WinResMon: A Tool for Discovering Software Dependencies, Configuration and Requirements in Microsoft Windows

Rajiv Ramnath – National University of Singapore
Sufatrio – Temasek Laboratories, National University of Singapore
Roland H. C. Yap and Wu Yongzheng – National University of Singapore

ABSTRACT

This paper describes WinResMon, a system tool for determining resource usage and interactions among programs in Microsoft Windows environments. Think of WinResMon as a debugging tool to assist with software maintenance in a Microsoft Windows environment. It shows the current system state in terms of how resources are used and explains how the system arrived at that state. WinResMon can be used to determine how a program uses the registry and which files are needed by that program. WinResMon differs from other systems/tools in that it is integrated, designed to answer queries about resource usage and dependencies over time, and extensible, allowing the addition of new functions and tools.

Introduction

The tasks of software maintenance and configuration require a precise understanding of system resources, the individual requirements of each piece of software, and interdependencies between every software program on the system. We use the term *software maintenance* to describe the system administration task of ensuring that software on a system is configured and maintained correctly over time. Examples of software dependencies include:

- file sharing: including dynamic libraries and external data storage
- sharing of software configurations: usually in the form of registry keys in Microsoft Windows
- interprocess communication and synchronization.

Although software maintenance tasks might seem conceptually trivial, they can be time consuming and difficult, especially in large or complex environments. System administrators often rely on documentation and on-line information such as FAQs or forums, but such information is often incomplete.

In various distributions of Linux, software dependency issues are partly addressed by the use of a package manager. The Red Hat Package Manager (RPM) [1], for example, records the currently installed packages and the files required and provided by these packages in a centralized database. RPM checks dependencies before removing a package to ensure that files required by other installed packages are not removed. Similar checks are done to prevent installing a package which contains files that conflicts with other existing packages.

In Microsoft Windows, however, software installation can be complex, and the exact dependencies between different software programs might not be clear. The confusion is further compounded by many implicit software interdependencies, e.g., registry keys which are part of shared software configurations. As a result, it is difficult to know whether to remove a file when uninstalling an application. File removal might lead to problems with another piece software or might create security vulnerabilities.

When installing two or more programs that share files, one program may cease to function correctly because the installation of the second blindly overwrote shared libraries (DLL files). There is also the question of when to perform a major software upgrade. System administrators may delay upgrading for fear of breaking existing software; yet such a choice has its own risks.

This paper focuses on the problem of software maintenance in Microsoft Windows NT-based operating systems (Microsoft Windows XP, Microsoft Windows 2000, Microsoft Windows 2003).[1] We present *WinResMon*, a discovery and system debugging tool for determining a program's resource usage as well as the resource usage interactions between multiple programs.

Motivation and Applications

We believe that the key to solving the software maintenance problem is to understand the life cycle of the system and programs therein. We also wish to empower ordinary users, removing the requirement of knowing every minutiae of Microsoft Windows. Although WinResMon is not tailored specially for system security, it can also be utilized as a security auditing tool.

We designed WinResMon to act as both an infrastructure or framework and a system utility. As a

[1] While an appropriate version of a tool similar to WinResMon could also be of use in UNIX, its value is much greater in a Microsoft Windows environment.

framework, it is extensible, and one can therefore add new functionality and build customized tools. As a tool, it comes with pre-built modules to answer typical questions about resource usage and dependencies.

When used as a debugger, WinResMon investigates the current system state, i.e., which program uses which registry keys, and determines how the system has arrived at that state. WinResMon accomplishes this by recording information about the evolution of system software dependencies and resource usage over time. To solve general problems in software maintenance, WinResMon monitors: files, the registry, and interprocess communication and synchronization. However, it is not feasible to continuously and permanently record all changes to the system since the required space would be prohibitive. WinResMon employs a reasonable compromise by maintaining detailed usage records over the current time period and a subset of information that can be maintained over the lifetime of all software in the system.

We illustrate the software maintenance problem with some simple examples. One attack vector for spyware is to register itself as a start-up program, thereby hiding itself from the end user. Also consider a music player which may require some sound decoding libraries. This application only functions correctly with certain versions of the libraries. Thus, replacing a library can lead to software failure. Various pieces of software may also conflict, e.g., two mail transfer agents (MTAs) usually do not co-exist.

Some common system administration questions and tasks which WinResMon can assist with include:

