Comprehensive Shellcode Detection using Runtime Heuristics

Michalis Polychronakis
Columbia University, USA
mikepo@cs.columbia.edu

Kostas G. Anagnostakis
Niometrics, Singapore
kostas@niometrics.com

Evangelos P. Markatos
FORTH-ICS, Greece
markatos@ics.forth.gr

ABSTRACT

A promising method for the detection of previously unknown code injection attacks is the identification of the shellcode that is part of the attack vector using payload execution. Existing systems based on this approach rely on the self-decrypting behavior of polymorphic code and can identify only that particular class of shellcode. Plain, and more importantly, *metamorphic* shellcode do not carry a decryption routine nor exhibit any self-modifications and thus both evade existing detection systems. In this paper, we present a comprehensive shellcode detection technique that uses a set of runtime heuristics to identify the presence of shellcode in arbitrary data streams. We have identified fundamental machine-level operations that are inescapably performed by different shellcode types, based on which we have designed heuristics that enable the detection of plain and metamorphic shellcode regardless of the use of self-decryption. We have implemented our technique in Gene, a code injection attack detection system based on passive network monitoring. Our experimental evaluation and real-world deployment show that Gene can effectively detect a large and diverse set of shellcode samples that are currently missed by existing detectors, while so far it has not generated any false positives.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software

General Terms

Security

Keywords

Shellcode Detection, Payload Execution, Code Emulation

1. INTRODUCTION

Code injection attacks have become one of the primary methods of malware spreading. In a typical code injection attack, the attacker sends a malicious input that exploits a memory corruption vulnerability in a program running on the victim's computer. The injected code, known as *shellcode*, carries out the first stage of the attack, which usually involves the download and execution of a malware binary on the compromised host.

Once sophisticated tricks of the most skilled virus authors, advanced evasion techniques like code obfuscation and polymorphism are now the norm in most instances of malicious code [19]. The wide availability of ready-to-use shellcode construction and obfuscation toolkits and the discovery rate of new vulnerabilities have rendered exploit or vulnerability specific detection techniques ineffective [31]. A promising approach for the generic detection of code injection attacks is to focus on the identification of the shellcode that is indispensably part of the attack vector, a technique initially known as abstract payload execution [33]. Identifying the presence of the shellcode itself allows for the detection of previously unknown attacks without caring about the particular exploitation method used or the vulnerability being exploited.

Initial implementations of this approach attempt to identify the presence of shellcode in network inputs using static code analysis [33–35]. However, methods based on static analysis cannot effectively handle malicious code that employs advanced obfuscation tricks such as indirect jumps and self-modifications. Dynamic code analysis using emulation is not hindered by such obfuscations and can detect even extensively obfuscated shellcode. This kind of "actual" payload execution has proved quite effective in practice [22] and is being used in network-level and host-level systems for the zero-day detection of both server-side and client-side code injection attacks [9, 14, 15, 23, 38].

A limitation of the above techniques is that they are confined to the detection of a particular class of polymorphic shellcode that exhibits self-decrypting behavior. Although shellcode "packing" and encryption are commonly used for evading signature-based detectors, attackers can achieve the same or even higher level of evasiveness without the use of self-decrypting code, rendering above systems ineffective. Besides code encryption, polymorphism can instead be achieved by mutating the actual instructions of the shellcode before launching the attack—a technique known as *metamorphism* [32]. Metamorphism has been widely used by virus authors and thus can trivially be applied for shellcode mutation. Surprisingly, even *plain* shellcode, i.e., shellcode that does not change across different instances, is also not detected by existing payload execution methods. Technically, a plain shellcode is no different than any instance of metamorphic shellcode, since both do not carry a decryption routine nor exhibit any self-modifications or dynamic code generation. Consequently, an attack that uses a previously unknown static analysis-resistant plain shellcode will manage to evade existing detection systems.

In this paper, we present a comprehensive shellcode detection technique based on payload execution. In contrast to previous ap-

ACSAC '10 Dec. 6-10, 2010, Austin, Texas USA

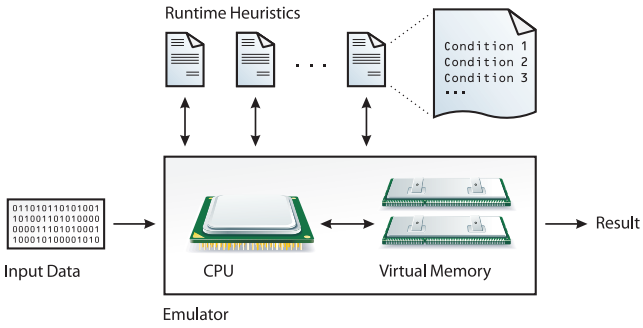

Figure 1: Overview of the proposed shellcode detection architecture.

proaches that use a single detection algorithm for a particular class of shellcode, our method relies on several runtime heuristics tailored to the identification of different shellcode types. We have designed four heuristics for the detection of plain and metamorphic shellcode targeting Windows systems. Polymorphic shellcode is in essence a self-decrypting version of a plain shellcode, and thus it is also effectively detected, since the concealed plain shellcode is revealed during execution. In fact, we also enable the detection of polymorphic shellcode that uses SEH-based GetPC code, which is currently not handled by existing polymorphic shellcode detectors. Furthermore, instead of solely using a CPU emulator, our approach couples the heuristics with an appropriate image of the complete address space of a real process, enabling the correct execution of shellcode that depends on certain kinds of host-level context.

We have implemented the above technique in Gene, a network-level detector that scans all client-initiated streams for code injection attacks against network services. Gene is based on passive network monitoring, which offers the benefits of easy large-scale deployment and protection of multiple hosts using a single sensor, while it allows us to test the effectiveness of our technique in real-world environments. Nevertheless, although Gene operates at the network level, its core inspection engine can analyze arbitrary data coming from any source. This allows our approach to be readily embedded in existing systems that employ emulation-based detection in other domains, e.g., for the detection of malicious websites [15] or in browser add-ons for the detection of drive-by download attacks [14].

Our evaluation with publicly available shellcode samples and shellcode construction toolkits, shows that Gene can effectively detect many different shellcode instances without prior knowledge about each particular implementation. At the same time, after extensive testing of the runtime heuristics using a large and diverse set of generated and real data, in addition to a five-month deployment in production networks, Gene has not generated any false positives.

2. ARCHITECTURE

The proposed shellcode detection system is built around a CPU emulator that executes valid instruction sequences found in the inspected input. An overview of our approach is illustrated in Fig. 1. Each input is mapped to an arbitrary location in the virtual address space of a supposed process, and a new execution begins from each and every byte of the input, since the position of the first instruction of the shellcode is unknown and can be easily obfuscated. The detection engine is based on multiple heuristics that match runtime patterns inherent in different types of shellcode. During execution, the system checks several conditions that should all be satisfied in order for a heuristic to match some shellcode. Moreover, new heuristics can easily be added due to the extensible nature of the system.

