A Framework for Detecting Network-based Code Injection Attacks Targeting Windows and UNIX

Stig Andersson, Andrew Clark, George Mohay, Bradley Schatz and Jacob Zimmermann
Information Security Institute, Queensland University of Technology
GPO Box 2434, Brisbane 4000, Australia
{sa.andersson, a.clark, g.mohay, b.schatz, j.zimmermann}@qut.edu.au

Abstract

Code injection vulnerabilities continue to prevail. Attacks of this kind such as stack buffer overflows and heap buffer overflows account for roughly half of the vulnerabilities discovered in software every year. The research presented in this paper extends earlier work in the area of code injection attack detection in UNIX environments. It presents a framework for detecting new or previously unseen code injection attacks in a heterogeneous networking environment and compares code injection attack and detection strategies used in the UNIX and Windows environments. The approach presented is capable of detecting both obfuscated and clear text attacks, and is suitable for implementation in the Windows environment. A prototype intrusion detection system (IDS) capable of detecting code injection attacks, both clear text attacks and obfuscated attacks, which targets Windows systems is presented.

1. Introduction

Code injection vulnerabilities continue to prevail despite increasing efforts to secure applications, by methods such as secure programming practices and patch distribution. It has been estimated that roughly half of the vulnerabilities discovered in software every year are code injection vulnerabilities. Code injection attacks operate by injecting executable code of the attacker's choice into a vulnerable process. The executable code consists of machine instructions for the targeted architecture. This code is referred to throughout this paper as shellcode, as historically the injected code was constructed to spawn a shell on the target system. The consequence of this type of attack is that the attacker can potentially run arbitrary code with the privileges of the vulnerable process.

A code injection attack injects some executable code through an injection vector manifesting itself as a vulnerability in the targeted process. Examples of attack strategies that utilise different injection vectors are the stack buffer overflow, heap buffer overflow and format string bugs. The aim of this process is to inject the attacker's executable code and gain control of the instruction pointer of the targeted process so that it points to, and thus transfers execution to the attacker's injected code.

In this paper we present a new network-based intrusion detection system (NIDS) approach for detecting new or previously unseen code injection attacks. The research presented in this paper extends earlier work [2] in the area of obfuscated code injection attack detection for the UNIX environment by presenting a framework for detecting both obfuscated and clear text code injection attacks in a heterogeneous environment. The requirements for detecting attacks targeting the Windows platform are described by comparing with the previous UNIX detection work. (We are collectively referring to Windows 2000, XP and 2003 as the Windows platform).

Section 2 discusses existing approaches for detecting code injection attacks. Sections 3 and 4 examine code injection attack strategies, and approaches for monitoring executable code in the Windows environment. Section 5 presents our framework for detecting the relevant attacks in a heterogeneous environment. Sections 6 and 7 presents our implementation of the Windows monitoring environment and test results for detecting attacks targeting Windows respectively. Finally conclusions and future work are presented in Section 8.

2. Related Work

This section examines a taxonomy of intrusion detection systems and existing approaches to detecting code injection attacks. The examined systems are discussed with regards to desirable properties identified in the taxonomy.

The goal of an intrusion detection system is to detect attacks against computer systems and networks. This func-

tionality is needed because of the difficulty in creating computer systems that are provably secure for all intended uses. Debar et al. have created a taxonomy of intrusion detection systems [6], by which they identify several measures of IDS efficiency. Primary measures are accuracy and completeness. These terms will be used throughout the paper as defined by Debar et al.

Accuracy provides an indication of the IDS's ability to identify malicious activity. An erroneous alert generated by an IDS is referred to as a false positive. Completeness refers to the proportion of all attacks an IDS is capable of detecting. The failure of an IDS to generate an alert in the presence of malicious activity is referred to as a false negative. The rest of this section reviews existing approaches to detect code injection attacks with focus on the accuracy and completeness properties of each approach.

Signature based IDSes operate by comparing network traffic to a database of signatures describing known attacks. If a signature matches the pattern of a sequence of bytes in the packet the packet is considered to represent malicious activity and an alert is generated. Signature based IDSes can have good accuracy characteristics and generate very few false positives provided that the signatures used are well formulated. However, in practise, attaining completeness is hampered by the following issues. Resent research in the area of IDS avoidance [9] has shown that producing exploits that avoid detection is trivial provided the signature describes the exploit rather than the vulnerability. The completeness property of a signature based system is also limited to known attacks since the system must have a signature for all malicious activity to provide complete coverage. Despite obvious shortcomings, signature based systems are the most commonly deployed intrusion detection systems and popular open source detectors such as Snort [5] are freely available.

The NOP detection approach programmatically identifies in network traffic a sequence of executable instructions which, when executed by a processor, have the effect of increasing the instruction pointer while producing negligible side effects on the state of the program. Examples of instructions searched for in this approach are the NOP, POP, INC, DEC and JMP instructions. The NOP detection approach provides an excellent completeness property for the UNIX environment since most UNIX code injection attacks rely on the presence of a NOP sledge. This is due to the difficulty in determining the location at which the exploit code will be injected in memory of target UNIX hosts (this is discussed in more detail in Section 3).