1. ***Can we safely remove a particular DLL file?***
   Some applications provide shared libraries (DLL files) for use with other applications. When the system administrator uninstalls an application, she can also choose to remove the DLLs. Removing a DLL can cause other programs which still use it to malfunction. On the other hand, blindly retaining all DLLs will cause the system to keep growing and may create security vulnerabilities. The system administrator generally lacks adequate information to determine whether another program uses a shared library. WinResMon can be used to record the utilization of each DLL so that the system administrator can determine which programs use which DLLs.
2. ***Why does a program need administrator privilege to run?***
   Running programs with administrator privilege is discouraged because malware such as viruses/spyware or poorly written applications can damage the system. However, some programs may need to run as the administrator without an obvious reason. WinResMon can detect whether a program needs administrator access. The idea is to understand the reasons for elevated privileges and configure the system to limit the use of privileges. If it finds applications that require certain administrator privileges to function correctly, the system administrator can set up a policy that restricts the administrator privileges to the needed resources (files, registry keys, etc.). We remark that this approach can be contrasted with confinement systems such as systrace [2] in UNIX. WinResMon is an auditing tool, it does not confine system calls, but provides useful input for system administrators to create policies on resource access which can then be used to limit privileges.
3. ***Monitoring sensitive registry locations to detect spyware.***
   Managing the Microsoft Windows registry is difficult due to its complexity. Spyware often takes advantage of this complexity to bury itself in the registry, making it difficult for the user to remove it completely. In [3], the authors have listed the most common entry points for spyware to enter a Windows NT system. The following are some of the configuration settings WinResMon can monitor:
   - *Autostarts*: monitor which programs load on startup.
   - *Internet explorer hooks*: track hooks which define the default search page, toolbars and browser helper objects (BHO), etc.
   - *Winlogon*: look for applications that hook into system resources.
   - *Services*: monitor services such as automatic startup services (e.g., task scheduler) or drivers which are installed as services.
   - *DLL injection*: monitor DLL injection attacks (any application that uses user32.dll can be hijacked by having a DLL injected into its process space).
   - *File associations*: monitor the registration of file extensions with applications. For example, .DOC is registered to Microsoft Word.

System Design

The WinResMon system infrastructure shown in Figure 1 consists of the following components: logger, archiver, query API, and user-log API. The logger generates resource-access traces which are later used by the analyzer. The archiver performs log compaction/summarization of old traces. Query and user-log API provide the interface to the trace database.

The Logger

The logger consists of a system call (syscall) interceptor and a trace generator module. The syscall interceptor is an in-kernel driver which monitors system calls made by each process and sends the monitored event information to the trace generator. The trace generator is a user-space service (daemon) which collects event information sent from the syscall interceptor and

generates resource access traces. Ideally, the logger is meant to run *all the time* so as to record the entire life cycle of how resources are used by software.[2]

A log trace file consists of a list of records, each representing an access operation on one of the following types of resource:

1. *File*: covering both directories and regular files
2. *Registry*
3. *Process*: mainly to record process creation
4. *Synchronization objects*: which records information on inter-process synchronization mechanisms provided by Microsoft Windows (mutex, semaphore, event and waitable timer)
5. *IPC*: including named pipes and mailslots.

In addition to these five basic resource types, WinResMon also captures *system event* information such as: process termination, system boot and shutdown, user login and logout. Moreover, it also provides a user-log API for users or applications to insert user/application-defined milestone events. One potential usage of custom events is to demarcate and distinguish the software installation and uninstallation portions of the log.

A record entry contains common information and specific information relevant to that record type. The common information consists of: *record-type*, *time*, *process-id*, and the *error-code* (in the case of failure). The specific information recorded by different record types are:

1. **File:**
   - absolute path of the file
   - file operation: *open*, *read*, *write*, *delete* or *move*
   - operation-specific information. For open: the access flags. For read and write: the number of bytes read or written. For move: the new path.
2. **Registry:**
   - absolute path of the registry key
   - operation: *open-key*, *query-value*, *set-value*, or *delete-key*
   - operation specific information. For *open-key*: the access flags. For *query-value* and *set-value*: the type, size, and value of the registry key.

   Due to the importance of the registry in software maintenance, WinResMon logs the actual data changes made to the registry; whereas for file I/O, it is not practical to record the data.
3. **Process:**
   - absolute path of the executable corresponding to the newly created process
   - command line arguments
   - process id of the newly created process.
4. **Synchronization objects:**
   - type of the object: *mutex*, *semaphore*, *event*, or *waitable-timer*
   - name of the object
   - operation: *create*, *open*, or *delete*.

   We are interested only in objects which have names in the system-wide namespace, because anonymous and process-wide named objects will not interfere with other programs, thus they are unrelated to software dependencies.
5. **IPC:**
   - type of the IPC: *named-pipe*, or *mailslot*
   - name of the IPC
   - operation: *create*, *open*, *delete*, *send*, or *receive*.

   WinResMon records the synchronization objects and IPC operations listed above since they involve global (system-wide) namespace which could be used by different programs to interact with each other. One can use WinResMon to uncover the causes of the following problems:

   a) If process *A* has created a semaphore *s*, and process *B*, which is unaware of the existence of *A*, is trying to create a semaphore with the same name *s*, *B*'s operation will fail.
   b) If process *A* fails to run correctly and semaphore *s* is not created, then process *B*, assuming the existence of *A*, will fail trying to use *s*.

[2]One might only run the logger at select times instead, but this means WinResMon could miss critical information.