Abbreviation	Matching Shellcode Behavior
PEB	`kernel32.dll` base address resolution
BACKWD	`kernel32.dll` base address resolution
SEH	Memory scanning / SEH-based GetPC code
SYSCALL	Memory scanning

Table 1: Overview of the shellcode detection heuristics used in Gene.

Existing polymorphic shellcode detection methods focus on the identification of self-decrypting behavior, which can be simulated without any host-level information [23]. For example, accesses to addresses other than the memory area of the shellcode itself are ignored. However, shellcode is meant to be injected into a running process and it usually accesses certain parts of the process' address space, e.g., for retrieving and calling API functions. In contrast to previous approaches, the emulator used in our system is equipped with a fully blown virtual memory subsystem that handles all user-level memory accesses and enables the initialization of memory pages with arbitrary content. This allows us to populate the virtual address space of the supposed process with an image of the mapped pages of a process taken from a real system.

The purpose of this functionality is twofold: First, it enables the construction of heuristics that check for memory accesses to process-specific data structures. Although the heuristics presented in this paper target Windows shellcode, and thus the address space image used in conjunction with these heuristics is taken from a Windows process, some other heuristic can use a different memory image, e.g., taken from a Linux process. Second, this allows to some extent the correct execution of non-self-contained shellcode that may perform accesses to known memory locations for evasion purposes [10]. We discuss this issue further in Sec. 6.

3. RUNTIME HEURISTICS

Each heuristic used in Gene is composed of a sequence of conditions that should *all* be satisfied *in order* during the execution of malicious code. Table 1 gives an overview of the four heuristics presented in this section. The heuristics focus on the identification of the first actions of different shellcode types, according to their functionality, regardless of any self-decrypting behavior.

3.1 Resolving kernel32.dll

The typical end goal of the shellcode is to give the attacker full control of the victim system. This usually involves just a few simple operations, such as downloading and executing a malware binary on the compromised host. These operations require interaction with the OS through the system call interface, or in case of Microsoft Windows, through the user-level Windows API.

The Windows API is divided into several dynamic load libraries (DLLs). In order to call an API function, the shellcode must first find its absolute address in the address space of the process. This can be achieved in a reliable way by searching for the Relative Virtual Addresses (RVAs) of the function in the Export Directory Table (EDT) of the DLL. The absolute Virtual Memory Address (VMA) of the function can then be easily computed by adding the DLL's base address to the function's RVA. In fact, `kernel32.dll` provides the quite convenient functions `LoadLibrary`, which loads the specified DLL into the address space of the calling process and returns its base address, and `GetProcAddress`, which returns

```
1  xor eax, eax            ; eax = 0
2  mov eax, fs:[eax+0x30]  ; eax = PEB
3  mov eax, [eax+0x0C]     ; eax = PEB.LoaderData
4  mov esi, [eax+0x1C]     ; esi = InInitializationOrder
                                   ModuleList.Flink
5  lodsd                   ; eax = 2nd list entry
                                   (kernel32.dll)
6  mov eax, [eax+0x08]     ; eax = LDR_MODULE.BaseAddress
```

Figure 2: A typical example of code that resolves the base address of `kernel32.dll` through the PEB.

the address of an exported function from the specified DLL. After resolving these two functions, any other function in any DLL can be loaded and used directly. However, custom function searching using hashes is usually preferable in modern shellcode, since `GetProcAddress` takes as argument the actual name of the function to be resolved, which increases the shellcode size considerably.

No matter which method is used, a common fundamental operation in all above cases is that the shellcode has to first locate the base address of `kernel32.dll`. Since this is an inherent operation that must be performed by any Windows shellcode that needs to call a Windows API function, it is a perfect candidate for the development of a generic shellcode detection heuristic.

3.1.1 Process Environment Block

Probably the most reliable and widely used technique for determining the base address of `kernel32.dll` takes advantage of the Process Environment Block (PEB), a user-level structure that holds extensive process-specific information. Figure 2 shows a typical example of PEB-based code for resolving `kernel32.dll`. The shellcode first gets a pointer to the PEB (line 2) through the Thread Information Block (TIB), which is always accessible at a zero offset from the segment specified by the FS register. A pointer to the PEB exists 0x30 bytes into the TIB, as shown in Fig. 3. The absolute memory address of the TIB and the PEB varies among processes, and thus the only reliable way to get a handle to the PEB is through the FS register, and specifically, by reading the pointer located at address `FS:[0x30]`.

Condition P1. This fundamental constraint is the basis of our first detection heuristic **(PEB)**. If during the execution of some input the following condition is true **(P1)**: *(i) the linear address of `FS:[0x30]` is read, and (ii) the current or any previous instruction involved the FS register*, then this input may correspond to a shellcode that resolves `kernel32.dll` through the PEB.

The second predicate is necessary for two reasons. First, it is useful for excluding random instructions in benign inputs that happen to read from the linear address of `FS:[0x30]` without involving the FS register. For example, if `FS:[0x30]` corresponds to address `0x7FFDF030` (as shown in the example of Fig. 3), the following code will correctly not match the above condition:

```
mov ebx, 0x7FFD0000
mov eax, [ebx+0xF030]  ; eax = FS:[0x30]
```

On the other hand, the memory access to `FS:[0x30]` can be made through an instruction that does not use the FS register directly. For example, an attacker could take advantage of other segment registers and replace the first two lines in Fig. 2 with:

```
mov ax, fs             ; ax = fs
mov bx, es             ; preserve es
mov es, ax             ; es = fs
mov eax, es:[0x30]     ; load FS:[0x30] to eax
mov es, bx             ; restore es
```

The code loads the segment selector of the FS register to ES (`mov` between segment registers is not supported), reads the pointer to the PEB, and then restores the original value of the ES register.

The linear address of the TIB is also contained in the TIB itself at the location `FS:[0x18]`, as shown in Fig. 3. Thus, another way of reading the pointer to the PEB without using the FS register in the same instruction is the following:

```
xor eax,eax            ; eax = 0
xor eax,fs:[eax+0x18]  ; eax = TIB address
mov eax,[eax+0x30]     ; eax = PEB address
```

Note in the above example that other instructions besides `mov` can be used to indirectly read a memory address through the FS register (`xor` in this case). No matter how obfuscated the code is, the condition remains robust since it does not rely on the execution of particular instructions.

Although condition P1 is quite restrictive, the possibility of encountering a random read from `FS:[0x30]` during the execution of some benign input is not negligible. Thus, it is desirable to strengthen the heuristic with more operations exhibited by any PEB-based `kernel32.dll` resolution code.

Condition P2. Having a pointer to the PEB, the next step of the shellcode is to obtain a pointer to the `PEB_LDR_DATA` structure that holds the list of loaded modules (line 3 in Fig. 2). Such a pointer exists 0xC bytes into the PEB, in the `LoaderData` field. Since this is the only available reference to that data structure, the shellcode unavoidably has to read the `PEB.LoaderData` pointer. We can use this constraint as a second condition for the PEB heuristic **(P2)**: *the linear address of `PEB.LoaderData` is read*.