This detection approach does not however perform as well in the Windows environment as the NOP sledge technique is not typically needed. This is due to the more deterministic nature of memory layout on this platform. Windows is distributed as binary files, which in turn are produced by a smaller number of variants of compiler or programming environment, leading to predictable and reproducible determination of the location of instructions needed to seize control of the instruction pointer. Furthermore the accuracy of the NOP detection approach is poor because signatures matching NOP instructions also match amounts of normal traffic and therefore this approach generates scores of false positives.

The abstract payload execution detection approach [12] operates by searching for consecutive correct and valid machine instructions in network traffic. Correctness refers to bytes identified in the network stream that represent valid machine instructions and validity refers to valid memory references for instructions that operate on memory. This approach has been implemented with good completeness and accuracy properties for DNS and HTTP requests. We speculate that this approach is better suited for text based protocols utilising lower case ASCII characters than it would be for protocols that transmit binary data or protocols transmitting a large number of consecutive upper case ASCII characters since their hex value all represent valid single byte machine instructions.

Specification based systems provide excellent accuracy and completeness properties because they explicitly model all of the aspects of the protocols and application IO used on the network they protect. A specification based system may know that the input buffer an application or a field in a protocol should be 256 bytes long and therefore reject all data that exceeds the length of this input buffer. Overflowing a buffer in an application is therefore impossible without being detected and the system would never detect an attack without the presence of malicious traffic. This approach does however require that all protected applications and protocols be analysed with regard to the size of their input buffers and structure of protocols. This exercise may be impractical especially for applications whose source is not freely available.

The system call detection approach [1] operates by analysing network traffic searching for evidence of system call usage in the packet payloads. Code injection attacks that exploit UNIX hosts usually carry payloads with frequent use of system calls. This approach provides good completeness and accuracy properties for UNIX attacks that carry the payload in clear text. The approach is however vulnerable to attacks that carry obfuscated payloads. It is also not suited to Windows attack detection, as the majority of exploits operate by calling functions in Dynamic Link Libraries (DLLs) instead of interacting directly with the operating system (as discussed in Section 3).

The following section gives a background to code injection attack strategies. It examines how control is gained over the instruction pointer for Windows and UNIX environments and how the payload of the attack must be crafted

differently depending on the platform that is targeted for the attack.

3. Code Injection Attack Strategies

This section examines code injection attacks, the steps required to gain control of the instruction pointer by modifying the instruction pointer register (EIP) and the construction of the payload used in the attack. The approach favoured in the Windows environment is compared to that favoured in the UNIX environment.

3.1. Seizing Control of the Instruction Pointer

Code injection attacks represent a group of attacks that attempt to break the execution path of a process by seizing control of the instruction pointer so that the attacker's injected code is executed. This is achieved by inserting some executable code through an injection vector. The stack overflow, heap overflow and format string bugs all represent injection vectors for code injection attacks. By employing an attack of this type the attacker may potentially execute arbitrary code with the privileges of the vulnerable process. The injected code may consist of machine instructions produced by compiling a program for the operating system and architecture of the attacked system or it may contain addresses of instructions on the system the attacker wishes to execute. In the latter case the attack is referred to as a *return-into-libc* style attack.

The stack overflow is the simplest code injection vector. The attacker's code is injected into a buffer on the stack and the return address of the vulnerable function is overwritten to execute the attack's payload. The difficulty in performing an attack of this type is to identify the new address to overwrite the vulnerable function's return address with. There are two solutions to this problem:

- A NOP sledge may be employed to increase the address space of the entry point of the executable code.
- The exact location of an instruction that returns execution to the injected code must be used.

In the Windows environment, the latter of the two solutions is the favoured approach since it minimises the size of the injected code and therefore also the required buffer space. This approach does however require that the exact address of the needed instruction be known, which is in the case of Linux not reliably determined, due to the following. Linux systems come in many flavours. There are different distributions and versions of software and libraries used and they have been compiled using different compilers and different versions of compilers. Resultantly, it is nearly impossible to exactly determine both what the absolute layout of the binary image of a process loaded in memory is, and where the individual instructions are absolutely located, in a widely applicable way. The NOP sledge approach is therefore favoured in most UNIX environments. By supplying a long sequence of operations that logically perform no operations before the payload and an equally long sequence of return addresses after the payload, it is sufficient to guess the return address with a few hundred bytes accuracy.

The Windows environment is different because it has been distributed in binary files. The version of the software may be determined and it has been compiled using the same compiler for each distribution. The locations where individual DLL files are held in memory are largely deterministic, making use of a NOP sledge unnecessary. The return address of the vulnerable function may be overwritten with an address of an instruction in a DLL file that returns to a register pointing to the injected code on the stack.

Once the instruction pointer has been seized, execution is transferred to the payload of the exploit. The following section describes how the API design differences between UNIX and Windows effects the construction of these payloads.

3.2. Attack Payload Construction

The classic architecture of operating system kernel and protected user space processes is shared by both the Windows NT family and UNIX. Processes running in user space are protected from modifying each other and modifying the kernel by allocating separate and protected memory partitions, and by providing a single strictly policed interface to kernel services, which in the UNIX world is generally called the system call interface, and the Windows world the Native API.