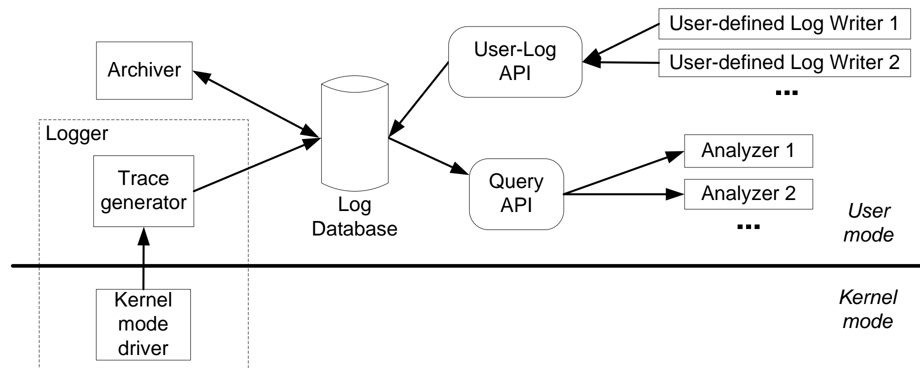

Figure 1: WinResMon overall system architecture.

6. **System event:**
   - system event type: *process-termination*, *boot*, *shutdown*, *user-login*, or *logout*
   - any event specific information: path of the executable of the process in the case of process termination, user name in the case of user login/logout, etc.
7. **User defined event:**
   - the path of the executable of the process generating the event
   - a binary string describing the event.

Not all operations need to be logged into the database as some resources are not significant to record, e.g., the temporary directory, C:\temp. A filter can therefore be used in the logger to prevent logging resource access of no interest to the system administrator.

The Log Database

The log database consists of a number of log files and one distinguished *active log* to which the logger records current resource access activity. To maintain reasonable log file sizes, WinResMon performs a "log switch" to create a new active log for recording subsequent entries. The old log files are then subject to the log compaction/summarization process by the archiver. Any of the following conditions can trigger a log switch:

- The size of the current log file reaches the specified Max_log_size.
- The number of entries in the current log file reaches Max_log_entries.
- The user manually initiates a log switch process.

Archiving Old Traces

As WinResMon logs system activities over a long period of time, the trace becomes very large. If old traces are discarded, some early yet potentially valuable information, such as identifying which program first created a file, is lost. Since it is necessary to eventually prune some information to avoid excessively large log files, WinResMon summarizes old trace files into a "*compacted trace database*". This maintains a balance between compactness and the ability to answer important questions about system resource access.

There are two main issues in performing trace compaction, determining when to initiate the compaction and how to perform compaction on old trace entries. WinResMon uses a module called the archiver which runs based on a specified Archive_interval_check time interval. Upon activation, the archiver compacts previously uncompacted entries which are older than the specified Old_log_age. WinResMon employs the following strategies in performing the compaction:

Log Entry Summarization/aggregation

WinResMon can summarize multiple entries with similar information to produce an aggregated entry in the compacted trace database. It applies the following policies on various resource types:

- **File:** multiple read or write operations on the same file are summarized by recording the time of the first and last operations and the total number of time the operations were done.
- **Registry:** multiple query-value operations on the same registry key are aggregated. Multiple set-value on a key are aggregated only if the values written are identical.
- **IPC:** send and receive operations of one IPC object are aggregated by recording the time of the first and last operations.

When matching a query on an aggregated entry which records a time interval, WinResMon considers that the entry satisfies the specified time constraint if *one* of the values matches the constraint.

Selective Priority-based Entry Removal

One strategy to reduce the old traces involves removing entries deemed to be of little value for answering future questions. WinResMon implements selective entry removal strategy based on a user-supplied configuration file. The configuration file assigns a priority to log entries. For example, a log entry for writing a registry key might be considered more important to keep than one for reading a registry key.

A fragment of an example configuration is shown in Appendix 2. This example uses priority values ranging from 1 (least important) to 5 (most important). For each resource type, configuration entries are matched in a sequential order, and mappings are listed from most to least specific. It is possible to omit some arguments, i.e., with wildcards. To simplify the priority assignment, WinResMon classifies file open operations in Microsoft Windows into one of three modes: RO (read-only), RW (read-write) and WO (write-only/append). The archiver translates the semantics of each operation from file flags in the raw log entries to the appropriate values for matching against the configuration.

The existing applications and anticipated usage, together with some general principles, can be used to derive the priority assignment for the configuration. One general principle is that "transient information," such as that those on synchronization objects and IPC, becomes less relevant after system shutdown and can be given low priority. During the removal process, all log entries with priority lower than the Lowest_priority_retained value will be purged.

Auto Deletion of Old Log Files

To maintain reasonable storage usage, WinResMon eventually needs to remove entries deemed too old. The log file whose newest entry timestamp exceeds Max_log_lifetime is deleted. If necessary, it is also possible to additionally provide an API function (in a secure manner) to perform deletion on selected trace entries based on a specified selection condition.

A Query API and Analyzer

We want to support trace analyzers which answer queries solving particular software maintenance problems.

WinResMon therefore provides a query API which can be used to obtain information from the trace database. One can view the trace as a database and the provided query API as the query language by which the analyzers extract relevant information from the database.

The main query API, analogous to an SQL Select command, is trace_select([*selection condition*], [*field projection*], [*time interval*], [*output order*]), where the caller specifies the matching condition(s) and fields to return. The *selection condition*, specified on string types such as program name and registry key path, takes the form of a regular expression. Logical operators can be used to combine matching conditions.

Continuing the database analogy, the caller uses projection to specify which fields to return. For example, suppose one only wants to know the list of programs accessing a certain file. In other cases, one wishes to know all access operations on the file (i.e., the order and access time matter). In the former, the analyzer only returns a set of program names. In the latter, it returns the sequence of all access operations.