Condition P3. Moving on, the shellcode has to walk through the loaded modules list and locate the second entry (`kernel32.dll`). A pointer to the first entry of the list exists in the `InInitializationOrderModuleList.Flink` field located 0x1C bytes into the `PEB_LDR_DATA` structure. The read operation from this memory location (line 4 in Fig. 2) allows for strengthening further the detection heuristic with a third condition.

Although this is the most well known [5,26,27], and widely used technique for all Windows versions up to Windows Vista, it does not work "as-is" for Windows 7. In that version, `kernel32.dll` is found in the third instead of the second position in the modules list [7]. A more generic and robust technique is to walk through the list and check the actual name of each module until `kernel32.dll` is found [7, 29]. In fact, the `PEB_LDR_DATA` structure contains two more lists of the loaded modules that differ in the order of the DLLs. All three lists are implemented as doubly linked lists, and their corresponding `LIST_ENTRY` records contain two pointers to the first (`Flink`) and last (`Blink`) entry in the list.

Based on the above, and given that (i) `kernel32.dll` can be resolved through any of the three lists, and (ii) list traversing can be made in both directions, the third condition of the heuristic can be specified as follows **(P3)**: *the linear address of any of the `Flink` or `Blink` pointers in the `InLoadOrderModuleList`, `InMemoryOrderModuleList`, or `InInitializationOrderModuleList` records of the `PEB_LDR_DATA` structure is read*.

3.1.2 Backwards Searching

An alternative technique for locating `kernel32.dll` is to find a pointer that points somewhere into the memory area where the `kernel32.dll` has been loaded, and then search backwards until the beginning of the DLL is located [27]. A reliable way to obtain a pointer into the address space of `kernel32.dll` is to take advantage of the Structured Exception Handling (SEH) mechanism of Windows [21], which provides a unified way of handling hardware and software exceptions. When an exception occurs, the exception dispatcher walks through a list of exception handlers for

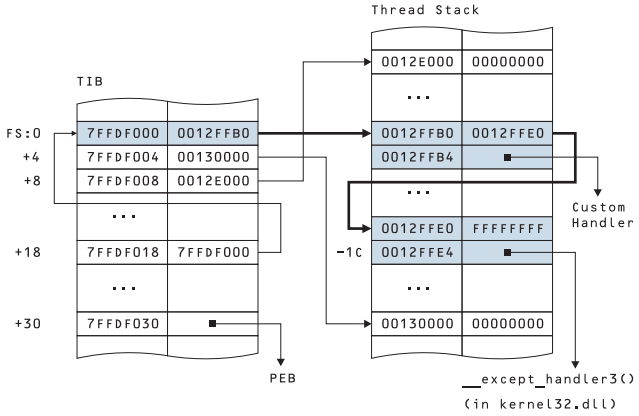

Figure 3: A snapshot of the TIB and the stack memory areas of a typical Windows process. The SEH chain consisting of two nodes is highlighted.

the current thread and gives each handler the opportunity to handle the exception or pass it on to the next handler. The list is stored on the stack of each thread, and each node is a SEH frame that consists of two pointers to the next frame and the actual handler routine. Figure 3 shows a typical snapshot of the TIB and the stack memory areas of a process with two SEH handlers. A pointer to the current SEH frame exists in the first field of the Thread Information Block and is always accessible through `FS:[0]`.

At the end of the SEH chain (bottom of the stack) there is a default exception handler that is registered by the system for every thread. The `Handler` pointer of this SEH record points to a routine that is located in `kernel32.dll`, as shown in Fig. 3. Thus, the shellcode can start from `FS:[0]` and walk the SEH chain until reaching the last SEH frame, and from there get a pointer into `kernel32.dll` by reading its `Handler` field.

Another technique to reach the last SEH frame, known as "TOPSTACK" [27], uses the stack of the exploited thread. The default exception handler is registered by the system during thread creation, making its relative location from the bottom of the stack fairly stable. Although the absolute address of the stack may vary, a pointer to the bottom of the stack is always found in the second field of the TIB at `FS:[0x4]`. The `Handler` pointer of the default SEH handler can then be found 0x1C bytes into the stack, as shown in Fig. 3. In fact, the TIB contains a second pointer to the top of the stack at `FS:[0x8]`.

Condition B1. Based on the same approach as in the previous section, the first condition for the detection heuristic **(BACKWD)** that matches the "backwards searching" method is the following **(B1)**: *(i) any of the linear address between `FS:[0]`–`FS:[0x8]` is read, and (ii) the current or any previous instruction involved the `FS` register*. The rationale is that a shellcode that uses the backwards searching technique should unavoidably read either i) the memory location at `FS:[0]` for walking the SEH chain, or ii) one of the locations at `FS:[0x4]` and `FS:[0x8]` for accessing the stack directly.

Condition B2. In any case, the code will reach the default exception record on the stack and read its `Handler` pointer. Since this is a mandatory operation for landing into `kernel32.dll`, we can use this dependency as our second condition **(B2)**: *the linear address of the `Handler` field of the default SEH handler is read*.

Condition B3. Finally, during the backwards searching phase, the shellcode will inevitably perform several memory accesses to the address space of `kernel32.dll` in order to check whether each 64KB-aligned address corresponds to the base address of the DLL. In our experiments with typical code injection attacks in Windows XP, the shellcode performed at least four memory reads in `kernel32.dll`. Thus, after the first two conditions have been met, we expect to encounter **(B3)**: *at least one memory read form the address space of `kernel32.dll`*.

3.2 Process Memory Scanning

Some memory corruption vulnerabilities allow only a limited space for the injected code—usually not enough for a fully functional shellcode. In most such exploits though the attacker can inject a second, much larger payload which however will land at a random, non-deterministic location, e.g., in a buffer allocated in the heap. The first-stage shellcode can then sweep the address space of the process and search for the second-stage shellcode (also known as the "egg"), which can be identified by a long-enough characteristic byte sequence. This type of first-stage payload is known as "egg-hunt" shellcode [28].

Blindly searching the memory of a process in a reliable way requires some method of determining whether a given memory page is mapped into the address space of the process. In the rest of this section, we describe two known memory scanning techniques and the corresponding detection heuristics that can capture these behaviors, and thus, identify the execution of egg-hunt shellcode.

3.2.1 SEH

The first memory scanning technique takes advantage of the structured exception handling mechanism and relies on installing a custom exception handler that is invoked in case of a memory access violation.

Condition S1. As discussed in Sec. 3.1.2, the list of SEH frames is stored on the stack, and the current SEH frame is always accessible through `FS:[0]`. The first-stage shellcode can register a custom exception handler that has priority over all previous handlers in two ways: create a new SEH frame and adjust the current SEH frame pointer of the TIB to point to it [28], or directly modify the `Handler` pointer of the current SEH frame to point to the attacker's handler routine. In the first case, the shellcode must update the SEH list head pointer at `FS:[0]`, while in the second case, it has to access the current SEH frame in order to modify its `Handler` field, which is only possible by reading the pointer at `FS:[0]`. Thus, the first condition of the SEH-based memory scanning detection heuristic **(SEH)** is **(S1)**: *(i) the linear address of `FS:[0]` is read or written, and (ii) the current or any previous instruction involved the `FS` register*.

