European Commission
Directorate-General Home Affairs

Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme

HOME/2009/CIPS/AG/C2-050
i-Code: Real-time Malicious Code Identification

Deliverable D1: System Design

Workpackage:	WP1: System Design
Contractual delivery date:	June 2011
Actual delivery date:	July 2011
Deliverable Dissemination Level:	Public
Editor	Herbert Bos (VU)
Contributors	FORTH, POLIMI, EURECOM, TUV
Internal Reviewers:	FORTH

Executive Summary: In this deliverable, we describe the overall system design for the i-Code real-time malicious code detection system, focusing on its subsystems and their integration. The system design brings together the components of network-level attack vector detection forensics tools, techniques for the classification and clustering of shellcode based on the control flow graph, and behavioral-based malware detection by benign-malicious action comparisons. It also incorporates an infrastructure design for malcode detection in high-speed networks.

With the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme. European Commission - Directorate-General Home Affairs[†].

[†]This project has been funded with the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme of European Commission - Directorate-General Home Affairs. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

Contents

List of Figures

CHAPTER 1

Introduction

The i-Code project aims to detect and analyse malicious code and Internet attacks at real time. Its scope includes the detection of attacks in the network and on the host, the analysis of the malicious code, and post-attack forensics. Bringing together such different extremes of the security spectrum in itself makes i-Code both ambitious and valuable.

However, the project is more ambitious than that. Rather than a mere combination of existing tools, the i-Code consortium intends to develop new detection architectures and analysis tools. For instance, the project partners have experience in analyzing and classifying malware (developed in prior projects), but in this project we turn our attention to doing something similar for the initial attack vectors (the shellcode). To do so, we must develop new tools and new analysis techniques, mostly from scratch.

Similarly, the project aims to detect network attacks, and specifically shellcode, using payload execution at wire-speed, that is to say at gigabit rates. While payload execution is well-known, the processing involved is so expensive that it is currently only suitable for very low-speed links: a few tens of megabits per second. This is unfortunate, because payload execution is one of very few methods capable of detecting polymorphic shellcode in the network with high accuracy. Unlike existing network intrusion detection systems (like Snort), it is not easily fooled by code that modifies itself all the time, by one or more levels of packing and encoding. Thus, the need to scale up payload execution is urgent.

Of course, we do not limit ourselves to initial attack vectors. On the contrary, we look at a wide variety of malicious activities, both on the host and in the network. This includes malware that is already installed on a victim machine. What makes i-Code unique, is that it brings together all

these detection and analysis techniques, enabling administrators to correlate events from a variety of sources. Moreover, the shellcode captured by any one of the sensors (say, in-network payload execution) will be analysed by our shellcode analysis component. By combining and processing the information, i-Code will enable forensics on integrated data sets that would be hard to achieve with individual tools.

Outline In this document, we sketch the design of the i-Code architecture, highlighting all the individual components, as well as the way in which they will be integrated. In Chapter 2, we present the overall design at a fairly high level. Chapter 3 will discuss the payload execution by means of emulation. In Chapter 4, we present our design for the classification and clustering of shellcode, while Chapter 5 explains how we will use behavioural patterns to detect malware. Chapter 7 proposes and discusses a layout for the i-Code console. The console will be implemented using web technologies and will be used for browsing and analysing the events collected by the supported sensors. Chapter 6 is devoted in more detail to the I/O architecture that we will develop to speed up payload execution. Finally, in Chapter 8, we will summarise and conclude.

CHAPTER 2

System Design: The Big Picture

i-Code brings together a variety of techniques for real-time detection and analysis of cyber attacks. Since the nature of these techniques varies wildly, we should design a solution that unites a number of different inputs in a meaningful way. In this chapter, we will motivate the need for an integrated security console, and present a high-level design of the i-Code solution. In the next few chapters, we will look in more detail at the components that make up the design.