Calling from user space code to kernel space is achieved by CPU specific techniques coded in assembly language, and as such, a library is usually provided which contains C language proxy functions to the system calls. In the UNIX programming environment these system call functions are usually provided as a subset of the C Standard Library (libc). In the Windows environment the Native API (often called NTAPI) is provided by the NTDLL.dll library.

When we consider the relationship between the APIs which most system applications are written in with the system call interface APIs on both platforms, we see a marked divergence. On the UNIX platform, the system call interface is generally well documented and readily available via the standard C library API. The windows system call interface, the Native API, is however poorly documented and hidden from the average programmer beneath a number of higher level APIs, such as WIN32, POSIX and OS/2. It is the first of these API's which has been promoted most by Microsoft, and has consequently become the most widely

adopted API by developers on the Windows platform.

This has led to marked differences in the way that exploits are written between the UNIX and WIN32 platforms. While UNIX exploit writers have on a whole used the system calls directly to their advantage, Windows exploit writers have to date avoided writing exploits directly to the NTDLL system call interface, and instead used a number of techniques for calling higher level API library code to achieve their goals. For example, the Slammer worm uses the *GetTickCount* function from the kernel32.dll library, and the functions *socket* and *sendto* from the *winsock* Windows sockets library directly.

The following section examines different strategies for how the execution of processes may be monitored in Windows. Tracing code execution is the foundation of our approach to detecting code injection attacks from the network.

4. Windows Process Tracing

In earlier work [2] we have demonstrated that sandboxing and executing potentially malicious code in a monitored environment may provide excellent accuracy and completeness properties for intrusion detection. The prototype implementation for the Linux operating environment provided complete coverage where all code injection attacks including attacks using polymorphic payloads were detected and no false positives have been recorded. This section examines how a similar system may be implemented for the Windows operating environment by examining how processes may be traced in Windows.

When we consider tracing the program execution of a particular piece of code, and more specifically, code which may potentially be an exploit, it is important to consider the environment that the code will be executing in. The principle characteristics of the environment that are relevant to our discussion are: whether the exploit code runs from user process space or kernel space; what the contents of the stack are; what the contents of the heap are; and which code libraries are loaded.

In our previous work [2], we took the approach of tracing the system call API on a Linux system by using the *ptrace* facilities of the operating system. This enabled us to capture a trace of what system calls were made by the code in question. Applying this approach makes it straightforward to determine the operation of a piece of code and the basic operation of potentially malicious injected executable instructions.

In contrast to this approach, most attempts at tracing APIs on windows have grown out of developers' desires to "hook" specific API calls, for reasons such as implementing on-demand virus scanning and adding functionality to closed source binary code. Due to the poorly documented NTAPI, these efforts have concentrated on tracing calls at the more widely understood WIN32 API level, and have required the combination of code injection and hooking techniques.

Code injection in the context of execution tracing refers to both the means of getting executable code into the address space of the subject program, and the means of ensuring the tracing functionality is executed to produce the tracing functionality. One of the simplest ways of tracing is through the Proxy DLL technique. This involves generating a DLL which contains on a one to one basis all of the functions, symbols and resources exported by the real API implementation library in question. The implementation of these functions performs a logging operation, then redirects (or proxies) the call to the real implementation. A proxy DLL is introduced into a program as the real library by naming the proxy DLL the same as the DLL in question. By nature of the dynamic linker of Windows, DLL's are searched for first in the directory of the main executable of the program, then by the system path, so the proxy DLL will be loaded instead of the real implementation. This technique combines elements of code injection and hooking and is illustrated the user space region of Figure 1 where the proxy DLL functions are represented by a grey box.

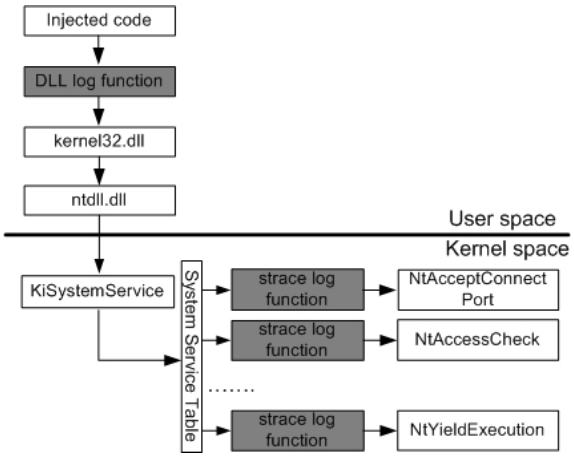

Figure 1. Detours vs strace tracing method

Other more advanced injection techniques include using the AppInit_DLLs Registry, the System Wide Hook *SetWindowsHookEx*, and Thread Injection techniques. However these more advanced techniques also require active modification (or patching) of the API calls through the use of Import Address Table (IAT) patching or API implementation patching. A good overview of these techniques is contained in [8]. We use the Detours injection and hooking library provided by Microsoft Research as the basis of our API spy utility. The Detours library implements a number of strategies mentioned above for providing injection and hooking services to applications.