Condition S2. Another mandatory operation that will be encountered during execution is that the `Handler` field of the custom SEH frame (irrespectively if its a new frame or an existing one) should be modified to point to the custom exception handler routine. This operation is reflected by the second condition **(S2)**: *the linear address of the `Handler` field in the custom SEH frame is or has been written*. Note that in case of a newly created SEH frame, the `Handler` pointer can be written before or after `FS:[0]` is modified.

Condition S3. Although the above conditions are quite constraining, we can apply a third condition by exploiting the fact that upon the registration of the custom SEH handler, the linked list of SEH frames should be valid. In the risk of stack corruption, the exception dispatcher routine performs thorough checks on the integrity of the SEH chain, e.g., ensuring that each SEH frame is dword-

```
1  push  edx       ; preserve edx across system call
2  push  0x8
3  pop   eax       ; eax = NtAddAtom
4  int   0x2e      ; system call
5  cmp   al, 0x05  ; check for STATUS_ACCESS_VIOLATION
6  pop   edx       ; restore edx
```

Figure 4: A typical system call invocation for checking if the supplied address is valid.

aligned within the stack and is located higher than the previous SEH frame [21]. Thus, the third condition requires that **(S3)**: *starting from* `FS:[0]`*, all SEH frames should reside on the stack, and the* `Handler` *field of the last frame should be set to 0xFFFFFFFF*. In essence, the above condition validates that the custom handler registration has been performed correctly.

3.2.2 System Call

The extensive abuse of the SEH mechanism in various memory corruption vulnerabilities led to the introduction of SafeSEH, a linker option that produces a table with all the legitimate exception handlers of the image. In case the exploitation of some SafeSEH-protected vulnerable application requires the use of egg-hunt shellcode, an alternative but less reliable method for safely scanning the process address space is to check whether a page is mapped—before actually accessing it—using a system call [27, 28]. As already discussed, although the use of system calls in Windows shellcode is not common, since they are prone to changes between OS versions and do not provide crucial functionality such as network access, they can prove useful for determining if a memory address is accessible.

Some Windows system calls accept as an argument a pointer to an input parameter. If the supplied pointer is invalid, the system call returns with a return value of `STATUS_ACCESS_VIOLATION`. Thus, the egg-hunt shellcode can check the return value of the system call, and proceed accordingly by searching for the egg or moving on to the next address [28]. In Windows, a system call is initiated by generating a software interrupt through the `int 0x2e` instruction.

Figure 4 shows a typical code that checks the address stored in `edx` using the `NtAddAtom` system call. In Windows, a system call is initiated by generating a software interrupt through the `int 0x2e` instruction (line 4). The actual system call that is going to be executed is specified by the value stored in the `eax` register (line 3). Upon return from the system call, the code checks if the return value equals the code for `STATUS_ACCESS_VIOLATION`. The actual value of this code is 0xC0000005, but checking only the lower byte is enough in return for more compact code (line 5).

Condition C1. System call execution has several constraints that can be used for deriving a detection heuristic for this kind of egg-hunt shellcode. First, the immediate operand of the `int` instruction should be set to 0x2E. Looking just for the `int 0x2e` instruction is clearly not enough since any two-byte instruction will be encountered roughly once every 64KB of arbitrary binary input. However, when encountering an `int 0x2e` instruction that corresponds to an actual system call execution, the `ebx` register should also have been previously set to the proper system call number.

The publicly available egg-hunt shellcode implementations we found (see Sec. 5.1) use one of the following system calls: `NtAccessCheckAndAuditAlarm` (0x2), `NtAddAtom` (0x8), and `NtDisplayString` (0x39 in Windows 2000, 0x43 in XP, 0x46 in 2003 Server, and 0x7F in Vista). The variability of the system call number for `NtDisplayString` across the different Windows versions is indicative of the complexity introduced in an exploit by the direct use of system calls. Based on the above, a necessary condition during the execution of a system call in egg-hunt shellcode is **(C1)**: *the execution of an* `int 0x2e` *instruction with the* `eax` *register set to one of the following values: 0x2, 0x8, 0x39, 0x43, 0x46, 0x7F*.

Condition C2. As shown in Sec. 5.2.2, condition C1 alone can happen to hold true during the execution of random code, although rarely. However, the heuristic can be strengthened based on the following observation. The egg-hunt shellcode will have to scan a large part of the address space until it finds the egg. Even when assuming that the egg can be located only at the beginning of a page [37], the shellcode will have to search hundreds or thousands of addresses, e.g., by repeatedly calling the code in Fig. 4 in a loop. Hence, condition C1 will hold several times. The detection heuristic **(SYSCALL)** can then be defined as a meta-condition **(C{N})**: *C1 holds true N times*. As shown in Sec. 5.2.2, a value of N = 2 does not produce any false positives.

In case other system calls can be used for validating an arbitrary address, they can easily be included in the above condition. Starting from Windows XP, system calls can also be made using the more efficient `sysenter` instruction if it is supported by the system's processor. The above heuristic can easily be extended to also support this type of system call invocation.

3.3 SEH-based GetPC Code

Before decrypting itself, polymorphic shellcode needs to first find the absolute address at which it resides in the address space of the vulnerable process. The most widely used types of GetPC code for this purpose rely on some instruction from the `call` or `fstenv` instruction groups [23]. These instructions push on the stack the address of the following instruction, which can then be used to calculate the absolute address of the encrypted code. However, this type of GetPC code cannot be used in purely alphanumeric shellcode [19], because the opcodes of the required instructions fall outside the range of allowed ASCII bytes. In such cases, the attacker can follow a different approach and take advantage of the SEH mechanism to get a handle to the absolute memory address of the injected shellcode [30].

When an exception occurs, the system generates an exception record that contains the necessary information for handling the exception, including a snapshot of the execution state of the thread, which contains the value of the program counter at the time the exception was triggered. This information is stored on the stack, so the shellcode can register a custom exception handler, trigger an exception, and then extract the absolute memory address of the faulting instruction. By writing the handler routine on the heap, this technique can work even in Windows XP SP3, bypassing any SEH protection mechanisms [30].

In essence, the SEH-based memory scanning detection heuristic described in Sec. 3.2.1 does not identify the scanning behavior per se, but the proper registration of a custom exception handler. Although this is an inherent operation of any SEH-based egg-hunt shellcode, any shellcode that installs a custom exception handler can be detected, including polymorphic shellcode that uses SEH-based GetPC code.