2.1 The need for an integrated security console

Offering a single view on a variety of sensors and tools benefits administrators by making it easier to correlate events from different sources. Suppose, for instance, that an administrator first observes suspicious data in the network, destined for the non-privileged DNS server on host A. A little while later, the administrator receives a second alert, about an unusual `brk()` system call made by the DNS server. Shortly after that, a third alert arrives, suggesting that an unknown program on host A is hooking certain system functions on the victim system, in a way that resembles that of rootkits.

While none of these alerts are conclusive evidence of an attack by themselves, the three of them occurring in sequence almost certainly represents an attack. Indeed, using *requires/provides* analysis and conclude that there must have been (a) a successful attack on the DNS server by the exploit observed in the network, (b) that subsequently led to privilege escalation by means of the `brk()` kernel exploit to obtain root privileges, which (c) in turn allowed the attackers to install a rootkit by hooking a variety of functions.

Besides better correlation, however, the combination of multiple detection and analysis tools may also improve the usefulness of each of the individual tools. For instance, given a technique to detect shellcode in the network and a technique to analyse shellcode, the integrated console can immediately relay the shellcode samples to the analysis tools. If the analysis tools flags the samples as benign, perhaps no real alert should be presented to the administrator. If, on the other hand, the analysis tool also finds the sample suspicious, it may be a good idea to raise an alarm.

2.2 i-Code System Design

The high-level i-Code design is shown in Figure 2.1. It combines host and network level attack detection tools and various analysis techniques. Alerts are consolidated in a single interface, known as the i-Code console, to facilitate the administrator's tasks. The red arrows indicate the data produced by the various other components that are consumed by the console and, where appropriate, made available to the administrator. The green arrows indicate data streams set up by the security console itself. For instance, the console may enable suspicious code captured in the network to an analysis tool. Similarly, the console may select events to provide to a forensics component.

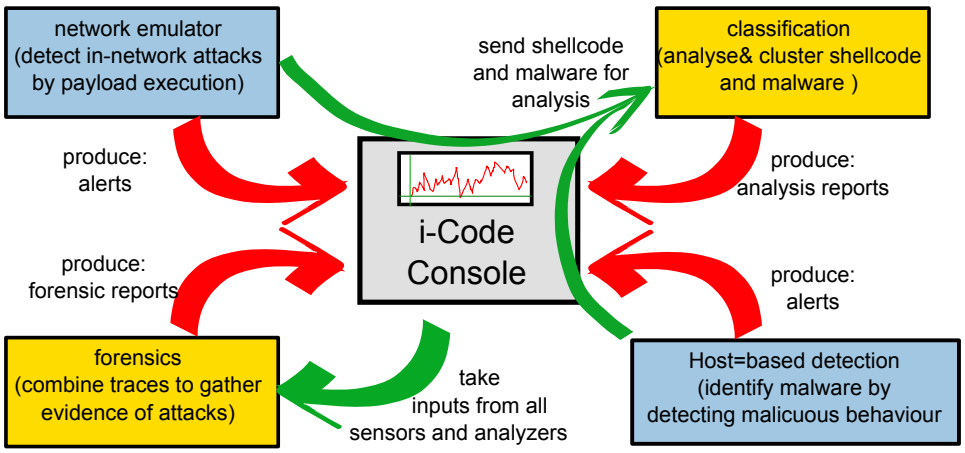

Figure 2.1: i-Code System Design.

In the project, we target novel detection and analysis techniques, although we plan to make the console extensible so that others can plug in

new tools. Specifically, we want to provide for attack detection techniques in the network and on the host:

Network emulation Network emulation is a novel technique to detect an intrusion by means of executing the payload of network traffic on the fly, and verifying whether or not it contains code that looks like an attack (shellcode).

Behaviour-based detection Behaviour-based detection means that we look at the normal behaviour of applications in order to detect deviations that are likely to be caused by malware.