We specify *time* in the format below:

1. "YYYY-MM-DD hh:mm:ss": an explicit timestamp
2. "-[count][m|h|d]": a relative time earlier than the current time
3. "OLDEST": the time of the oldest available entry in the log
4. "NOW": the current time.

The time interval is then specified as: "[*start time*] TO [*end time*]".[3] Some examples of time interval definitions are:

- "2006-01-25 11:47:51 TO 2006-08-21 13:41:16"
- "-1d TO NOW" (in the last 24 hours)
- "OLDEST TO -8h" (everything except the past 8 hours).

Output order controls the ordering of query results. It can be either: FORWARD, to list the oldest entry first or BACKWARD, to show the most recent entry first. The BACKWARD option is useful to get the *k* most recent operations.

After obtaining the *trace_handle* from trace_select(), one can call trace_next (*trace_handle*) to retrieve the records and trace_close(*trace_handle*) to finish retrieval. Some sample analyzer applications using this query API are described later.

The User-Log API

The user-log API lets applications add their own entries into the log database. As applications are not allowed to write directly to the log file, custom events are generated using an ioctl interface to the kernel driver. One use of user-entries includes marking events related to software installation, making it easier to determine what files and registry keys are created/modified during installation. This feature also provides a general purpose logging facility.

The API for user-entries is winresmon_userlog(*logdata*, *length*). It signals the trace generator to record the *logdata* to the log database in binary form. The ownership information of a user log entry (i.e., the program pathname and process) is always recorded.

Figure 2 shows a simple wrapper for an installer program which generates custom installation events. In this example, the logdata consists of the name of the software and path of the installer program. Invoking the wrapper as "C:\ins-wrapper.exe photoshop_cs2 H:\setup.exe", generates logdata containing "INBGH:\setup.exe|photoshop_cs2" and "INEDH:\setup.exe|photoshop_ cs2".

```
#define MAGIC_INSTALL_BEGIN "INBG"
#define MAGIC_INSTALL_END "INED"

int main (int argc, char **argv)
{
   char buff[256];

   if (argc != 3) {
     printf("example: %s software_name"
         " c:\...\installer.exe",
         argv[0]);
     exit(1);
   }

   buff[sizeof(buff)-1] = '\0';
   _snprintf(buff, sizeof(buff)-1,
       MAGIC_INSTALL_BEGIN "%s|%s",
       argv[2], argv[1]);
   winresmon_userlog(buff, strlen(buff));
   system(argv[2]); // fork and wait
   _snprintf(buff, sizeof(buff)-1,
       MAGIC_INSTALL_END "%s|%s",
       argv[2], argv[1]);
   winresmon_userlog(buff, strlen(buff));
   return 0;
}
```

Figure 2: A sample installer wrapper.

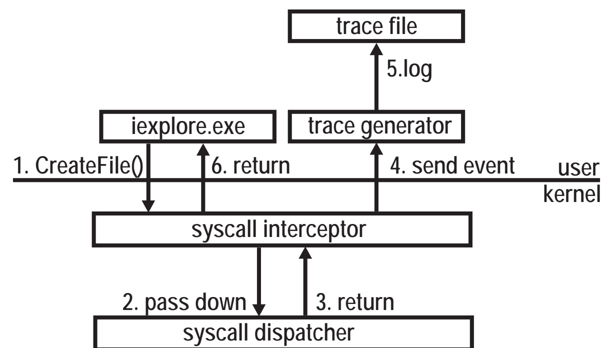

Figure 3: Overview of how the logger works.

Implementation

Figure 3 gives an overview of how the logger works. The information flow is as follows:

[3]Although an analyzer can include constraints on log's timestamp as conditions in the *selection condition*, an explicit time interval specification is less complex.

1. A sample program iexplore.exe calls CreateFile("C:\WINDOWS\system32\Macromed\Flash\Flash8.ocx", READ, ...).
2. This is intercepted by our system call interceptor which passes the parameters to the original syscall handling routine.
3. The syscall handling routine returns the file handle.
4. The handle, together with the system call parameters are then sent to the trace generator.
5. As the call is successful, the trace generator updates the file handle/path lookup table. Since "foo" is a relative path, the trace generator concatenates "foo" with the current directory and add it to the trace.
6. The handle is returned to iexplore.exe and operation continues as per normal.[4]

System Call Interception

The overview of the logger in Figure 3 shows that resource usage is captured by intercepting system calls. In practice, this is actually more complex. Rather than defining and using the actual system calls, the Microsoft Windows API is described at a level higher than the operating system using the Win32 API. Although most programs use Win32, they can use the native API [4] directly. The native system call interface API is unfortunately not well documented and supported. Furthermore, the view of the operating system at the native API level is not quite the same as at the Win32 level. This means that intercepting system calls at what regular programs might think of as the Microsoft Windows API is problematic, and there could be discrepancies and mismatches between API use at the Win32 level and native level. To ensure accuracy, WinResMon intercepts system calls at the native API level.

