Intrusion detection has emerged as an important approach to network, host and application security. Network security includes analysing network packet payload and other inert network packet profiles for intrusive trends; whereas, host security may employ system logs for intrusion detection. In this paper, we contribute to the research community by tackling application security and attempt to detect intrusion via differentiating normal and abnormal application behaviour. A method for anomaly intrusion detection for applications is proposed based on deterministic system call traces derived from a monitored target application’s dynamic link libraries (DLLs). We isolate associated DLLs of a monitored target application; log system call traces of the application in real time and use heuristic method to detect intrusion before the application is fully compromised. Our investigative research experiment methodology and set-up are reported, alongside our experimental procedure and results that show our research effort is effective and efficient, and can be used in practice to monitor a target application in real time.
of the system calls made by the application at runtime, it is possible to detect and prevent malicious system calls that attackers issue before an application is fully compromised.

The mechanics of system call-based anomaly detection is well understood, owing to past research efforts of Forrest et al. (1998). Forrest et al. proved that anomalous program behaviours produce system call sequences that have not been observed under normal operation. In order to make the learning algorithm computationally traceable, Forrest broke a system calls sequence into sub-strings of fixed length \( N \). These strings, called \( N \) grams, are learnt by storing them in a table for analysis and comparisons. In practice, \( N \) must be small since the number of \( N \) gram grows exponentially with \( N \).

A drawback of using small values of \( N \) is that the learning algorithm becomes ineffective in capturing correlations among system calls that occur over longer spans. The second deficiency with \( N \) grams is that it can recognise only the set of \( N \) gram encountered during training; similar behaviours that produce small variations in the \( N \) gram will be considered anomalous. Ghosh and Schwartzbard (1999) report that this lack of generalization in the \( N \) gram learning algorithm leads to a relatively high degree of false alarms. An alternative approach for learning strings is to use finite state automata (FSA) (Sekar et al., 2001). Unlike the \( N \) gram algorithm that limits both the length and number of sequences of arbitrary length using finite storage, FSA can remember short- and long-range correlation. Moreover, FSA can capture structures such as loops and branches in applications by traversing these structures in different ways; it is possible to produce new behaviours that are similar to behaviours encountered in training data.

In this paper we single out Microsoft Windows Internet Explorer (IE) as our monitored target application because it is often used for Internet browsing and thus, often exposed to attacks. Our methodology is not exclusive to IE, but can easily be adapted to other Microsoft Windows applications. The main contribution of this paper, is investigating the use of system calls of IE to detect abnormal application behaviour. To achieve this, we derive IE derived directly related dynamic link libraries (DLLs) via its Portable Executable file format. On knowing our target application Microsoft supplied related DLLs, we created bogus DLLs with the same name to enable us intercept associated system calls generated by both our target application and Microsoft supplied DLLs.

This paper is structured as follows. Section 2 presents the background and motivation of our research work, a detailed review of related work can be seen in Section 3, with our research approach described in Section 4. Section 5 presents our experimental procedure, methodology and results, and we summarise our research work in Section 6.

### 2. Background and motivation

Owing to the first formalised publication of the seminal report by Anderson in 1980, anomaly detection technologies have been applied to the problem of detecting intrusions. Alongside this, the first comprehensive model of IDS was proposed by Denning in 1987. This highly cited model, includes anomaly records as one of the six basic components of a generic IDS.

Inferences from Denning published model contributed to the wide variety of methods to detect anomalous activity, applied by intrusion detection researchers.

The earliest proposed methods for intrusion detection focused on the application of statistical methods to identify anomalous activity (Helman and Liepins, 1993). Many early systems (Lankewick and Benard, 1991) employ this method. In addition, a number of on going projects (Jou et al., 2000) continue to employ statistical methods for anomaly detection, typically in combination with other methods. More recently, anomaly detection methods employ a wide variety of classification schemes to identify anomalous activities. These schemes include, among others, rule induction (Habra et al., 1992), neural network (Mukkamala et al., 2002), fuzzy set theory (Lin, 1994), classical machine learning algorithm (Lee and Stolfo, 2000), artificial immune system (Forrest et al., 1997), signal processing methods (Lee and Dong, 2001), and temporal sequence learning (Kosoresow and Hofmeyer, 1997).