4. IMPLEMENTATION

We have implemented the proposed detection method in Gene, a network-level attack detector that uses a custom IA-32 emulator to identify the presence of shellcode in network streams. Gene scans the client-initiated part of each TCP connection using the runtime heuristics presented in this work. For evaluation purposes, a fifth

GetPC-based self-decrypting shellcode similar to the one used in existing detectors [9, 23, 38] can be enabled at will. Since the exact location of the shellcode in the input data is not known in advance, the emulator repeats the execution multiple times, starting from each and every position of the stream. In certain cases, however, the execution of some code paths can be skipped to optimize runtime performance [24].

The heuristics used in Gene are mostly based on memory accesses to certain locations in the address space of a vulnerable Windows process. To emulate correctly the execution of these accesses, the virtual memory of the emulator is initialized with an image of the complete address space of a typical Windows XP process taken from a real system. The image consists of 971 pages (4KB each), including the stack, heap, PEB/TIB, and loaded modules. All four heuristics use the same memory image and thus can be evaluated in parallel during execution.

Among other initializations before the beginning of a new execution [23], the segment register `FS` is set to the segment selector corresponding to the base address of the Thread Information Block, the stack pointer is set accordingly, while any changes to the original process image from the previous execution are reverted.

The runtime evaluation of the heuristics requires keeping some state about the occurrence of instructions with an operand that involved the `FS` register, as well as about read and write accesses to the memory locations specified in the heuristics. Regarding the SEH-based memory scanning heuristic (Sec. 3.2.1), although SEH chain validation is more complex compared to other instrumentation operations, it is triggered only if conditions S1 and S2 are true, which in practice happens very rarely.

When an `int 0x2e` instruction is executed, the `eax` register is checked for a value corresponding to one of the system calls that can be used for memory scanning, as described in Sec. 3.2.2. Although the actual functionality of the system call is not emulated, the proper return value is stored in the `eax` register depending on the validity of the supplied memory address. In case of an egg-hunt shellcode, this behavior allows the scanning loop to continue normally, resulting to several system call invocations.

5. EXPERIMENTAL EVALUATION

5.1 Detection Effectiveness

We began our evaluation with the shellcodes contained in the Metasploit Framework [2]. For Windows targets, Metasploit includes six basic payloads for downloading and executing a file, spawning a shell, adding a user account, and so on, as well as nine "stagers." In contrast to an egg-hunt shellcode, which searches for a second payload that has already been injected into the vulnerable process along with the egg-hunt shellcode, a stager establishes a channel between the attacking and the victim host for uploading other second-stage payloads. We generated plain (i.e., non-encrypted) instances of the above 15 shellcodes, as well as another 15 polymorphic instances of the same shellcodes using the ShikataGaNai encoder. As shown in Fig. 5, both Gene and the GetPC-based heuristic detected the polymorphic versions of the shellcodes. However, the original (plain) versions do not exhibit any self-decrypting behavior and are thus detected only by Gene. For both plain and polymorphic versions, Gene identified the shellcode using the PEB heuristic. The use of the PEB-based method for locating `kernel32.dll` is probably preferred in Metasploit due to its reliability.

We continued our evaluation with 22 samples downloaded from the shellcode repository of the Nepenthes Project [6]. Two of the samples had a broken decryptor and could not be executed properly. By manually unpacking the two payloads and scanning them with Gene, in both cases the shellcode was identified by the PEB heuristic. From the rest 20 shellcodes, 16 were identified by the PEB heuristic, and one, named "Saalfeld," by the SEH heuristic. The Saalfeld shellcode is of particular interest due to the use of a custom SEH handler although it is not an egg-hunt shellcode. The SEH handler is registered for safely searching the address space of the vulnerable process starting from address 0x77E00000, with the aim to reliably detect the base address of `kernel32.dll`. The SEH heuristic identifies the proper registration of a custom SEH handler, so the shellcode was successfully identified.

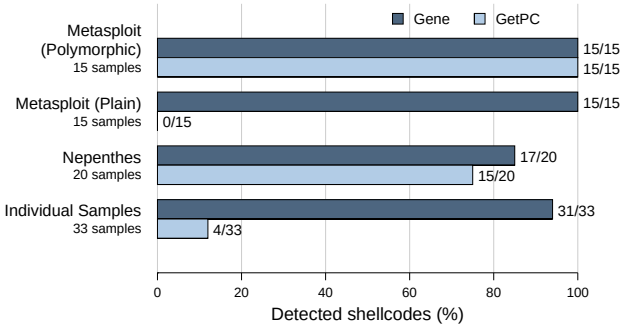

Figure 5: Number of shellcodes detected by Gene and the existing GetPC-based heuristic [9, 23, 38] for different shellcode sets. From a total of 83 different shellcode implementations, Gene detected 78 samples (94%), compared to 34 (41%) for the GetPC heuristic.

The remaining three shellcodes were missed due to the use of hard-coded addresses, e.g., the linear address of `kernel32.dll`, instead of reliable base address resolution. It would be trivial to implement another detection heuristic similar to the PEB heuristic based on commonly used hard-coded addresses in place of addressing based on the `FS` register to detect this kind of shellcode. However, these samples correspond to quite old attacks and this style naively implemented kind of shellcode is now encountered rarely. From the 20 shellcodes, 15 are self-decrypting and are thus detected by the GetPC-based heuristic.

Besides a few proof-of-concept implementations [5, 27] which are identified correctly by Gene, we were not able to find any other shellcode samples that locate `kernel32.dll` using backwards searching, probably due to the simplicity of the alternative PEB-based technique. In addition to the Saalfeld shellcode, the SEH heuristic detected a proof-of-concept SEH-based egg-hunt implementation [28], as well as the "omelet" shellcode [36], an egg-hunt variation that locates and recombines multiple smaller eggs into the whole original payload. The SEH heuristic was also effective in detecting polymorphic shellcode that uses SEH-based GetPC code [30], which is currently missed by existing payload execution systems. The SYSCALL heuristic was tested with three different egg-hunt shellcode implementations [27,28,37], which were identified correctly. In addition to these eight shellcode implementations, we gathered more Windows shellcode samples from public repositories [1, 3, 4], totaling 33 different samples. As shown in Fig. 5, the GetPC-based heuristic detected only four of the shellcodes that use simple XOR encryption, while Gene detected all but two of the samples, again due to the use of hard-coded addresses.

Finally, as an extra verification experiment, we tested Gene with a large dataset of real polymorphic attacks captured in production networks by Nemu [22]. Without using any self-decryption heuristic, this data set allows us to test the effectiveness of Gene in iden-

tifying the actual plain shellcode after the decryption process has completed. Gene analyzed more than 1.2 million attacks, which after the decryption process resulted to 98,602 unique payloads, and in all cases it identified the decrypted plain shellcode correctly. Not surprisingly, all shellcodes were identified by the PEB heuristic.