WinResMon implements the syscall interceptor by means of a kernel mode driver. The driver captures syscall requests made by a process by "hooking" the native system calls as described in [5]. Appendix 1 lists the system calls intercepted by WinResMon. We found these to be the most common system calls arising from file, registry, IPC, synchronization, and process operations.[5] To gather information about processes, WinResMon uses a simpler method. Microsoft Windows NT exports a set of process callback functions in the kernel space [6]. WinResMon makes use of PsSetCreateProcessNotifyRoutine() and PsSetLoadImageNotifyRoutine() which notifies the call back function during process creation or termination.

Event Handling

When sending the data from the kernel to the user space, it is inefficient to send every event as it arrives. WinResMon's implementation therefore makes use of double buffering. When the size of a buffer reaches a threshold, WinResMon sends the contents of the buffer to the user space and switches to another buffer to continue logging in the kernel space. This strategy tries to ensure that most, if not all, events are captured and reduces the system overhead due to context switching. Again, due to the undocumented nature of the kernel, we can not claim to capture all events.

In the prototype implementation, the kernel and user space communicate through an ioctl mechanism. Ioctls are used to define a protocol to synchronize data transfer between the driver and user level trace generator. WinResMon can also be extended into a remote-monitoring tool.

```
struct trace_struct *handle;
struct trace_entry *entry;

trace_handle = trace_select(
   "type==\"registry\" && "
   "prog_path=~\"/iexplorer.exe$/\"",
   "time, registry.path",
   "-1d TO NOW",
   FORWARD);
if (trace_handle == NULL)
 exit(1);
while ((entry = trace_next(trace_handle))
   != NULL) {
   printf("time=%s, registry=%s",
   entry->fields[0],
   entry->fields[1]);
}
trace_close(trace_handle);
```

Figure 4: A sample analyzer.

Writing Custom Analyzers

This section demonstrates how to write custom analyzers on top of the WinResMon framework by means of examples.

- An analyzer to show all the programs which read C:\foo.txt after 2005/1/1 could use the following query:

```
trace_select (
   "file.path == \"C:\\foo.txt\"",
   "prog_path",
   "2005-1-1 00:00:00 TO NOW",
   FORWARD);
```

- A more complicated example asks, "What's the most recent execution of msnmsgr.exe before this boot?" First determine the last *shutdown_time* with the query:

```
trace_select (
   "sysevent.type == \"shutdown\"",
   "time",
   "OLDEST TO NOW",
   BACKWARD);
```

The next query gets the whole process creation event for msnmsgr.exe:

[4] Step 6 and 4 are independent. Thus, depending on the scheduler, step 6 is not necessarily executed after step 4.

[5] Since Microsoft Windows is closed source and the native API is only partially documented, it is difficult to make any guarantees about completeness.

```
trace_select ("proc.childname =~ "
   "\"/^.*\\msnmagr.exe$/\"",
   NULL,
   "OLDEST TO last_shutdown_time",
   BACKWARD);
```

- Figure 4 shows a code fragment from a simple analyzer which searches for all the registry keys opened by Internet Explorer.

  WinResMon issues the query with selection type == "registry" && prog_path =~ "/iexplorer.exe$/" and projection on time and registry.path. After correctly obtaining a trace_handle, it iterates over all selected records by using trace_next(). It prints all the fields, time and registry.path, for each selected trace entry. After iterating over all relevant records, it closes the handle.

Using WinResMon

The extended example below shows the use of WinResMon to solve software maintenance problems.

Yahoo Toolbar [7] adds tabbed browsing to Internet Explorer (IE) and adds various icons and links from within IE to different Yahoo services. The following walk-through illustrates monitoring the Yahoo toolbar throughout its entire life cycle. There are three stages: Installation, Program usage, and Uninstalling.

Installation

The provided installer refuses to run under a standard user account as it needs administrator privileges. Upon successful installation under an administrator account, it creates the following DLLS and keys:

DLLs

```
C:\ProgramFiles\Yahoo!\Companion\
Installs\cpn\yt.dll
C:\ProgramFiles\Yahoo!\Companion\
Installs\cpn\YTabBar.dll
...
```

Registry keys

```
HKEY\_LOCAL\_MACHINE\SOFTWARE\Yahoo
```

From our list of sensitive registry locations, we note the following.

Before Installation

Search Page:

```
http://www.microsoft.com/isapi/
redir.dll?prd=ie&ar=iesearch
```

Search Bar: -

After Installation

Search Page:

```
http://us.rd.yahoo.com/customize/
ycomp/defaults/sp/*http://www.yahoo.com
```

Search Bar:

```
http://us.rd.yahoo.com/customize/
ycomp/defaults/sb/*http://www.yahoo.com/
search/ie.html
```

Yahoo! Toolbar Helper:

```
02478D38-C3F9-4EFB-9B51-7695ECA05670 -
C:\ProgramFiles\Yahoo!\Companion\
Installs\cpn0\yt.dll
```

Among the changes made, Yahoo replaced MSN search as the default search engine.

Program Usage

Since the database is persistent, WinResMon logs all the events associated with Yahoo and preserves the information even if the system reboots. This helps analyze the behavior of Yahoo toolbar over a period of time.

Uninstalling

When uninstalling Yahoo toolbar, WinResMon observes that all the files have been removed. However, the registry settings it made are left unchanged. As a result, Yahoo remains the default search engine for the system.