Clearly, the inclusion of too much data will adversely impact the performance of the system, while the inclusion of too little data will reduce the overall effectiveness of the system. Data gathering from learning application behaviour and building application profiles has become an alternative method in intrusion detection (Warrender et al., 1999). Monitoring the application execution and capturing the system calls associated with the application can generate an application profile. Compared to user behaviour profile, application profiles are more stable over time because the range of program behaviour is more limited. Furthermore, it would be more difficult for attackers to perform intrusive activities without revealing their tracks in the execution logs. Therefore application profiles provide concise and stable tracks for intrusion detection.

We aim to limit the impact an adversary can have on a system by restricting the operations an application is allowed to execute. The observation that security relevant changes are performed via system calls makes the enforcement of restriction at the system call level a natural choice. An application is confined by a set of restrictions that are expressed by security profiles. The operation of an application is based on the system calls it makes. Defining a correct system call route is difficult and not possible without knowing all possible code paths that an uncompromising application may take. We may use the system call routes as specification that describes the expected or unexpected behaviour of an application. When monitoring the operation an application attempts to execute, any deviation from the system call route may indicate a security compromise (Ko et al., 1994). To further facilitate forensic analysis of an intrusion, we also wish to generate an audit log of previous operations related to the application. There are several methods that can be used to monitor and intercept system calls for the above purposes as described below.

Often most of the client and server communication is between user and application processes on the same computer via win32 APIs (Nebbet, 2000). Microsoft Windows APIs are exposed by four dynamic libraries: user32.dll, gdi32.dll, kernel32.dll and advapi32.dll. The APIs in user32.dll and gdi32.dll invoke the APIs implemented in the kernel mode by a win32.sys module. The APIs exposed by kernel32.dll (system APIs) use a particular library named Ntdll.dll that invokes native APIs (Table 1) in the kernel, as shown in Table 1 and...
system calls derived by these DLLs is required. A number of methods can be used to monitor the win32 system calls made by application (Forrest et al., 1996). We chose to use a similar method to the direct patching method implemented by the Detour package (Hunt and Brubacher, 1999) that instruments the DLLs contained in win32 API at load time. By directly patching the entry point of each win32 API, all win32 system calls can be monitored. Patching DLLs at load time allows application executables to be monitored selectively. Fig. 1 depicts how a call to a win32 API occurs from an application process when the API is patched with Detours. The EXE process makes a system call into the API function, the first instruction (1) of which is an unconditional jump to Win32.dll (A). While the real Win32.dll (A) as been replaced with a Doutour created Win32.dll (A) and renamed Win32.dll (A2). On receiving a call, Win32.dll (A) analysis and stores the call before passing control (2) to the real Win32.dll (A) or Win32.dll (A2). Further still, Win32.dll (A2) calls (3) Win32.dll (B) that as been replaced with a Doutour created Win32.dll (B) and renamed Win32.dll (B2). This process repeats with Doutour created Win32.dll storing and analysing system calls before passing control to their respective authentic Win32.dlls.

Other monitoring approaches described in www.codeproject.com include: proxy DLL, code overwriting, spying by a debugger and spying by altering of the Import Addresses table just to name a few.

### 3. Related work

A GridBox architecture aims to provide additional security for Grid applications, using Access Control List (ACL) and sandbox functionality for specific tasks (Dodonov et al., 2004). A series of different security approaches were presented including, process separation: limiting file system access (chroot, sandbox and jail approaches), where each process can only access its files, preventing it from affecting other applications; security enforcement: attempts to define all possible process functions and allowing limited access to the system via ACL; and system restriction: VMlinux (Dike, 2000) and VServer (Linux vserver Project, 2005) were used to introduce a new security model, which consists of the creation of different virtual servers in the same operating system. This way, each process runs independently from each other. This research is similar to ours, but where we used system calls to enforce security they use ACL. Their implementation of ACL is probably easier than system calls, since it can be carried out in user mode and in addition more user friendly. Nevertheless, security implementation carried out in user mode layer is more susceptible compromise than our system call implemented in both user and kernel mode layers.