5.2 Heuristic Robustness

5.2.1 False Positives Evaluation

We tested the robustness of the heuristics against false positives using a large and diverse set of benign inputs. For our first experiment, we captured the internal and external traffic in two research and educational networks and kept the client-initiated stream of each TCP flow, since currently Gene detects only attacks against network services. Collectively, the data set consists of 15.5 million streams, totaling more than 48GB of data. Depending on its size, a stream can have from a few hundreds to many thousands of valid instruction sequences which are all analyzed independently by Gene. Thus, we consider as a false positive any benign input with at least one instruction sequence that matches one of the heuristics. When scanning the 15.5 million streams of this data set with Gene, none of the inputs matched any of the heuristics, resulting to zero false positives.

Seeking more evidence for the resilience of the heuristics against false positives, we continued the experiments with a much larger set of artificially generated benign data. The purpose of this experiment is to ensure that the random IA-32 machine code that is derived by interpreting arbitrary data as code does not match any of the heuristics. For this purpose, we used a script that continuously generates inputs of random binary and ASCII data that are subsequently scanned by Gene. The script generated 20 million 32KB-inputs of each type, totaling more than 1.3TB of data. The rationale behind using inputs consisting of random ASCII characters, in addition to random binary data, is to approximate the random code found in network streams that use text-based protocols. Similarly to the previous experiment, the false positive rate was again kept at zero.

5.2.2 Heuristic Analysis

We repeated the experiments of the previous section with the aim to explore in depth the behavior of the heuristics when operating on benign data. This time we measured the number of inputs with at least one instruction sequence that matched the first, the first two, or all three conditions of a heuristic.

Figure 6(a) shows the percentage of network streams that matched a given number of conditions. Out of 15.5 million inputs, only 82 (0.0005%) had an instruction sequence with a memory access to `FS:[0x30]` through the `FS` register—satisfying the first condition of the PEB heuristic. There were no streams that matched both the first and the second or all three conditions, which is a promising indication for the robustness of the PEB heuristic since all three conditions must be true for flagging an input as shellcode. The SYSCALL heuristic had a similar behavior, with just 51 of the inputs (0.0003%) exhibiting a single system call invocation, while there were no streams with two or more system calls.

A much larger number of streams matched the first condition of the BACKWD and SEH heuristics (8,620 and 41,063 streams, respectively). In both heuristics, the first condition includes a memory access to `FS:[0]`, which seems to appear more frequently in random code compared to accesses at `FS:[0x30]`. A possible explanation for this is that the effective address computation in the memory operand of some instruction can result to zero with a higher probability compared to other values. For example, when a `mov ebx,fs:[eax]` instruction is executed, it is more likely that `eax` will have been zeroed out, e.g., due to a previous two-byte long `xor eax,eax` instruction, instead of being set to 0x30. However, the percentage of inputs that matched both the first and the second condition is very low (0.0003% and 0.0004%, respectively), and no inputs matched all three conditions.

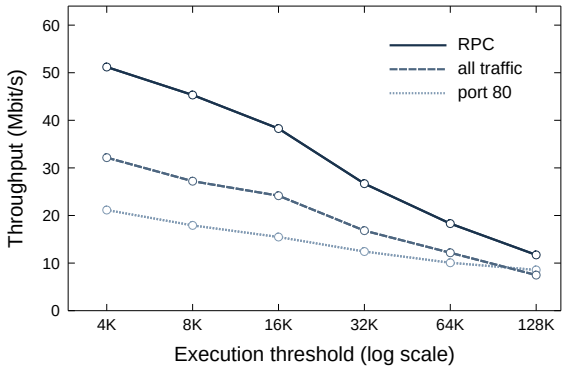

Figure 7: The raw processing throughput of Gene for different execution thresholds.

As shown in Fig. 6(b), the overall behavior when operating on random binary data is comparable to that for network streams, with no inputs fully matching any of the heuristics. However, for ASCII data (Fig. 6(c)), although the first condition in the PEB, BACKWD, and SEH heuristics matched in roughly 0.03% of the inputs, there were no inputs matching any of the subsequent conditions. The opcode for the `int` instruction falls outside the ASCII range, so no input matched not even the first condition of the SYSCALL heuristic. Overall, all heuristics seem to perform even better when operating on ASCII data.

5.3 Runtime Performance

We evaluated the processing throughput of Gene using the real network traffic traces presented in Sec. 5.2.1. Gene was running on a system with a Xeon 1.86GHz processor and 2GB of RAM. Figure 7 shows the raw processing throughput of Gene for different execution thresholds. The throughput is mainly affected by the number of CPU cycles spent on each input. As the execution threshold increases, the achieved throughput decreases because more emulated instructions are executed per stream. A threshold in the order of 8–16K instructions is sufficient for the detection of plain as well as the most advanced polymorphic shellcodes [24]. For port 80 traffic, the random code due to ASCII data tends to form long instruction sequences that result to degraded performance compared to binary data.

The overall runtime throughput is slightly lower compared to existing emulation-based detectors [23, 24] due to the overhead added by the virtual memory subsystem, as well as because Gene does not use the zero-delimited chunk optimization used in these systems [23]. Previous approaches skip the execution of zero-byte delimited regions smaller than 50 bytes, with the rationale that most memory corruption vulnerabilities cannot be exploited if the attack vector contains null bytes. However, the detection heuristics of Gene can identify shellcode in other attack vectors that may contain null bytes, such as document files. Furthermore, our approach can be applied in other domains [14, 15], for example for the detection of client-side attacks, in which the shellcode is usually encrypted at a higher level using some script language, and thus can be fully functional even if it contains null bytes.

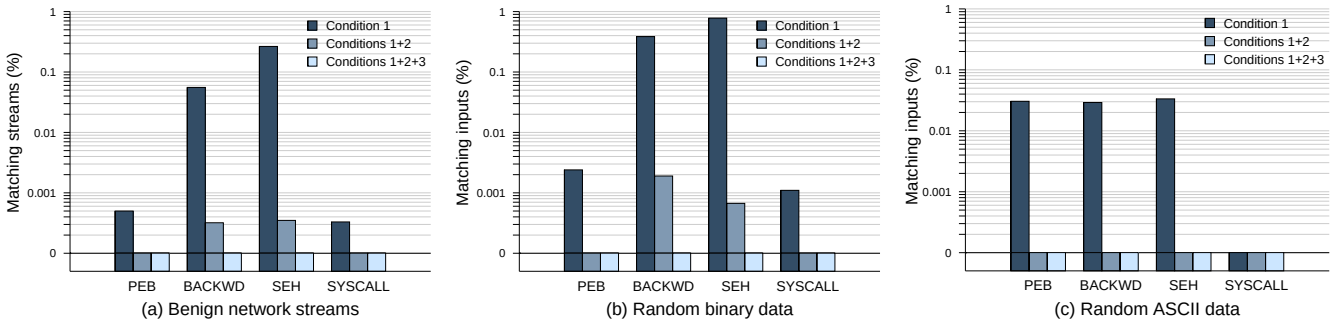

Figure 6: False positives evaluation with (a) 15.5 million real network streams (48GB total data size), (b) 20 million randomly generated binary inputs (650GB), and (c) 20 million randomly generated ASCII inputs (650GB). For all heuristics, none of the inputs matched all three conditions, resulting to zero false positives.