WinResMon Overhead

To measure the performance overhead resulting from constant monitoring of systems with WinResMon, we first look at some worst case scenarios using micro-benchmarks consisting of only repeated system calls.

Our micro-benchmarks comprise of: seven benchmarks on file access, five on registry access, and two on process creation. All of these micro-benchmarks run on a Pentium 4 2.4GHz machine with 512MB running Microsoft Windows XP with SP2. The benchmarking procedure consists of first running the benchmarks on a clean Microsoft Windows XP (with SP2) to get the baseline performance. The next battery runs with WinResMon loaded. The last battery is run with FileMon [8] loaded for file access benchmarks and RegMon [9] loaded for registry access benchmarks. Each micro-benchmark repeats an operation n times. We performed each benchmark four times to get the average execution time. Tables 1, 2 and 3 show the average and standard deviation of the execution time in seconds.

The file access benchmarks consist of:

(F_1) Open an existing file. The same filename is used every time.

(F_2) Create a new file and delete it. A different filename is used every time.

(F_3) Read 1 byte from a file. We ensure that the file is large enough so that EOF is never met for multiple reads.

(F_4) Read 4,096 bytes from a file. The file size is a multiple of 4,096. When we reach EOF, we rewind to the beginning of the file.

(F_5) Write 1 byte to a file. We start with an empty file.

(F_6) Write 4,096 bytes to a file. When we reach EOF, we rewind to the beginning of the file.

(F_7) Create a new directory and delete it. A different filename is used every time.

The benchmarks for registry access are:

(R_1) Open an existing registry key. The same key is used every time.

(R_2) Create a new registry key and delete it. A different name is used every time.

(R_3) Create a new volatile registry key and delete it. RegCreateKeyEx is used with the REG_OPTION_VOLATILE option.

(R_4) Query the value of a registry key. The type REG_DWORD is used.

(R_5) Set the value of a registry key. The type REG_DWORD is used.

The benchmarks for process creation are:

(P_1) Create a dummy console process and wait for its termination.

(P_2) Create a dummy GUI process and wait for its termination.

Table 1 shows the execution time (in seconds) for the file access benchmarks. In order to avoid any extraneous overhead from the FileMon GUI, the window is always minimized during the experiments. It appears that FileMon does not capture all the operations during the performed micro-benchmark, though. For example, during the "Read 1 byte" test, 10 M events occurred, but FileMon only captured about 15K. During the "Create a new file" test, 600 K events occurred, but only about 18 K were actually captured. We observe that WinResMon, by contrast, captured all operations in all the tests.

Table 2 shows the execution time (in seconds) of the registry related benchmarks. This benchmark is conducted similarly to the file benchmark. It appears that RegMon also drops events. For example, during one of the "Query value" test, 1 M events occurred, but only 993,938 were actually captured by RegMon.

Note that FileMon, RegMon and WinResMon address different goals. FileMon and RegMon are meant for short term monitoring while WinResMon is designed for long term use and is therefore always running in the background. These two benchmark comparisons merely give us a baseline on how WinResMon compares with other, similar monitoring software.

Table 3 shows the results of the process creation benchmark. The console process in the benchmark creates a dummy child process using the CreateProcess() function and waits for its termination using the WaitForSingleObject() function. The difference between a console program and a GUI program is that the GUI program uses the WinMain() entry function and is linked using the /SUBSYSTEM:WINDOWS option, while the console program uses main() and /SUBSYSTEM:CONSOLE. The measured overhead is quite small because process creation is a slower operation than file or registry access.

We would expect normal programs to have much smaller overhead than that of the micro-benchmarks because the micro-benchmarks are very system call intensive. Normal programs, such as our macro-benchmarks, typically make significantly fewer system calls, spending more time in the application rather than the kernel. Table 4 gives some macro-benchmark results which show the impact of WinResMon on the following normal programs: WinRAR, gcc, LaTeX, and Lame. The benchmarks perform the following:

- WinRAR: compress a 150 MB file
- gcc: compile a 500 K-line C program
- latex: compile a 2,000-page LaTeX file into PDF using the pdflatex program
- Lame: encode a 100 M wave file into a mp3 file.

The macro-benchmark results demonstrate that running WinResMon all the time is quite reasonable under typical usage.

Related Work

From a high level perspective, WinResMon differs from previous systems/tools in that it is:

File Operation	n	Clean	WinResMon	FileMon
(F_1) Open an existing file	1 M	20.457 ± 0.240	46.266 ± 3.271 (126.2%)	44.168 ± 0.279 (116.0%)
(F_2) Create a new file	100 K	53.004 ± 2.532	67.539 ± 0.469 (27.4%)	73.117 ± 0.265 (37.9%)
(F_3) Read 1 byte	10 M	14.277 ± 1.084	278.175 ± 14.282 (1848.4%)	107.414 ± 4.765 (652.4%)
(F_4) Read 4096 bytes	10 M	41.207 ± 0.133	328.203 ± 34.869 (696.5%)	138.816 ± 0.793 (236.9%)
(F_5) Write 1 byte	10 M	49.824 ± 1.160	388.050 ± 0.837 (678.8%)	172.422 ± 1.114 (246.1%)
(F_6) Write 4096 bytes	10 M	116.355 ± 0.716	448.933 ± 2.192 (285.8%)	212.828 ± 2.950 (82.9%)
(F_7) Create a new directory	100 K	46.546 ± 0.344	57.750 ± 9.565 (24.1%)	56.395 ± 0.282 (21.2%)

Table 1: Performance comparison on file access (in seconds).