For analysis, the project will provide a clustering technique to classify suspicious shellcode and malware. By clustering such malcode, we can easily check whether something we detected is entirely new, or resembles code that we have seen before. As security software vendors receive many throusands of new samples each day, being able to separate the new ones from the known ones is increasingly important. The process of selecting what alerts to focus on, is known as triage. The i-Code console will help separate "serious cases" from "old news."

2.3 Requirements

The i-Code security console is the central component in the design. It is responsible for the integration of all other components. Since different components use different formats, the integration is not trivial. In general, we designed the i-Code console to meet the following requirements:

Extensible. There are many detection tools and equally many analysis tools. In this project, we primarily aim for novel techniques, but the design will be open and extensible, so other components can be added later.

Even correlation The user should have control over which events and alerts to view in the console. By presenting two event streams next to each other in the console, correlating them will be easier.

Flexible presentation The console should support a variety of techniques to represent events an data. For instance, we may look at alerts in a bar chart, a table, or even in raw format. The console should provide generic methods to allow different data to be represented in a number of ways.

Forwarding. Also, whenever suspicious code is found either on the host, or in the network, we want to make it possible to automatically feed it to a code analysis component that will classify the code to see whether it resembles malcode.

2.4 Meeting the requirements

The console will consist of a viewer and a library of presentation functions—graphs, bar and pie charts, tables, etc. By means of selection (e.g., ticking boxes), administrators using the console can select which data to represent and how to represent it. Thus we achieve the requirements of correlation and flexibility in presentation.

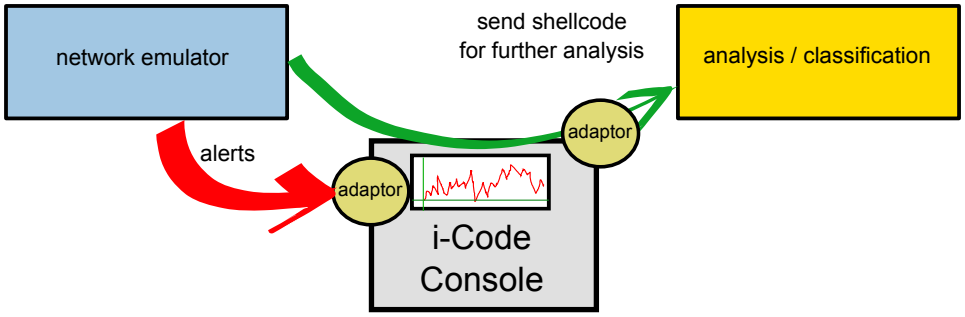

Figure 2.2: Adaptors are responsible for conversion between formats.

The problem is that the formats in which the various detection and analysis techniques expect their inputs and produce their outputs should align in order to integrate the data coherently. For new techniques, we could agree on some common format that is used by all.

The downside of a common format is that it requires a form of standardisation, and standardisation procedures are typically lengthy and tedious. In addition, it precludes the integration of a wealth of existing techniques never designed with the i-Code console in mind.

Rather than standardising on a common format, i-Code takes a pragmatic approach, and relies on *adaptors* to transform data formats from one component to that of another. Adaptation is illustrated in Figure 2.2, which zooms in on the top half of Figure 2.1. If the formats happen to coincide, the adaptor can be very thin, but if not, it will have to handle the type conversion.

Adaptation occurs both when components export data and events to the console, and when the console produces data for the components. In some cases, the console will forward data from one component to another. In that case, format adaptation between the two components is required.

CHAPTER 3

Network-level Emulation

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 [31]. Identifying the presence of the shellcode itself in network data 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 [31, 3, 35, 34]. However, methods based on static analysis cannot effectively handle malicious code that employs advanced obfuscation tricks such as indirect jumps and self-modifications.