In practice, Gene can monitor high speed links when scanning for server-side attacks because client-initiated traffic (requests) is usually a fraction of the server-initiated traffic (responses). In our preliminary deployments in production networks, Gene can scan traffic of up to 100 Mbit/s without dropping packets. Furthermore, Gene currently scans the whole input blindly, without any knowledge about the actual network protocol used. Augmenting the inspection engine with protocol parsing would significantly improve the scanning throughput by inspecting each protocol field separately.

5.4 Real-world Deployment

We have deployed Gene in two University networks, where it has been operational since 25 November 2009. In these two deployments, Gene scans the traffic between the internal network and the Internet, as well as the traffic between selected internal subnets. As of 17 April 2010, Gene has detected 116,513 code injection attacks against internal and external hosts in these two networks. Although we cannot know how many of the attacks actually infected the targeted host, since many systems might had been previously patched, in all cases the attacker was able to connect and send the malicious input to the potentially vulnerable service. Almost one third of the attacks were launched from internal PCs, probably already infected by malware. About 86% of the attacks targeted port 445, while there were also attacks against ports 80, 135, 139, and 2967.

In both deployments, Gene uses the four new heuristics presented in this paper, as well as the GetPC heuristic used in existing polymorphic shellcode detectors, allowing us to compare the detection coverage of both approaches. The PEB heuristic matched in all of the attacks, supporting the fact that this is the most widely used technique for resolving `kernel32.dll`. However, the GetPC heuristic was triggered only by 85,144 attacks, i.e., 31,369 attacks (27%) did not use any form of self-decrypting shellcode. This means that the ability of Gene to detect plain shellcode increased the detection coverage for server-side code injection attacks by 37% compared to existing polymorphic shellcode detection approaches. By statically analyzing the identified machine code [22] we confirmed that in all cases it corresponds to actual shellcode, and so far we have not encountered any false positives.

6. DISCUSSION

The runtime heuristics presented in this paper allows Gene to detect a broad range of different shellcode classes. Of course, we cannot exclude the possibility that there are other kinds of Windows shellcode, or alternative techniques to those on which the heuristics are based, that may have missed our attention or have not been publicly released yet. Nevertheless, the architecture of Gene allows the parallel evaluation of multiple heuristics, and thus the detection engine can be easily extended with more heuristics for other shellcode types. For example, for our experimental evaluation, we have already implemented a fifth heuristic based on the widely used GetPC code technique used in existing polymorphic shellcode detectors [23, 24, 38]. In our future work, we plan to implement heuristics for the detection of the code required in a swarm attack [13], Linux-specific plain shellcode, Windows shellcode that uses hard-coded addresses, and so on.

A well known evasion technique against dynamic code analysis systems is the use of very long loops that force the detector to spend countless cycles until reaching the execution threshold, before any signs of malicious behavior are shown [32]. Gene uses infinite loop squashing [23] to reduce the number of inputs that reach the execution threshold. As stated in the literature [23, 24], the percentage of inputs with an instruction sequence that reaches the execution threshold ranges between 3–6%, which we also verified during the experiments of this paper. Since this is a small fraction of all inspected inputs, the endless loops in these sequences can potentially be analyzed further at a second stage using other techniques such as static analysis or symbolic execution [25].

Another inherent limitation of emulation-based shellcode detection is the lack of an accurate view of the system's state at the time the injected code would run on the victim system. This information includes the values of the CPU registers, as well as the complete address space of the particular exploited process [10, 23]. Although register values can sometimes be inferred [24], and Gene augments the emulator with the complete address space of a typical Windows process, which includes the most common system DLLs used by Windows shellcode, the shellcode may perform memory accesses to application-specific DLLs that are not known in advance, and thus cannot be followed by the emulator [16]. Fortunately, when protecting specific services, exact memory images of each service can be used in place of the generic process image. However, as already discussed, since the linear addresses of DLLs change quite often across different systems, and due to the increasing adoption of address space layout randomization and DLL rebasing, the use of absolute addressing results to less reliable shellcode. On the other hand, when the emulator runs within the context of a protected application, as for example in the browser-embedded detector proposed by Egele et al. [14], the emulator can have full access to the complete address space of the process.

Some of the operations matched by the heuristics, such as the registration of a custom exception handler, might also be found in legitimate executables. However, Gene is tailored for scanning inputs that otherwise should not contain executable IA-32 code. In case of file uploads, Gene can easily be extended to identify and extract executable files by looking for executables' headers in the inspected traffic, and then pass them on to a virus scanner.

7. RELATED WORK

Having realised the limitations of signature-based approaches in the face of polymorphic code and targeted attacks, several research efforts turned to static binary code analysis for identifying the presence of shellcode in network streams. One of the first such approaches by Toth and Kruegel uses code disassembly on network streams to identify the NOP-sled that sometimes precedes the shellcode [33]. Focusing on the shellcode itself, Anderson et al. [8] propose to scan each input for multiple occurrences of instruction sequences that end with an `int 0x80` instruction for the identification of Linux shellcode, with the rationale that the shellcode will have to execute several system calls. Other detection methods that use static code analysis aim to detect previously unknown polymorphic shellcode based on the identification of structural similarities among different worm instances [17], control and data flow analysis [12, 34, 35], or neural networks [20].

However, methods based on static analysis can be easily evaded by malicious code that uses obfuscation methods such as indirect jumps and self-modifications [23], which are widely used by current malware packers and polymorphic shellcode engines. In contrast, emulation-based detection can correctly handle even extensively obfuscated malicious code [23]. Polychronakis et al. propose the use of code emulation for the detection of self decrypting shellcode at the network level [23, 24]. The detection algorithm is based on the identification of the GetPC code and the self-references that take place during the execution of the shellcode. Zhang et al. propose to combine network-level emulation with static and data flow analysis for improving the runtime performance of the GetPC heuristic [38].

`Libemu` [9] is an open-source x86 emulation library tailored to shellcode analysis and detection. Shellcode execution is identified using the GetPC heuristic. `Libemu` can also emulate the execution of Windows API calls by creating a minimalistic process environment that allows the user to install custom hooks to API functions. Although the actual execution of API functions can be used as an indication for the execution of shellcode, these actions will be observed only after `kernel32.dll` has been resolved and the required API functions have been located through the EDT or IAT. Compared to the `kernel32.dll` resolution heuristics presented in Section 3.1, this technique would require the execution of a much larger number of instructions until the first API function is called, and also the emulation of the actual functionality of each API call thereafter. This means that the execution threshold of the detector should be set much higher, resulting to degraded runtime performance. For applications in which the emulator can spend more cycles on each input, both heuristics can coexist and operate in parallel, e.g., along with all other heuristics used in Gene, offering even better detection accuracy.