In more recent times, the NTAPI has begun to receive more attention from security researchers, due to contributions from the likes of Russinovich [10] and Schreiber [11]. Similar to the *ptrace* technique on Linux for tracing system calls, a simple form of tracing of system calls on Windows has been identified and has recently culminated in the release of *strace* for NT, which is named after the Linux system call trace program with the same name. *strace* operates by patching the function pointers in the system call table in the OS kernel to point to functions that log the system call and then in turn calls the real system call. The approach is illustrated the kernel space region of Figure 1 where the functions inserted by *strace* are represented by grey boxes.

Very little work in the area of low level system monitoring for intrusion detection in the Windows environment has been presented by the research community. One existing system is the Windows Host Intrusion Prevention System (WHIPS) [4]. Their approach is based on monitoring security critical system calls and applying access control mechanisms by checking their parameters using a reference monitor at call time. The system calls that are monitored are: *NtCreateFile, NtDeleteFile, NtClose, NtOpenProcess, NtLoadDriver, NtUnloadDriver* and *NtOpenFile*. The parameters to the individual system calls are paramount to the functionality of this system. Our approach only relies on the existence of operating system resource usage in a section of code and therefore we examine the extended subset of system calls provided by strace.

The following section presents our detection framework for networks containing both Windows and UNIX hosts.

5. A Generic Detection Framework

The framework for detecting code injection attacks consists of an executable instruction detector residing on an IDS and a series of sandbox execution environments matching the monitored application environments provided by the operating systems in the network. Once the IDS has identified a series of executable instructions in network traffic it relays the suspicious packet's payload to an execution environment matching the packet's destination. The appropriate execution environment is identified by examining the destination IP address of the incoming packet. The payload is executed in the corresponding monitored environment and a report containing the payload's OS resource usage is returned to the IDS. The IDS generates appropriate alerts upon receiving the report from the execution environments if the report contains evidence of resource usage. Otherwise the packet is considered non-malicious.

Detecting code injection attacks by using a sandbox requires three steps [2]. First the exact entry point or the first executable instruction of the code must be identified. Including a byte that is not part of the injected code or omitting instructions at the beginning of the code may result in invalid machine instructions and execution failure. Secondly, if the code has been protocol encoded, the protocol data must be removed from the executable instructions to obtain the original code or execution will fail. Again, bytes that are inserted into the payload may represent illegal machine instructions at execution time and lead to execution failure. Finally the code must be executed in a monitored environment to gather evidence of operating system resource usage. The first two steps, the executable instruction identification and the protocol decoding are, in our framework, carried out by the IDS. The packet payload is then relayed to the execution environment where the third step, the execution monitoring, is performed.

The research presented in this paper relies on the existence of an efficient method of identifying executable instructions in a network stream and does not further examine approaches to solve this problem. In our implementation we utilise the Snort IDS together with the NOP detector Fnord to identify potential executable code. Fnord is a Snort pre-processor that programmatically detects NOP sledges by counting instructions that logically perform no operations that impacts the executing code. We set the threshold low so that if fnord detects four consecutive NOP instructions the packet is considered to require further analysis. We recognise that this approach is not capable of detecting all Windows attacks that return into DLL files and that does not include any NOP instructions but demonstrate that the approach is capable of detecting a number of Windows code injection attacks. Upon detection of a sequence of NOP instructions the IDS carries out a number of steps. First the appropriate execution environment is selected based on the destination IP address of the packet. Secondly the appropriate protocol decoding is applied to the payload based on the destination port number of the packet and finally the entire packet payload is sent to our analyser starting from the first instruction identified as being part of the executable code.

Correct configuration of the execution environments is important to provide accurate detection results. Both the architecture and the operating system of the execution environment must match that of the destination of the incoming payload. The reason for this is that machine instructions compiled for SPARC for example are different to instructions compiled for the i386 architecture, and if executed on a different architecture than it was compiled for will only represent illegal instructions. The operating system of the execution environment must also match because resource usage may be implemented differently from operating system to operating system.

Configuration of the Windows environment is more critical than that of the UNIX environment if a full trace of the injected code is to be obtained. The reason for this is that the injected code utilises user space APIs that may dif-

fer from release to release. Since the UNIX system call API is available directly to the user it is pretty much cast in stone. If it is changed all applications must be rewritten. The Windows operating system has been designed to only export the higher level APIs. This leaves the creators free to change any implementation as long as the higher level APIs provide the services provided at the previous release. Certain systems may also have DLL files that are missing on other systems. Due to these issues the execution environment must match that of the protected host as close as possible. Especially, the DLL files provided on the protected system must also reside on the monitoring environment.

The following section examines how the detection system is implemented to capture traffic on the network, analyse the payload of the packet to determine whether or not it may contain executable code, execute and monitor the execution to detect potentially executable code in a Windows environment.

6. Windows Monitoring Environment Implementation

The analyser resides on the Windows test environment and is located on a host separate from the IDS. Its function is to execute the potentially executable code, monitor its execution and generate a log that the IDS can evaluate to determine whether or not the packet contains executable instructions.