In i-Code, we take an alternative approach based on dynamic code analysis using emulation, which is not hindered by such obfuscations and can detect even extensively obfuscated shellcode [23]. In contrast to previous approaches 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 on the emulator. 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. Each heuristic matches inherent runtime patterns that are always exhibited during shellcode execution irrespective of its particular implemen-

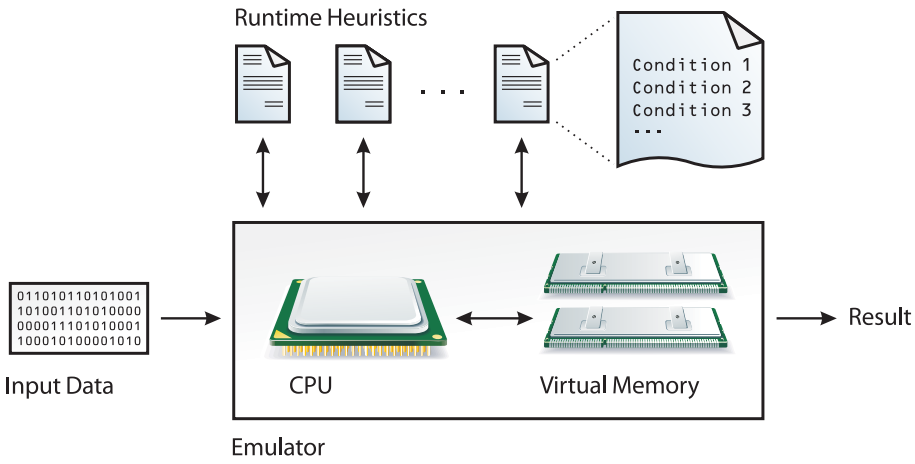

Figure 3.1: Network-level shellcode detection system architecture.

tation. 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.

3.1 Architecture

The shellcode detection subsystem of i-Code 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. 3.1. 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 for other shellcode types and OSes can easily be added due to the extensible nature of the system.

The overall concept can be thought as analogous to the operation of a typical signature-based intrusion detection system, with some key differences: each input is treated as code instead of a series of bytes, the detection engine uses code emulation instead of string or regular expression matching, and each "signature" describes a generic, inherent behavior found in all instances of a particular type of malicious code, instead of an exploit or vulnerability-specific attack vector.

Existing polymorphic shellcode detection methods focus on the identification of self-decrypting behavior, which can be simulated using solely a CPU emulator without any host-level information [24]. 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 i-Code 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 address space of the hypothetical process in the context of which the inspected input is being executed 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 developed within the context of i-Code 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 [4].

The heuristics are orthogonal to each other, which means that more than one heuristic may match during the execution of an actual shellcode, giving increased detection confidence. For example, besides the four new heuristics presented in this paper, we have also incorporated for evaluation purposes a fifth heuristic similar to the GetPC-based polymorphic shellcode detection heuristic used in existing detectors [24]. Since any polymorphic shellcode carries an encrypted version of a plain shellcode, the execution of polymorphic shellcode usually triggers both self-decrypting and plain shellcode heuristics when all five heuristics are enabled.

In the following section, we describe in detail four new detection heuristics for the identification of plain or metamorphic shellcode, egg-hunt shellcode, as well as polymorphic shellcode that uses SEH-based GetPC code.

3.2 Runtime Heuristics

An effective and robust shellcode detection heuristic should fulfil two opposing goals. On one hand, it must be generic enough to capture as many different implementations of the intended execution behavior as possible in order to be robust against evasion attempts. On the other hand, it must be specific enough to precisely describe a large enough set of characteris-

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 3.1: Overview of the shellcode detection heuristics used in the emulation-based detector.

tic runtime operations of the shellcode in order to be resilient against false positives.

Each heuristic used in i-Code is composed of a sequence of conditions that should *all* be satisfied *in order* during the execution of malicious code. A succeeding condition can thus be satisfied only if the preceding condition has already been met. Table 3.1 gives an overview of the four heuristics presented in this section. The heuristics are tailored to the detection of Windows shellcode, given that the vast majority of code injection attacks target this platform. They focus on the identification of the first actions of different shellcode types, according to their functionality, regardless of any self-decrypting behavior.