Besides the detection of code injection attacks against network services [22], emulation-based shellcode detection using the GetPC heuristic has been used for the detection of drive-by download attacks and malicious web sites. Egele et al. [14] propose a technique that uses a browser-embedded CPU emulator to identify javascript string buffers that contain shellcode. Wepawet [15] is a service for web-based malware detection that scans and identifies malicious web pages based on various indications, including the presence of shellcode. The CPU emulator in both projects is based on `libemu`.

Shellcode analysis systems help analysts study and understand the structure and functionality of a shellcode sample. Ma et al. [18] used code emulation to extract the actual runtime instruction sequence of shellcode samples captured in the wild. Spector [11] uses symbolic execution to extract the sequence of library calls made by the shellcode, along with their arguments, and at the end of the execution generates a low-level execution trace. Yataglass [25] improves the analysis capabilities of Spector by handling shellcode that uses memory-scanning attacks.

8. CONCLUSION

The increasing professionalism of cyber criminals and the vast number of malware variants and malicious websites make the need for effective code injection attack detection a critical challenge. To this end, shellcode detection using payload execution offers important advantages, including generic detection without exploit or vulnerability-specific signatures, practically zero false positives, while it is effective against targeted attacks.

In this paper we present a comprehensive shellcode detection method based on code emulation. Our approach expands the range of malicious code types that can be detected by enabling the parallel evaluation of multiple runtime heuristics that match inherent low-level operations during the execution of different shellcode types. The runtime heuristics presented in this work enable the effective detection of plain and metamorphic shellcode, both of which are not identified by existing shellcode detectors. This is achieved regardless of the use of self-modifying code or dynamic code generation, on which existing emulation-based polymorphic shellcode detectors are exclusively based.

Our experimental evaluation shows that the proposed approach can effectively detect a broad range of diverse shellcode types and implementations, increasing significantly the detection coverage compared to existing emulation-based detectors, while extensive testing with a large set of benign data did not produce any false positives. Gene, our prototype implementation of the proposed technique for the detection of server-side code injection attacks detected 116,513 attacks against production systems in a period of almost five months without false positives.

Although Gene currently operates at the network level, the proposed detection heuristics can be readily implemented in emulation-based systems in other domains, including host-level or application-specific detectors. As part of our future work, we plan to implement more heuristics to cover the detection of less widely used shellcode types, such as shellcode that uses hard-coded addresses, and explore the design of a description language that would expedite the development of new heuristics.

Acknowledgments

We would like to thank Periklis Akritidis and Angelos Keromytis for their valuable feedback on earlier versions of this paper. This work was supported in part by the Marie Curie FP7-PEOPLE-2009-IOF project MALCODE and the FP7 project SysSec, funded by the European Commission under Grant Agreements No. 254116 and No. 257007, and by the project i-Code, funded by the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of the European Commission—Directorate-General for Home Affairs under Grant Agreement No. JLS/2009/CIPS/AG/C2-050. This publication reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained herein. Michalis Polychronakis is also with FORTH-ICS. Evangelos Markatos is also with the University of Crete.

9. REFERENCES

[1] Goodfellas security research team. http://goodfellas.shellcode.com.ar/.

[2] The metasploit project. http://www.metasploit.com/.

[3] milw0rm. http://milw0rm.com/shellcode/win32/.

[4] Packet storm. http://www.packetstormsecurity.org/.

[5] Win32 assembly components, Dec. 2002. http://lsd-pl.net.

[6] Common shellcode naming initiative, 2009. http://nepenthes.carnivore.it/csni.

[7] Retrieving kernel32's base address, June 2009. http://www.harmonysecurity.com/blog/2009/06/retrieving-kernel32s-base-address.html.

[8] S. Andersson, A. Clark, and G. Mohay. Network-based buffer overflow detection by exploit code analysis. In *Proceedings of the Asia Pacific Information Technology Security Conference (AusCERT)*, 2004.

[9] P. Baecher and M. Koetter. libemu, 2009. http://libemu.carnivore.it/.

[10] P. Bania. Evading network-level emulation, 2009. http://piotrbania.com/all/articles/pbania-evading-nemu2009.pdf.

[11] K. Borders, A. Prakash, and M. Zielinski. Spector: Automatically analyzing shell code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[12] R. Chinchani and E. V. D. Berg. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[13] S. P. Chung and A. K. Mok. Swarm attacks against network-level emulation/analysis. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2008.

[14] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[15] S. Ford, M. Cova, C. Kruegel, and G. Vigna. Wepawet, 2009. http://wepawet.cs.ucsb.edu/.

[16] I)ruid. Context-keyed payload encoding. *Uninformed*, 9, Oct. 2007.

[17] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2005.

[18] J. Ma, J. Dunagan, H. J. Wang, S. Savage, and G. M. Voelker. Finding diversity in remote code injection exploits. In *Proceedings of the 6th Internet Measurement Conference (IMC)*, 2006.

[19] J. Mason, S. Small, F. Monrose, and G. MacManus. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, 2009.

[20] U. Payer, P. Teufl, and M. Lamberger. Hybrid engine for polymorphic shellcode detection. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 19–31, July 2005.

[21] M. Pietrek. A crash course on the depths of Win32™structured exception handling, 1997. http://www.microsoft.com/msj/0197/exception/exception.aspx.

[22] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, April 2009.

[23] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2006.

[24] M. Polychronakis, E. P. Markatos, and K. G. Anagnostakis. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007.

[25] M. Shimamura and K. Kono. Yataglass: Network-level code emulation for analyzing memory-scanning attacks. In *Proceedings of the 6th international conference on Detection of Intrusions and Malware, & Vulnerability Assessment (DIMVA)*, 2009.

[26] sk. History and advances in windows shellcode. *Phrack*, 11(62), July 2004.

[27] Skape. Understanding windows shellcode, 2003. http://www.hick.org/code/skape/papers/win32-shellcode.pdf.

[28] Skape. Safely searching process virtual address space, 2004. http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf.

[29] SkyLined. Finding the base address of kernel32 in Windows 7. http://skypher.com/index.php/2009/07/22/shellcode-finding-kernel32-in-windows-7/.

[30] SkyLined. SEH GetPC (XP SP3), July 2009. http://skypher.com/wiki/index.php/Hacking/Shellcode/Alphanumeric/ALPHA3/x86/ASCII/Mixedcase/SEH_GetPC_(XP_sp3).

[31] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, 2007.

[32] P. Ször. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[33] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th Symposium on Recent Advances in Intrusion Detection (RAID)*, Oct. 2002.

[34] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Still: Exploit code detection via static taint and initialization analyses. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2008.

[35] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of the USENIX Security Symposium*, Aug. 2006.

[36] B.-J. Wever. SEH Omelet Shellcode, 2009. http://code.google.com/p/w32-seh-omelet-shellcode/.

[37] G. Wicherski. Win32 egg search shellcode, 33 bytes. http://blog.oxff.net/2009/02/win32-egg-search-shellcode-33-bytes.html.

[38] Q. Zhang, D. S. Reeves, P. Ning, and S. P. Lyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2007.