Our implementation of the Windows execution environment is implemented using the VMware GSX Server and Windows 2000 Professional is installed as a virtual machine. VMware provides features that are desirable for this application. A snapshot feature is provided which enables us to create a backup of the system when it is known to be in a safe state. The safe state may then be restored at an arbitrary point in time. VMware can be configured to revert to the snapshot instead of shutting down. The GSX version also has a perl API that makes it possible to build applications capable of reverting the state of the virtual machine. Using this feature we are firstly guaranteed that the system is restored to the known safe state if some malicious code attempts to shut the environment down. Secondly the IDS is able to restore the known safe state of the execution environment at any given time.

Ideally the known safe state should be restored after every single execution. This operation however does incur substantial overhead, so unless the system is implemented with a cluster of identical execution environments, service would be lost for a few seconds. We therefore create a separate user on the Windows execution environment with restricted privileges that executes the potentially executable instructions received from the IDS. The restricted user provides added protection from malicious code due to its limited privileges. A restricted user may use the operating environment and save files but does not have sufficient privileges to install software or make changes to system files or settings or add new users to the system. Despite the fact that processes in the execution environment are running as unprivileged processes, we are dealing with untrusted code and measures must be taken to recover from unforeseen events. Should some malicious code shut down the system, VMware will revert it to the known state. This will cause loss of service from the execution environment for a few seconds. The IDS should also return the execution environment to the known safe state with regular intervals to compensate for potential Trojans on the system. The time spent returning the environment to the initial state is relatively short, but to provide the service at all times a secondary system is needed to fall back on during these time intervals.

A network server is running in the test environment listening on a port known to the IDS. The server accepts connections from the IDS once some potentially executable instructions have been identified and the IDS transmits the potentially executable instructions to the server. The server creates a process containing the potentially executable instructions and executes it. The execution is monitored using the monitoring approaches identified in Section 4.

Tracing at the NTAPI level initially promised to be a compelling technique, promising both a simple implementation of API hook functions and the isolation of trace implementation away from exploit code. Since the tracing is implemented in the kernel, there would be no way for the code to introspect that it was being traced. However, in practise, we observed that there was no readily observable way to relate the NTAPI logs to higher level goals such as TCP *bind*, *listen*, *accept*, etc. Furthermore, *strace* requires the loading of drivers to modify the system call table. This operation complicates the implementation of the execution environment if the potentially executable code is to be executed with the privileges of a restricted user, as this user does not have sufficient privileges to perform this task.

The entries in Table 1 compare log entries generated by Detours and *strace*. *strace* provides a much more granular trace of the payload execution. The trace using this method only includes calls to the operating system, and provides no tracing of user space function calling. The log entries contained in Table 1 represent a trace of the *ws2_32.WSASocketA* call used to create a socket. This API call in turn calls *LoadLibraryExW* to load msafd.dll.

Although the trace produced by *strace* is much more granular than that produced by Detours, it does not provide us with any more detail useful for analysis. The information provided by Detours is easier to analyse because the log entries represent the more widely known programming API provided by Windows. Under certain circumstances *strace*

Table 1. Detours vs strace trace

Detours

```
ExpandEnvironmentStringsA(%SystemRoot%\system32\msafd.dll,12f570,104)
ExpandEnvironmentStringsA(,C:\WINNT\system32\msafd.dll,) -> 1c
LoadLibraryA(C:\WINNT\system32\msafd.dll)
LoadLibraryExA(C:\WINNT\system32\msafd.dll,0,0)
LoadLibraryExW(C:\WINNT\system32\msafd.dll,0,0)
```

Strace

```
NtFsControlFile (24,0,0x0,0x0,0x90028,0x0,0,0, ...  {status=0x0, info=0}, 0x0, ) == 0x0
NtQueryAttributesFile ({24,0,0x40,0,0,"\??\C:\WINNT\system32\msafd.dll"},1240756,..)==0x0
NtFsControlFile (24,0,0x0,0x0,0x90028,0x0,0,0,...{status=0x0, info=0}, 0x0, ) == 0x0
NtFsControlFile (24,0,0x0,0x0,0x90028,0x0,0,0,...{status=0x0, info=0}, 0x0, ) == 0x0
NtOpenFile (0x100020,{24,0,0x40,0,0,"\??\C:\WINNT\system32\msafd.dll"},5,96,...80, {status=0x0,
info=1}, ) == 0x0
NtCreateSection (0xe, 0x0, 0x0, 16, 134217728, 80, ...  84, ) == 0x0
NtClose (80, ...  )  == 0x0
NtMapViewOfSection (84, -1, (0x0), 0, 0, 0x0, 0, 1, 0, 16, ...  (0x450000), 0x0, 57344, ) == 0x0
NtClose (84, ...  )  == 0x0
NtUnmapViewOfSection (-1, 0x450000, ...  )  == 0x0
NtFsControlFile (24, 0, 0x0, 0x0, 0x90028, 0x0, 0, 0, ...  {status=0x0, info=0}, 0x0, ) == 0x0
NtQueryAttributesFile ({24, 0, 0x40, 0, 0, "\??\C:\WINNT\system32\msafd.dll"}, 1241316, ...  )  == 0x0
NtFsControlFile (24, 0, 0x0, 0x0, 0x90028, 0x0, 0, 0, ...  {status=0x0, info=0}, 0x0, ) == 0x0
NtFsControlFile (24, 0, 0x0, 0x0, 0x90028, 0x0, 0, 0, ...  {status=0x0, info=0}, 0x0, ) == 0x0
NtOpenFile (0x100020, {24, 0, 0x40, 0, 0, "\??\C:\WINNT\system32\msafd.dll"}, 5, 96, ...  84,
{status=0x0, info=1}, ) == 0x0
NtCreateSection (0xf, 0x0, 0x0, 16, 16777216, 84, ...  80, ) == 0x0
NtClose (84, ...  )  == 0x0
NtMapViewOfSection (80, -1, (0x0), 0, 0, 0x0, 0, 1, 0, 4, ...  (0x74fd0000), 0x0, 69632, ) == 0x0
NtClose (80, ...  )  == 0x0
NtOpenSection (0xe, {24, 20, 0x40, 0, 0, "USER32.DLL"}, ...  80, ) == 0x0
NtMapViewOfSection (80, -1, (0x0), 0, 0, 0x0, 0, 1, 0, 4, ...  (0x77e10000), 0x0, 413696, ) == 0x0
NtClose (80, ...  )  == 0x0
NtOpenSection (0xe, {24, 20, 0x40, 0, 0, "GDI32.DLL"}, ...  80, ) == 0x0
NtMapViewOfSection (80, -1, (0x0), 0, 0, 0x0, 0, 1, 0, 4, ...  (0x77f40000), 0x0, 245760, ) == 0x0
NtClose (80, ...  )  == 0x0
```