Bruschi et al. (1998) presents a tool for the automatic detection of buffer overrun vulnerabilities in object code. It can be applied to operating system components as well as ordinary

---

**Table 1 – Native API categories**

<table>
<thead>
<tr>
<th>Index</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Special files</td>
<td>Used to create files</td>
</tr>
<tr>
<td>2</td>
<td>Drives</td>
<td>Used to load and unload drivers</td>
</tr>
<tr>
<td>3</td>
<td>Processors and bus</td>
<td>Controls processor register</td>
</tr>
<tr>
<td>4</td>
<td>Debugging and profiling</td>
<td>Used for kernel mode execution</td>
</tr>
<tr>
<td>5</td>
<td>Channels</td>
<td>Used for communication mechanism</td>
</tr>
<tr>
<td>6</td>
<td>Power</td>
<td>Used for power management</td>
</tr>
<tr>
<td>7</td>
<td>Plug-and-play</td>
<td>Similar to power API</td>
</tr>
<tr>
<td>8</td>
<td>Objects</td>
<td>Used for objects</td>
</tr>
<tr>
<td>9</td>
<td>Registry</td>
<td>Maps directly to win32 registry functions</td>
</tr>
<tr>
<td>10</td>
<td>LPC</td>
<td>Used inter-process communication</td>
</tr>
<tr>
<td>11</td>
<td>Security</td>
<td>Maps directly with w32 security API</td>
</tr>
<tr>
<td>12</td>
<td>Processes and threads</td>
<td>Used to control processes and threads</td>
</tr>
<tr>
<td>13</td>
<td>Atoms</td>
<td>Used for storage and referencing characters</td>
</tr>
<tr>
<td>14</td>
<td>Error handling</td>
<td>Used for driver and device errors handling</td>
</tr>
<tr>
<td>15</td>
<td>Execution environment</td>
<td>Used for general execution environment</td>
</tr>
<tr>
<td>16</td>
<td>Timers and system time</td>
<td>Accessible via win32 APIs</td>
</tr>
<tr>
<td>17</td>
<td>Synchronization</td>
<td>Used by PLC facility</td>
</tr>
<tr>
<td>18</td>
<td>Memory</td>
<td>Accessible by win32 memory API</td>
</tr>
<tr>
<td>19</td>
<td>File and general I/O</td>
<td>Used by device drivers</td>
</tr>
</tbody>
</table>

---

*Fig. 1 – Detouring win32 API Calls.*
programs. The tool searches the file system for critical programs and test them individually against buffer overruns. For the vulnerable ones, an exploit is built and executed against the program, then a notice is sent to a security personnel. Both their research work and ours are capable of detecting buffer overruns, ours mainly due to abnormal system calls generated and theirs via the program object code. An advantage of our research approach is in real-time detection and the detection of other malicious exploits than buffer overflow like intrusively crafted URLs.

A paper by Jain and Sekar (2000) presents a new approach for implementation of a system call extension infrastructure. This approach is characterised by a user-level implementation, where the system calls performed by one process are intercepted (and possibly modified) by another. This approach is contrary to majority of approaches in this area that employ an in-kennel implementation of system calls interpositions. This work improves upon (Goldberg et al., 1996) by providing a more extensive set of capabilities for extension code, developing an architecture and infrastructure that is easily ported to different versions of the Linux operating system, and presenting a comprehensive evaluation of the performance overhead associated with extensions. The author only briefly described the scope of DLLs used and how they are interfaced; hence direct comparison with our research approach cannot be made.

Ko et al. (2000), carried out research on detecting and counteracting system intrusion using software wrappers. They implemented several ID wrappers that employ three different major intrusion detection techniques and tested these software wrappers using varied attacks, concluding that intrusion detection algorithm can be easily encoded as wrappers that perform efficiently inside the kernel. This is roughly the closet research work to ours, but dwells more on the algorithm rather than the methodology of the research approach.

