider threat detection system is designed to be an asset to systems administrators. It leverages system management tools to obtain system information; it archives the information and compiles it into useful information about expected versus actual user behavior. It alerts administrators to take action when anomalous user activity is indicated.

```
// Specify which namespace the Management Event class is created
// in [assembly:Instrumented(‘‘Root/Default’’)]

// Let the system know you will run installUtil.exe against this
// assembly -- add reference to System.Configuration.Install.dll
// [System.ComponentModel.RunInstaller(true)]

public class CustomEventInstaller:DefaultManagementProjectInstaller{}

namespace WorkingSetMonitor
{
   // Event class: Renamed in the CIM using ManagedNameAttribute
   // [ManagedName(’’SuperEvent’’)]
   public class CustomEvent : BaseEvent
   {
      string data; // Not exposed to the CIM -- its primitive
      int state;   // Exposed via public property
      public CustomEvent(string data)
      {
         this.data = data
      }

      public int State
      {
         get { return state; }
         set { state = value; }
      }
   }
}
```

Figure 6. Creating custom events.

Moreover, it provides this functionality without placing any demands on systems administrators.

The insider threat detection system has a three-tier architecture. The first tier is the functionality provided by WMI. The second is the server-side insider threat detection application, which uses C# code to interface remotely with WMI. This application also performs statistical analyses on raw WMI data; the computations are performed on a central server to determine if aberrant user behavior is occurring. The third tier is a database that stores raw WMI data and the results of statistical analysis.

The database is required to store historical user behavior and to demonstrate that a threat actually exists. This is because insider threat detection relies on statistics that compare current user behavior with past behaviors.

6. WMI Service Expectations

The WMI repository must have up-to-date data; therefore, the time interval at which data is updated is an important issue. The more frequent the updates, the more likely the recorded data will accurately portray the current system state. An equally important issue is the speed at with WQL queries are fielded. This section discusses whether or not WMI can offer service guarantees on providing accurate, up-to-date information.

Because WQL is a subset of SQL92, optimizing the execution of the WQL Select statement falls in line with research for optimizing SQL statements in general [18]. More importantly, it is necessary to arrive at reasonable expectations of the speed of execution of queries and updates to the WMI repository.

Tunstall and Cole [20] suggest that it is possible to provide certain service guarantees. They discuss the notion of "high performance classes" that provide real-time information about current activity. Special registry keys are used to provide fast access to information on remote computers. The high performance classes built into Windows are exposed by WMI's performance counter classes. As explained in [20], a refresher object can be called to quickly make real-time data available to a WMI application. But "quickly" and "real-time" are relative terms that depend on the functioning of Windows internals, which is not public knowledge.

However, experiments can be conducted to obtain assessments of WMI service guarantees. Our insider threat detection system was interfaced with the WMI repository and a local SQL database; Microsoft SQL Server 2005 was attached to a local network. In our experiment, a simple remote WQL query was issued 1,000 times in succession to another machine on the same laboratory subnet, and the results were stored in a database housed on a server located elsewhere in the building. The query requested the remote machine for its local time (this type of query is consistent with querying for information about processes).

Our analysis showed that each operation set consisting of querying the remote machine, receiving a reply and writing to the database took 35.01 milliseconds on the average. We believe this meets moderate service expectations. Our future research will investigate the handling of forensic evidence that is more ephemeral in nature.

7. Conclusions

The insider threat detection system described in this paper is intended to operate as a watchdog within an enterprise environment, remotely and covertly gathering data from user workstations. The design lever-

ages several technologies that were originally designed to expose important enterprise management information to IT administration personnel. These include system-independent Web Based Enterprise Management (WBEM) schemas and a system-dependent implementation of Microsoft Windows Management Instrumentation (WMI).

It is important to note that the data exposed by WMI (and WBEM implementations for other operating systems) conveys user activity on individual workstations. Moreover, these implementations are ubiquitous and their activities are abstracted from the normal user by the operating system.

Our insider threat detection system facilitates the selection and storage of potential digital evidence based on anomalous user behavior with minimal administrative input. In particular, it leverages WMI to support remote access, expose current information about user activity, provide basic aggregated statistical information and event handling, and support custom event handling and statistical aggregation for applications that access WMI via the .NET framework.

References

[1] S. Axelsson, Intrusion Detection Systems: A Survey and Taxonomy, Technical Report 99-15, Department of Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 2000.

[2] P. Bradford, M. Brown, J. Perdue and B. Self, Towards proactive computer-system forensics, *Proceedings of the International Conference on Information Technology: Coding and Computing*, vol. 2, pp. 648–652, 2004.

[3] P. Bradford and N. Hu, A layered approach to insider threat detection and proactive forensics, *Proceedings of the Twenty-First Annual Computer Security Applications Conference (Technology Blitz)*, 2005.

[4] J. Cooperstein, Windows management instrumentation: Administering Windows and applications across your enterprise, *MSDN Magazine* (msdn.microsoft.com/msdnmag/issues/0500/wmiover), May 2000.

[5] D. Denning, An intrusion-detection model, *IEEE Transactions on Software Engineering*, vol. 13(2), pp. 222–232, 1987.

[6] J. Evers, Computer crime costs $67 billion FBI says, CNET News .com, January 19, 2006.

[7] M. Gerken, Statistical-based intrusion detection, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania (www.sei.cmu.edu/str/descriptions/sbid.htm).

[8] K. Goss, WMI made easy for C#, C# Help (www.csharphelp.com /archives2/archive334.html).

[9] C. Hobbs, *A Practical Approach to WBEM/CIM Management*, Auerbach/CRC Press, Boca Raton, Florida, 2004.

[10] A. Jones and R. Sielken, Computer System Intrusion Detection: A Survey, Technical Report, Department of Computer Science, University of Virginia, Charlottesville, Virginia, 2000.

[11] P. Kahai, M. Srinivasan, K. Namuduri and R. Pendse, Forensic profiling system, in *Advances in Digital Forensics*, M. Pollitt and S. Shenoi (Eds.), Springer, New York, pp. 153–164, 2005.

[12] W. Lee, S. Stolfo and K. Mok, A data mining framework for building intrusion detection models, *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 120–132, 1999.

[13] T. Lunt, Automated audit trail analysis and intrustion detection: A survey, *Proceedings of the Eleventh National Computer Security Conference*, 1988.

[14] T. Lunt, A survey of intrusion detection techniques, *Computers and Security*, vol. 12(4), pp. 405–418, 1993.

[15] Microsoft Corporation, WMI classes (msdn2.microsoft.com/en-us /library/aa394554.aspx), 2006.

[16] J. Murphy, A quick introduction to WMI from .NET, O'Reilly Network (www.ondotnet.com/pub/a/dotnet/2003/04/07/wmi.html), 2003.

[17] K. Salchner, An in-depth look at WMI and instrumentation, DeveloperLand (www.developerland.com/DotNet/Enterprise/145.aspx), 2004.

[18] L. Snow, Optimizing management queries, *.NET Developer's Journal* (dotnet.sys-con.com/read/38914.htm), July 21, 2003.

[19] SRI International, Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) (www.csl.sri.com/projects/emerald).

[20] C. Tunstall and G. Cole, *Developing WMI Solutions*, Pearson Education, Boston, Massachusetts, 2002.