may miss events completely. During monitoring tests using known exploits, *strace* did not generate any log entries for the execution of *bind, listen* or *accept*. The only instance *strace* can provide information Detours is incapable of is a attack where the whole shellcode is coded to use the Native API as recently demonstrated by Bania [3]. Encoding all the instructions needed to accomplish something useful using this API would require a larger input buffer and due to space limitations when exploiting a vulnerable process this method has to date not been favoured. Furthermore, writing shellcode using the native API limits exploit compatibility between versions of Windows. Therefore, due to the difficulty in analysing execution traces generated by *strace* and the fact that certain events valuable for analysis are completely missed, execution tracing using Detours is the favoured approach for our prototype system.

The process executing the potentially executable instructions may be terminated in two ways. Firstly the execution may finish normally. In this case all the instructions have been processed and the complete trace of the execution has been captured by the execution monitor. Some processes however will never finish and a second approach to terminate execution is required. An example of code that will never finish is code that starts a server listening for incoming connections. The execution will in this circumstance hang on the *accept* function call. We therefore terminate all monitored processes after the execution time exceeds a set threshold of five seconds. In this case the captured log may only contain parts of the instructions contained in an incoming packet but provides sufficient information for analysing the operations of the attack.

Once the execution is finished, the log is relayed back to the IDS. The IDS analyses the log searching for evidence of calls to DLL functions. A single function call is sufficient to determine that the incoming packet contained shellcode formatted executable instructions, and an alert is in this case generated. The following section contains test results gathered using the Windows execution environment to detect code injection attacks.

7. Test Results

We have tested our detector on a variety of exploits targeting Windows hosts. The NOP detection method that we

employ is better suited for the UNIX environment due to the possibility of using *return-into-DLL* style attack patterns to gain control of the instruction pointer in the Windows environment as discussed in Section 3.1. Therefore, although our system is highly accurate in its detection and generates few false positives, the completeness property of the system will never be better than its capability of detecting potential executable code in the network traffic stream. The approach will benefit from a more accurate algorithm for detecting potential executable code in network traffic than the NOP detection algorithm provides and once such an algorithm is identified the completeness property of the system will be further improved. Despite this fact our approach is capable of detecting a number of attacks. Examples of attacks detected are buffer overflow exploits against the RPCSS service (MS03-039), Microsoft FrontPage Server extensions (MS03-051) and WINS (MS04-045). We now examine the exploit against Microsoft FrontPage Server extensions with regards to its operation and detection.

The exploit targeting Microsoft FrontPage Server extensions overflows a buffer in fp30reg.dll with executable code that binds a shell on port 9999 on the vulnerable host. The attacker is then able to log in by using telnet, browse the file system and perform actions identical to a user having a *cmd* shell locally on the host. Parts of the machine instructions for the exploit are contained in Table 2. The first part of the exploit contains a decoding function which decodes the payload of the attack. This function serves two purposes. Firstly it ensures that the payload does not contain any terminating characters (NULL characters) when the attack is transmitted across the network. Secondly it obfuscates the payload of the attack and complicates pattern matching IDSes and IDSes utilising the abstract execution approach. The decoding function has been bolded in Table 2. The bolded decoding function XORs the rest of the payload with a key, in this case 88h, to obtain the clear text payload. The assembly instructions are contained in Table 3.