The Callgraph model proposed by Wagner and Dean (2001) characterizes the expected system call traces using static analysis of the program code. A non-deterministic finite automation (Ndfa) is created from a global control flow graph. The automation is non-deterministic because in general, which choice of branch an application process will take cannot be statically predicted. As a result, the N DFA can be used to monitor the application execution online. The operation of the N DFA is simulated on the observed system call trace non-deterministically. If all the non-deterministic paths are blocked at some point, there is an anomaly. It was stated that there was no false positives because all possible execution paths were considered in the automation. In case any possible paths were missed, a pushdown automation model, called the abstract stack model was proposed to curb this problem. One negative aspect of the above model is that the efficiency derived from monitoring the model is too low for many applications. Also, the overhead induced by monitoring is longer than 40 min per transaction for half of the application in their experiments (Wagner and Dean, 2001). This is because of the complexity of the pushdown automata and the non-determinisms of the simulation. Also, too much non-determinism may impair the ability to detect intrusions. This problem is not well addressed in this paper. There may be scalability problems, because of the human efforts in refining models for some libraries.

There are many methods that only model system call traces. The N gram method models program behaviours using fixed-length system call sequences (Hunt and Brubacher, 1999); data mining based approaches generate rules from system call sequences (Lee et al., 1997). Hidden Markov and Neural Networks have been used (Ghosh and Schwartzbard, 1999; Warrender et al., 1999); algorithms originally developed for computational biology were also introduced into this arena. The method proposed by Sekar et al. (2001) does not have the problems related to non-determinisms. Instead of statically analysing the source code or binary, the method called finite state automata (FSA) generated a deterministic FSA by monitoring the normal application execution at runtime. Each distinct program counter of the system is made a state and system calls are used as the label for transition. The FSA can then be used to monitor the application execution online. If the stack hosting the application crashes or the state or transition does not exist there may be an anomaly. There are false positives also because some legal transitions or states may never occur during training. FSA method suffers from varied possible path problems mentioned earlier in this section. This problem was not adequately addressed in this paper. Also, some implementation issues were not properly addressed. The way DLLs were handled is so simple that some intrusions on DLLs may be missed.

Ashcraft and Engler (2002) proposed to use programmer written compiler extensions to catch security holes. Their basic idea is to find violations of some simple rules using system specific static analysis. One example of these rules is “integers from untrusted sources must be sanitised before use”. While we agree that their method can be very useful in finding programming errors, we do not think it is a panacea that can solve all the problems. A lot of security requirements are subtle and cannot be described in simple rules. As a result, sometimes we can decide whether an action should be permitted only by checking whether this action occurs before normal situations. A problem with Ashcraft’s approach is that the rules have to be system specific, hence laborious.

Wespi et al. (2000) presented a novel technique to build a table of variable-length system call pattern based on the Teirebias algorithm. Teirebias algorithm was initially developed for discovering rigid patterns in unaligned biological sequences (Rigoutsos and Floratos, 1998). Wespi et al. announced that their method worked better than using N gram approach, although Teirebias algorithm is quite time and space consuming when applied on long traces containing many maximal pattern.

Buffer overflow based attacks have been around at least since 1980s, and many solutions have been proposed in the past to solve the problem. Some Unix distributors (BSDI – www.osdata.com/oses/bsd.html and OpenBSD – www.openbsd.org) have modified a linker to produce warning messages when an application uses dangerous functions. This approach implies many “false positives”, since the use of dangerous functions are not all incorrect, whereas overflows occur not only in standard libraries. There are both commercial (Hastings and Joyce, 1992) and public domain (Jones and Kelly, 2005) solutions that add array bounds checking capabilities to “C” programs. These packages can be considered as good debugging tools but, in a production environment, their use is not feasible since their performance penalty is barely acceptable.
Cowan et al. (1998) proposed a method called StackGuard to detect and prevent buffer overflow attacks. StackGuard is a compiler technique for providing code pointer integrity checking to the return address. The basic idea is to place a “canary word” next to the return address on the stack, and check if this word was modified before the function returns. A negative side to this technique is that it only detects buffer overflow attacks. While our research approach will detect other attacks like DoS.