Registry Operation	n	Clean	WinResMon	RegMon
(R_1) Open an existing key	1M	10.378 ± 0.039	35.324 ± 0.080 (240.4%)	361.438 ± 40.504 (3382.7%)
(R_2) Create a new key	100K	8.980 ± 0.037	13.769 ± 0.041 (53.3%)	134.879 ± 10.778 (1402.0%)
(R_3) Create a new temp key	100K	7.832 ± 0.045	12.750 ± 0.082 (62.8%)	142.961 ± 12.811 (1725.3%)
(R_4) Query value	1M	1.461 ± 0.009	27.203 ± 0.061 (1761.9%)	166.301 ± 4.406 (11382.7%)
(R_5) Set value	1M	22.890 ± 0.153	46.379 ± 0.090 (102.6%)	182.473 ± 7.272 (697.1%)

Table 2: Performance comparison on registry access (in seconds).

- *integrated*: since it monitors accesses on files and registry under one infrastructure
- *extensible*: system administrators can write their own custom modules to utilize the generated log
- *geared for log management*: system administrators can view resource access activities generated over time, and inspect their relationships with respect to software configuration and dependencies.

We briefly mention some other tools/systems below, and highlight the important differences with WinResMon.

FileMon [8] and RegMon [9] are file and registry monitoring tools, respectively. They monitor operations taking place on the registry or specified file system in real time. Although WinResMon shares the basic monitoring functionalities with these two tools, WinResMon's infrastructure is integrated, and its log database is designed to assist system administrators in inspecting software configuration and dependencies.

Strace [10] is a Linux/UNIX tool used to intercept and log system calls invoked by a process. There is also a Microsoft Windows NT port of strace [11] with similar functionality. WinResMon differs from strace in that is focused more on resource usage (files, registry, etc.) rather than system calls. In Microsoft Windows, a system call viewpoint can be confusing since there are multiple levels of APIs which translate into the poorly documented native API.

Systrace [2] is a UNIX tool for sandboxing untrusted code. Unlike Systrace, which examines system-call sequences issued by the monitored processes and applies a specific security policy, WinResMon is meant as a monitoring tool to inspect resource usage and interactions among programs in a system.

DTrace [12], SystemTap [13] and LBox [14] are auditing and instrumentation systems on various UNIX operating systems. They are all event based auditing systems, performing a specific action only on a specific event. DTrace and SystemTap allow administrators to dynamically execute supplied code in the kernel when certain event occurs. LBox allows the kernel to notify a user space program when certain events happen. Both LBox and WinResMon are designed for monitoring resource usage, while DTrace and SystemTap are designed for general system call instrumentation.

Conclusion

This paper presented the motivation, design, implementation and usage of WinResMon. Its main use is to inspect resource access and software dependency issues in Microsoft Windows environments. As WinResMon is extensible, system administrators can also build tools using WinResMon for custom queries and system analysis. Benchmarking shows that WinResMon is reliable and is comparable to other popular tools.

Future work is to increase the usability and robustness. We would also like to ensure that logging is as comprehensive as possible taking into account the undocumented and unsupported nature of the APIs in the Microsoft Windows NT kernel. We would also like to further increase the efficiency of the logging mechanism.

Acknowledgments

We acknowledge the support of the "Defence Science and Technology Agency" and "Temasek Laboratories." We also would like to thank Amy Rich for many useful comments and suggestions in improving the paper.

Author Biographies

Rajiv Ramnath is currently a final year student in the School of Computing at National University of Singapore. His interests include operating systems and computer security. He can be reached at rajivram@comp.nus.edu.sg .

Sufatrio holds a B.Sc. from University of Indonesia and an M.Sc. from National University of Singapore. He is currently a Ph.D. student in the School of Computing and an associate scientist in Temasek Laboratories at National University of Singapore. His interests include intrusion detection systems and infrastructure for secure program execution. He can be reached electronically at tslsufat@nus.edu.sg .

Roland Yap obtained his Ph.D. from Monash University. He is currently an associate professor in the School of Computing at National University of

Process Operation	*n*	Clean	WinResMon
(P_1) Create a console process	10K	35.488 ± 0.071	37.855 ± 0.150 (6.7%)
(P_2) Create a GUI process	10K	34.641 ± 0.044	36.938 ± 0.097 (6.7%)

Table 3: Performance of process creation (in seconds).

Test Case	Clean	WinResMon
WinRAR	224.443 ± 0.542	226.524 ± 3.502 (0.9%)
gcc	26.265 ± 1.219	26.973 ± 0.968 (2.70%)
LaTeX	27.211 ± 0.473	27.498 ± 0.981 (1.1%)
Lame	45.631 ± 0.538	45.662 ± 0.534 (0.6%)

Table 4: Performance of macro-benchmarks (in seconds).