Because our approach is based on executing malicious code in a sandbox, payload obfuscation techniques are rendered useless as an IDS avoidance measure. Once the decoding function has completed the attack payload is ready to be executed. First the base address of kernel32.dll is located. Then a call table is constructed and the address of *kernel32.LoadLibraryA* and ws2_32.dll is identified. The address of ws2_32.dll is used to initialise the WinSock API. Once this has been completed the rest of the call table is populated with addresses of functions used later during execution. These functions include *kernel32.WaitForSingleObject, kernel32.CreateProcessA, kernel32.ExitThread, ws2_32.WSAStartup, ws2_32.WSASocketA, ws2_32.bind, ws2_32.listen, ws2_32.accept* and *ws2_32.closesocket*. *ws2_32.WSAStartup* is called to initialise the WinSock API and then a socket is created using *ws2_32.WSASocketA*. *Bind, listen* and *accept* is called to bind the socket to port 9999, listen for incoming connections and accept connections from the attacker at a later stage.

The DLL hooking method implemented using Detours [7] logs resource usage by identifying calls to DLL functions. Table 4 contains some of the library calls the attack executes during runtime. The library trace starts with the *ws2_32.WSASocketA* call to create the socket and ends with the *ws2_32.accept* call when the process is in the state waiting for the attacker to connect to the host.

The log positively identifies the attack and provides sufficient information to identify the operations of the attack. DLL files loaded during execution are identified as well as a socket creation and subsequent calls to *bind, listen* and *accept*. The execution trace ends with the call to *accept* which indicates that the process is in a state waiting for incoming connections. This is sufficient to determine that the exploit attempts to start a server. Table 5 contains a list of the exploits we have tested together with the vulnerability they exploit and information whether the attack was detected or not. For attacks that could not be detected, detection failed because we could not identify the first executable instruction accurately. In these cases the first instruction of the exploit code was inserted directly into the analyser to generate the function call log. By analysing the logs generated, the first exploit adds an administrator user to the vulnerable host, exploits 2, 3 and 4 spawn a listening shell on ports 5555, 9999 and 101 respectively, and exploit 5 and 6 attempt to connect to an outside host to download more executable code.

The shellcode sandboxing approach has proven to have excellent accuracy and completeness properties when implemented correctly. During testing for the Linux operating environment all attacks tested were detected as well as all code injection attacks contained in the DARPA intrusion detection evaluation dataset without any false positives [2]. False positives are unlikely events when this approach is applied due to the difference in appearance of shellcode and normal binary executables. Shellcode has higher density of calls and due to its size long jumps never occur except for calls directly to DLL functions. We have however tested the detector on 3GB of Windows binary file transmission using the NOP executable instruction detection approach without recording any false positives.

8. Conclusion and Future Work

We have presented a framework for detecting code injection attacks in a heterogeneous environment. The framework is able to detect new or previously unseen attacks regardless of obfuscation techniques applied. The ap-

Table 2. fp30reg.dll exploit

```
\xEB\x03\x5D\xEB\x05\xE8\xF8\xFF\xFF\xFF\x8B\xC5\x83\xC0\x11\x33
\xC9\x66\xB9\xC9\x01\x80\x30\x88\x40\xE2\xFA
\xDD\x03\x64\x03\x7C\x09\x64\x08\x88\x88\x88\x60\xC4\x89\x88\x88
.......
```

Table 3. Decoding routine

Memory Address	Machine Instruction	Assembly	Comment
00421A41	66 B9 C9 01	MOV CX, 1C9	Length of payload
00421A45	80 30 88	XOR BYTE PTR DS: [EAX], 88	Key is 88H
00421A48	40	INC EAX	
00421A49	E2 FA	LOOPD SHORT 00421A45	

proach consists of an IDS that identifies potentially executable instructions and sends the payload of the packet to a monitored execution environment to determine whether the packet indeed contains executable code. The appropriate execution environment is selected based on the IP address of the incoming packet.

The execution environment for Windows code is also presented in this paper. We have compared two tracing approaches and determined that the DLL hooking approach currently performs better than that of the Native API monitoring approach. However, since attacks using only the native API may be missed, future work includes further examination of methods to perform system call monitoring in the Windows environment. The prototype implementation of the system is capable of detecting Windows code injection attacks as long as the IDS is able to identify executable instructions in a packet. Although the accuracy of the system is very good, the completeness property will never be better than the ability of the IDS to identify executable instructions. The NOP detection approach, although having moderate success during testing, does not identify all code injection attacks targeting Windows hosts. Future work includes identifying a more efficient algorithm to detect potentially executable instructions in a network stream so that the completeness property of the system can be further increased.

References

[1] S. Andersson, A. Clark, and G. Mohay. Network based buffer overflow detection by exploit code analysis. In G. Mohay, A. Clark, and K. Kerr, editors, *Proceedings of AusCERT Asia Pacific Information Technology Security Conference (AusCERT2004): R&D Stream*, Gold Coast, Australia, 2004. University of Queensland. ISBN: 1-86499-774-5.

[2] S. Andersson, A. Clark, and G. Mohay. Detecting network-based obfuscated code injection attacks using sandboxing. In A. Clark, K. Kerr, and G. Mohay, editors, *AusCERT Asia Pacific Information Technology Security Conference (AusCERT2005): Refereed R&D Stream*, Gold Coast, Australia, 2005. University of Queensland. ISBN: 1-86499-799-0.