The technique of Giffin et al. (2002) was developed for securing mobile code, such as remote procedure calls. It operates an executable code and creates models that are similar to the NDFA model of Wagner and Dean (2001). The authors suggest several application transformation techniques to reduce the amount of non-determinism and make the model more precise; such transformation may be appropriate for mobile code, but are unlikely to be appropriate for traditional host-based software because of legal and interoperability issues.

Other examples of research contributions to the area of intrusion detection via system calls not described above are briefly listed here: Bowen et al.’s (2000) research efforts in “Building survivable systems: an integrated approach based on intrusion detection and damage containment”; Spinellis (1994) involvement in “Trace: a tool for logging operating system call transactions”; Cabrera et al.’s (2001) attempts at “Detection and classification of intrusion and faults using sequence of system calls”; and finally Alexandrov et al.’s (1998) investigations of “Ufo: a personal global file system based on user-level extensions to the operating system”.

4. An overview of our research approach

In brief, our approach involves creating bogus DLLs that a target application will interact with via system calls. By doing this, we are able to log system call messages, distinguish abnormal from normal system call messages via heuristic methods and prevent attacks or exploitation of vulnerability that generate this abnormal system call messages. To achieve this, the main research questions we need to answer in this section are:

1. How do we determine DLLs used by a target application?
2. How do we hook into these DLLs to log relevant data within its functions?
3. What are the possible attack scenarios or intrusion exploits that a system call message could aid to detect and prevent?

We attempt to answer these questions in detail below.

4.1. Determining a target application’s DLLs

In locating a target application’s DLL, we start by introducing Portable Executable (PE) structure, since it is a possible answer. The term “Portable Executable” was chosen because the intent by Microsoft was to have a common file format, i.e., .exe, .dll, .ocx, .cpl; for all versions of Windows. Using Fig. 2 as a guide, every PE file starts with MS DOS Header executables that instruct the user on operational requirements.

![Fig. 2 — Portable Executable file format structure.](image)

The IMAGE_NT_HEADERS files found in the IMAGE_OPTION_HEADERS is the primary location where specifics of the PE file are stored. Immediately, following the IMAGE_NT_HEADERS field is the initialisation headers containing global variables and other parameters. The export fields include codes or data that will be used by other PE files. We refer to these codes or data as “symbols” for simplicity. Each exported symbol consists of an ordinal number associated with it that can be used to look it up from memory. The ordinal numbers of an exported symbol are stored in an Exported Address Table (EAT), in an array like manner. The EAT is located in the IMAGE_EXPORT_DIRECTORY section of the Export Table of a PE file. As an example, for a call to GetProcAddress on a AddAtomA API in Kernel32.dll, the operating system does a search for AddAtomA in Kernel32’s IMAGE_EXPORT_DIRECTORY. The search index at which AddAtomA is found in Entry Name Table (ENT) located in the IMAGE_EXPORT_DIRECTORY is the export ordinal value in Export Address Table (EAT), resulting in the actual address of AddAtomA being found. In summary, obtaining the PE file format for any of Microsoft files enables you to determine the DLLs that the file exports or call during operation.

The opposite of exporting a function or variable is importing it. The importing fields provide these facilities, using similar mechanism as the exporting fields described earlier; such as, IMAGE_IMPORT_DIRECTORY, Import Address Table (IAT) and Import Name Table (INT) including others. A PE file format consists of other fields that are not relevant to this research, as a result not discussed but some literature can be found at the work of Pietrek (2002).

There exist several utilities (ExeInfor, 2005) that are capable of inputting an exe or DLLs file and provide corresponding export functions or exports DLLs. Another method to determine export functions or associated DLLs of an application is to check Microsoft web site for its DLLshelp Database (2005). Table 2 shows a sample of DLLs related to Microsoft Internet Explorer obtained from the DLLshelp database.