Singapore. His interests include systems security, operating systems, programming languages, and distributed systems. He can be reached electronically at ryap@comp.nus.edu.sg .

Wu Yongzheng holds a B.Comp. from National University of Singapore. He is currently a Ph.D. student in the School of Computing at National University of Singapore. His interests include systems security and operating system. He can be reached at wuyongzh@comp.nus.edu.sg .

Bibliography

[1] http://www.rpm.org/ .

[2] Provos, N., "Improving Host Security with System Call Policies," *USENIX Security Symposium*, pp. 257-272, 2003.

[3] Wang, Y. M., R. Roussev, C. Verbowski, A. Johnson, M. W. Wu, Y. Huang and S. Y. Kuo, "Gatekeeper: Monitoring Auto-Start Extensibility Points (ASEPs) for Spyware Management," *Large Installation System Administration Conference*, pp. 33-46, 2004.

[4] Nebbett, G., *Windows NT/2000 Native API Reference*, Macmillan Technical Publishing, Indianapolis, 2000.

[5] http://www.ddj.com/184410109 .

[6] Microsoft MSDN, "Process Callbacks," http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Kernel_r/hh/Kernel_r/k108_a0f7bff2-270e-41fb-87d4-d8d533aa0bef.xml.asp .

[7] http://toolbar.yahoo.com/.

[8] http://www.sysinternals.com/Utilities/Filemon.html .

[9] http://www.sysinternals.com/Utilities/Regmon.html .

[10] http://www.liacs.nl/ wichert/strace/ .

[11] http://www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm .

[12] Cantrill, B. M., M. W. Shapiro and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," *USENIX Annual Technical Conference*, pp. 15-28, 2004.

[13] Prasad, V., W. Cohen, F. Eigler, M. Hunt, J. Keniston, B. Chen, "Locating System Problems Using Dynamic Instrumentation," *Linux Symposium*, Vol. 2, pp. 57-72, 2005.

[14] Wu, Y. Z. and R. H. C. Yap, "A User-level Framework for Auditing and Monitoring," *Annual Computer Security Applications Conference*, pp. 95-105, 2005.

Appendix 1: List of Intercepted System Calls

Table 5 lists the Microsoft Windows NT native-level system calls intercepted in our current implementation.

File	
ZwCreateFile	opens or creates a new file
ZwOpenFile	opens an existing file
ZwDeleteFile	deletes a file
ZwReadFile	reads from an open file
ZwWriteFile	writes to an open file
ZwQuerySystemInformation	queries for information internal to the system
Registry	
ZwCreateKey	opens an existing key or creates it if it does not exist
ZwDeleteKey	deletes a key
ZwOpenKey	opens an existing key
ZwQueryKey	provides information about the size and number of subkeys (if any)
ZwQueryValueKey	provides the value of a registry key entry
ZwSetValuekey	creates or replaces a registry key's value entry
ZwDeleteValueKey	deletes a registry key's value entry
Process	
ZwTerminateProcess	terminates a process and all its threads.
Synchronization Object (Mutex)	
ZwCreateMutant	creates a mutex or opens an existing mutex
ZwOpenMutant	opens an existing mutex
Synchronization Object (Semaphore)	
ZwCreateSemaphore	creates a semaphore or opens an existing semaphore
ZwOpenSemaphore	opens an existing semaphore
Synchronization Object (Event)	
ZwCreateEvent	creates a new event or opens an existing event
ZwOpenEvent	opens an existing event
Synchronization Object (Waitable Timer)	
ZwCreateTimer	creates a new timer object or opens an existing timer object
ZwOpenTimer	opens an existing timer object
IPC (Named pipe)	
ZwCreateNamedPipeFile	creates a named pipe
IPC (Mailslot)	
ZwCreateMailslotFile	creates a mailslot

Table 5: Intercepted system calls.

Appendix 2: Example of Log Priorities for Trace Compaction

The following example shows a fragment from a log configuration which assigns priorities on FILE resource-type entries. We map log entries into priority values ranging from 1 (least important to retain) to 5 (most important to retain).

```
# File Section
# Format:
# Type   Action  Object                                          Mode    Priority
FILE    Read    *                                               *         1
FILE    Write   *                                               *         1
File    Open    C:\Windows\Temp\*                               *         1
File    Open    C:\Windows\System32\*                           RW|WO     5
File    Open    C:\Windows\System32\*                           RO        4
File    Open    C:\Windows\*                                    RW|WO     4
File    Open    C:\Windows\*                                    RO        3
File    Open    C:\ProgramFiles\*                               RW|WO     3
File    Open    C:\ProgramFiles\*                               RO        2
File    Open    C:\Documentsand Settings\LocalSettings\
                {Temp\*|Temporary Internet Files\*}             *         1
File    Open    *                                               RW|WO     2
File    Open    *                                               RO        1
File    Delete  C:\Windows\Temp\*                               *         1
File    Delete  C:\Windows\System32\*                           *         5
File    Delete  C:\Windows\*                                    *         4
File    Delete  C:\ProgramFiles\*                               *         3
File    Delete  C:\Documentsand Settings\LocalSettings\
                {Temp\*|Temporary Internet Files\*}             *         1
File    Delete  *                                               *         2
....
```