[3] P. Bania. Windows syscall shellcode. http://www.securityfocus.com/infocus/1844, 2005. Accessed: 16/09-05.

[4] R. Battistoni, E. Gabrielli, and L. V. Mancini. A host intrusion prevention system for windows operating systems. In *ESORICS*, volume 3193 of *LNCS*, pages 352–369. Springer-Verlag, 2004.

[5] J. Beale, J. C. Foster, J. Posluns, and B. Caswell. *Snort 2.0 Intrusion Detection*. Syngress Publishing, Rockland, 2003.

[6] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31, 1999.

[7] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, WA, 1999.

[8] Y. Kaplan. API spying techniques for Windows 9x, NT and 2000. http://www.internals.com/articles/apispy/apispy.htm, 2000. Accessed: 16/09-05.

[9] D. Mutz, C. Kruegel, W. Robertson, G. Vigna, and R. Kemmerer. Reverse engineering of network signatures. In A. Clark, K. Kerr, and G. Mohay, editors, *AusCERT Asia Pacific Information Technology Security Conference (AusCERT2005): Refereed R&D Stream*, Gold Coast, Australia, 2005. University of Queensland. ISBN: 1-86499-799-0.

[10] M. Russinovich. Inside the Native API. http://www.sysinternals.com/Information/NativeApi.html, 2004. Accessed: 16/09-05.

[11] S. B. Schreiber. *Undocumented Windows 2000 Secrets*. Addison Wesley, Upper Saddle River, 2001.

[12] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances in Intrusion Detection*, volume 2516 of *Lecture Notes in Computer Science*, pages 274–291, Zurich, Switzerland, 2002. Springer.

Table 4. Detours exploit trace

```
WSASocketA(2,1,0,0,0,0)
   WSASocketW(2,1,0,0,0,0)
    .
     LoadLibraryA(C:\WINNT\system32\msafd.dll)
       LoadLibraryExA(C:\WINNT\system32\msafd.dll,0,0)
         LoadLibraryExW(C:\WINNT\system32\msafd.dll,0,0)
         .
         LoadLibraryExW(„) -> 74fd0000
       LoadLibraryExA(„) -> 74fd0000
     LoadLibraryA() -> 74fd0000
     GetProcAddress(74fd0000,WSPStartup)
     GetProcAddress(,) -> 74fd1f1c
     GetModuleFileNameA(74fd0000,12f3fc,104)
       GetModuleFileNameW(74fd0000,131fc0,104)
       GetModuleFileNameW(,C:\WINNT\system32\msafd.dll,) -> 1b
     GetModuleFileNameA(,C:\WINNT\system32\msafd.dll,) -> 1b
     LoadLibraryA(C:\WINNT\system32\msafd.dll)
       LoadLibraryExA(C:\WINNT\system32\msafd.dll,0,0)
         LoadLibraryExW(C:\WINNT\system32\msafd.dll,0,0)
           GetEnvironmentStringsW()
           GetEnvironmentStringsW() -> =::=::\
         LoadLibraryExW(„) -> 74fd0000
       LoadLibraryExA(„) -> 74fd0000
     LoadLibraryA() -> 74fd0000
     ExpandEnvironmentStringsW(%SystemRoot%\System32\wshtcpip.dll,13a9d0,104)
     ExpandEnvironmentStringsW(,C:\WINNT\System32\wshtcpip.dll,) -> 1f
     LoadLibraryW(C:\WINNT\System32\wshtcpip.dll)
       LoadLibraryExW(C:\WINNT\System32\wshtcpip.dll,0,0)
         GetEnvironmentStringsW()
         GetEnvironmentStringsW() -> =::=::\
         DisableThreadLibraryCalls(75010000)
         DisableThreadLibraryCalls() -> 1
       LoadLibraryExW(„) -> 75010000
     LoadLibraryW() -> 75010000
     GetProcAddress(75010000,WSHOpenSocket)
     .
     GetProcAddress(75010000,WSHIoctl)
     GetProcAddress(,) -> 75012d73
     ..
   WSASocketW(„„,) -> 74
 WSASocketA(„„,) -> 74
 bind(74,12fcc4,16)
   ntohs(f27)
     htons(f27)
     htons() -> 270f
   ntohs() -> 270f
 bind(„) -> 0
 listen(74,0)
 listen(,) -> 0
 accept(74,12ffb0,0)
   WSAAccept(74,12ffb0,0,0,0)
```

Table 5. Exploit table

Vulnerability	Exploit	Source	Detected
MS03-039	/ExploitTree/system/microsoft/remote/MS03-039-exp.c	www.securityforest.com	Yes
MS03-049	/ExploitTree/system/microsoft/remote/MS03-049ex.c	www.securityforest.com	No
MS03-051	/ExploitTree/application/webserver/iis/fp30reg.c	www.securityforest.com	Yes
MS04-011	/ExploitTree/system/microsoft/remote/ms04011lsass.c	www.securityforest.com	No
MS04-011	Sasser.B	NA	Yes
MS04-045	WINS Remote Heap Overflow Exploit	www.hat-squad.com	Yes