4.2. Hooking into a target application’s DLLs

On knowing the DLLs, a monitored application will directly call, we now have to intercept all system calls made by the application to these DLLs. As an illustration, we initially created a bogus DLL file named wininet.dll that is originally supplied...
by Microsoft for its Windows operating system and called by Internet Explorer (IE). To ensure IE calls to our DLL we carried out the following: placed our DLL in the same directory as IE and placed Microsoft’s supplied DLL in another directory, altered registry value APPInit_DLLs to allow our DLL to be loaded with all Window based Applications, and also changed registry value ‘SFCDisable’ to disable the write protection of Microsoft’s system files.

As a result of our actions carried out above, once IE starts up our created DLL is loaded and if it receives any system call messages it uses GetSystemDirectory API to locate Microsoft’s supplied DLL and calls LoadLibrary function to load it. Our created DLL then uses GetProcAddress on each exported function or API of the loaded Microsoft supplied DLL to pass related arguments received by IE to these exported functions. The final task of our created DLL is then to return the values from these exported functions back to IE, while logging all system call messages. A sample of our created DLL code is provided in Fig. 3, while a diagrammatic illustration of our approach is shown in Fig. 4. By using this approach on a target application like IE, we can monitor and log system call messages made by the target application to DLL exported functions under innocuous and malicious conditions. There exist various vulnerabilities in IE that when exploited can result in malicious conditions or attack scenarios and as a result loss of data integrity, confidentiality and availability of resources.

### 4.3. Possible attack scenarios

To evaluate our research approach in detecting malicious exploit that are lunched at Microsoft IE, we used over 80 attacks that described in CERT database and Secunia Stay Secure (2005). A sample of the list of attacks used is shown in Table 3.

We categorise these attacks into three categories:

- **Buffer overflow attacks**: this may occur when an Internet browser interprets a script and exceeds one of its variable boundaries in doing so. Possible consequences of the attack include execution of malicious program, a targeted application crashing, corruption of memory and system access to name a few. Our research approach would be able to detect this category of attacks owing to the abnormal system calls or messages that will be observed during the malicious act.

- **Denial of service (DoS) attacks**: this may occur when Internet browser cannot provide normal quality of service (QoS) or a certain resource owing to an attack. Our approach using system calls will detect the degradation of the quality of system calls during Internet browser operation and alert this as an abnormality. In addition, we will also prevent the attack by not passing these abnormal system calls to respective Microsoft supplied DLLs or a target application.

Expressed more formally,

Let: 

\[ \text{DoS exploitation attack} = \text{EXEDoS}(\text{App}) \]

Where:

\[ \text{EXEDoS} = \text{execution of DoS exploit} \]
\[ \text{App} = \text{exploited application e.g Internet Explorer} \]

Assuming that:

- System calls pertaining to App = H
- Since the execution of an application is a series of system calls

Therefore:

\[ \text{Execution of App} = H_0 \ldots H_{\text{end}} \]

Where:

- \( H_0 \) is the initial system call
- \( H_{\text{end}} \) is the end of execution system call

We can then say:

\[ \text{EXEDoS}(\text{APPL}) = H_0 \ldots H_{\text{DoS/a}} \ldots H_{\text{DoS/b}} \ldots H_{\text{DoS/c}} \ldots H_{\text{DoS/crash-1}} \ldots H_{\text{DoS/crash}} \]

Where:

- \( H_{\text{DoS/a}} \ldots H_{\text{DoS/crash-1}} \) system calls leading to abnormal application crash.
- \( H_{\text{DoS/crash}} \) system call that crashes the application.

From the above statement, we can conclude that a resistance to DoS exploitation attack flaw monitoring the summation of H and stopping any series of system calls during Internet browser operation and alert this as an abnormality. In addition, we will also prevent false positives rate of attacks using the above system calls,

Based on the above categories of attacks, reported in the database (Secunia Stay Secure, 2005), we note that the above categories of attacks represent more than half of the entire attacks reported in Secunia Stay Secure (2005) for Internet browsers over the past few years.

### 5. Experiment methodology

In order for us to evaluate the performance of our target application IE when exposed to our research approach described in Section 4 of this paper to detect abnormal system calls created from malicious vulnerabilities being exploited, an experiment with the following objectives were carried out:

- determine specific system calls in IE that are unique to malicious exploit only,
- measure the performance of our research approach against false positives rate of attacks using the above system calls,

---

**Table 2 – Sample of Internet Explorer DLLs**

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advapi32.dll</td>
<td>Win32 ADVAPI32 core component</td>
</tr>
<tr>
<td>Browseui.dll</td>
<td>Shell browser UI Library</td>
</tr>
<tr>
<td>Comctl32.dll</td>
<td>Common controls library</td>
</tr>
<tr>
<td>Ledetect.dll</td>
<td>Internet Explorer detection</td>
</tr>
<tr>
<td>Ledks32.dll</td>
<td>Customisation DLL</td>
</tr>
<tr>
<td>Lpeers.dll</td>
<td>Peer objects</td>
</tr>
<tr>
<td>Hlink.dll</td>
<td>Hyper link library</td>
</tr>
<tr>
<td>Inetres.dll</td>
<td>Internet messaging API</td>
</tr>
<tr>
<td>Merrated.dll</td>
<td>Local user management DLL</td>
</tr>
<tr>
<td>Wininet.dll</td>
<td>Internet extension for win32</td>
</tr>
</tbody>
</table>
compare true positive rates of attack using our research approach and Snort IDS (Roesch, 1999).

To achieve these objectives the following equipments were used: a personal computer with the following facilities; 13 GB Hard Disk, 128 MB RAM, CPU Speed of 400 MHz, Internet Information Services (IIS), Perl Compiler, Windows Operating System and Snort network IDS. Several plug-ins were enabled, such as, activeX-control, scripting a file download, to make our experiments more representative of a real users.

5.1. Experiment procedure

During our experimental stage which took over a week, we subjected Microsoft’s IE to over 50 malicious and innocuous attacks that generated more than a 1000 system call messages. The main aim of this training period was to select specific system calls that were common to malicious but not to innocuous operations. Using Fig. 5 as an illustrative guide, we created over 50 bogus DLLs that Microsoft supplied for IE and subjected IE to malicious operations, while storing related system calls. We then compared these system calls with ones generated from innocuous IE operations over several weeks. We followed through by isolating system call messages from individual DLLs that were only common to malicious and not to innocuous operations. These common system calls were then stored as our Rule Based Intrusion Detection Expert System Database (RBIDES), to compare with future real-time system call messages generated from IE.

To fine-tune our IDES we exposed it to a fresh series of innocuous IE operations over a four-day period, 6 h a day, generating over 500 system calls per day to measure its false positives rate of attacks. The results of these experiments are presented in Fig. 6. Also, Fig. 6 shows a high rate of convergence: the amount of training time needed to achieve a given

**Table 3 – Ten of 83 IE attack samples in Secunia vulnerability inventory**

<table>
<thead>
<tr>
<th>Name</th>
<th>Impact</th>
<th>Secunia index</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE &quot;javaproxy.dll” exploit</td>
<td>System access</td>
<td>Secunia No.: SA15891</td>
</tr>
<tr>
<td>Utility library</td>
<td>DoS</td>
<td>Secunia No.: SA8642</td>
</tr>
<tr>
<td>DoS attack</td>
<td>System access</td>
<td>Secunia No.: SA7188</td>
</tr>
<tr>
<td>Bypass IE security zone</td>
<td>DoS</td>
<td>Secunia No.: SA15546</td>
</tr>
<tr>
<td>IE “Windows” DoS</td>
<td>System access</td>
<td>Secunia No.: SA13124</td>
</tr>
<tr>
<td>URL handle vulnerability</td>
<td>System access</td>
<td>Secunia No.: SA12959</td>
</tr>
<tr>
<td>HTML buffer overflow</td>
<td>System access</td>
<td>Secunia No.: SA12321</td>
</tr>
<tr>
<td>Drag and drop vulnerability</td>
<td>System access</td>
<td>Secunia No.: SA12321</td>
</tr>
<tr>
<td>URL obfuscation</td>
<td>Spooﬁng</td>
<td>Secunia No.: SA11582</td>
</tr>
<tr>
<td>IE long share name exploit</td>
<td>System access</td>
<td>Secunia No.: SA1482</td>
</tr>
<tr>
<td>URL process vulnerability</td>
<td>System access</td>
<td>Secunia No.: SA11067</td>
</tr>
</tbody>
</table>
level of false positives, because we included only directly associated DLLs. For instance, if non-associated DLLs had been included, several redundant system calls would have been invoked, thus, increasing overhead of training processes.

To evaluate the true positive rate of attacks detection ability of our research approach using IDES, we compared the performance of our approach with Snort network IDS and present experimental results in Fig. 7. All attempts were made to use the most recent Snort network IDS and related Snort rules for optimal performance during our experiments, in addition, configuration of Snort (2005) was done in accordance with published manuals and instructions from online community.

In general our approach or method presented in this paper outperformed Snort's pattern matching method. The main difference between both methods is that Snort uses a stringent pattern matching that searches for specific intrusive patterns that if found an alert is raised. Whereas, our research approach uses intrusive pattern marching of system calls, these system calls are derived from DLLs called at various operational stages of the malicious operations. As our research methodology, offers greater intelligence and flexibility in detecting true positives rate of attacks. IDSs like Snort that employ misuse intrusion detection systems, use pattern matching that can be rigid in their design of intrusive signatures and any variation in the intrusive signature eludes the IDS. While our approach investigates the detection of malicious application system calls generated via DLLs as a result of an attack, hence, it is more difficult to elude. Furthermore, a comprehensive coverage of associated DLLs for IE, our target application was used during the experiments, enabling a more accurate detection.

6. Conclusions

In this paper, we presented an approach in securing applications from the exploitation of malicious vulnerabilities. To achieve this, we base the foundation of our research approach on Forrest et al.’s (1996, 1998) publications; these publications proved that an application behaved differently when exposed to innocuous and malicious operations. We follow through with this research and use system call messages as our medium to distinguish between these two operations. In doing this, we presented an architecture of Windows win32 API that showed both the user and kernel mode layers, and related DLLs. In addition, we described possible methods of injecting a DLL into an application process. To highlight the novelty of our research approach, a comprehensive literature review in this subject area was carried out and compared with our research approach. Our research approach provided possible answers to these following research questions: how do we determine the DLLs related to an application, how do we inject a DLL into an application process and finally, what are the possible attack scenarios a system call message will aid to detect. To evaluate our research approach, investigative experiments were carried out with the following objectives: determine specific system call messages that are unique to certain malicious operations only, measure the performance of our research approach against false positives rate of attack and compare true positives rate of attack between our research approach and Snort IDS. Alongside our
experimental methodology, we also presented related experimental procedure and published our experimental results. Inferences from our experimental results showed that the number false positives rate of attack generated by our approach by the end of our training period was far less than the recommended number in DARPA 1999 (Haines et al., 2001), which was 10 per day. Further still, our approach outperformed Snort IDS in detecting true positives rate of attack in all attack scenarios they were subjected to.

The achilles heel of most research works in this subject area is the issue of impossible paths. This refers to the numerous DLLs a target application can utilise during an operation, as a result generate new system calls that were not seen during an IDS training period. We tackle this issue by using directly linked DLLs or static DLLs that subscribe to the target application PE file format and in addition, we included DLLs related to our target application that were presented in Microsoft’s DLLshelp database. Although the coverage of DLL used during our research outnumbers those used in our related work section, we fail to tackle exploitation of new DLLs vulnerabilities introduced by new target application plugins. Another thorn in this area of research are the race conditions – symbolic, relative path and argument described in “Traps and pitfalls: practical problems in system call interposition based on security tools” by Garfinkel (2003), we thwart this by only addressing DLLs directly related to our target application, in addition, we only pass and return their associated system call messages.

REFERENCES


Bowen T, Chee D, Segal M. Building survivable systems: an integrated based on intrusion detection and damage containment. In: DISCEx; 2000.


Cabrera BD, Lewis L, Mehr RJ. Detection and classification of intrusions and faults using sequence of system calls. SIGMOD 2001;30(4).


OSR Open System Resources Inc.. Nt vs. Zw – clearing confusion on the native API. The NT Insider August 2003;10(4).