3.2.1 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, listening for a connection from the attacker and spawning a command shell, or adding a privileged user account. These operations require interaction with the OS through the system call interface, or in case of Microsoft Windows, through the user-level Windows API. Although the Native API exposes an interface for directly calling kernel-level services through `ntdll.dll`, it is rarely used in practice because system call numbers often change between different Windows versions and Service Packs, and most importantly, because it does not provide access to network operations which are mandatory for enabling communication between the attacking and victim hosts.

The Windows API is divided into several dynamic load libraries (DLLs). Most of the base services such as I/O, process, thread, and memory management are exported by `kernel32.dll`, which is always mapped into the address space of every process. Network operations such as the creation of sockets are provided by the Winsock library (`ws2_32.dll`). Other libraries commonly used in shellcode include `urlmon.dll` and `wininet.dll`, which provide handy functions for downloading files specified by a URL. In order to call an API function, the shellcode must first find its absolute address

in the address space of the vulnerable process, and load any missing dlls. 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 functions can be searched either by name, or more commonly, by comparing hashes of their names, which results to more compact code. 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 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.

Another method to resolve the required functions relies on the DLL's Import Address Table (IAT). The shellcode has to first load using `LoadLibrary` a DLL that depends on the same set of functions that need to be used in the shellcode, and then directly reads the VMAs of the required functions from the IAT. The address of `LoadLibrary` is again resolved through the EDT. However, this technique is prone to changes in the offsets of the imported symbols across different DLL versions.

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`, which is guaranteed to be present in the address space of the exploited process, and from there get a handle to its EDT. After resolving `LoadLibrary`, a handle to any other DLL can be easily obtained. 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. In the rest of this section, we present two heuristics that match the most widely used `kernel32.dll` resolution techniques.

Of course, a naive attacker could hard-code the VMAs of the required API functions in the shellcode, and call them directly on runtime. This however would result to highly unreliable shellcode because DLLs are not always loaded at the same address, and the function offsets inside a DLL may vary. The increasing use of security measures like DLL rebasing and address space layout randomization makes the practice of using absolute memory addresses even less effective. A more radical way of resolving an API function would be to scan the whole address space of the vulnerable process and locate the actual code of the function, e.g., using a precomputed hash of its first few instructions. This technique requires a reliable way of scanning the process address space without crashing in case of an illegal

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

access to an unmapped page. We discuss heuristics that match this memory scanning behavior in Sec. 3.2.2.

3.2.1.1 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. Of particular interest is a pointer to the `PEB_LDR_DATA` structure, which holds information about all loaded modules, including a list of the loaded DLLs in the order they have been initialized. The record for `kernel32.dll` is always present in the second position of the list (after `ntdll.dll`), and among its contents is a pointer to the base address where the DLL has been loaded. By walking through the above chain of data structures, the shellcode can resolve the absolute address of `kernel32.dll` in a reliable way.

Figure 3.2 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. 3.3. 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 to the PEB 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. 3.3), the following code will correctly not match the above condition:

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

Although the second instruction reads from address `0x7FFDF030`, it does not match the condition because the effective address computation in the second operand does not involve the `FS` register. and thus it does not belong to actual shellcode.

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. 3.2 with the following code:

```
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 temporarily 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. 3.3. 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. 3.2). 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. 3.2) allows for strengthening further the detection heuristic with a third condition.

Although this is the most well known 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. 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. 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.* We could strengthen further the heuristic with more conditions based on other subsequent mandatory operations, e.g., the actual read of the `BaseAddress` field once the record is found. However, these three conditions are enough for building a robust detection heuristic without false positives.

3.2.1.2 Backwards Searching

An alternative technique for locating the base address of `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 [28]. Searching can be implemented efficiently by exploiting the fact that in Windows, DLLs are loaded only in 64KB-aligned addresses [28]. The DLL can be identified by looking if the first two bytes of each 64KB-aligned address are equal to `MZ`, the beginning of the MS-DOS header of the DLL, or by checking other characteristic values in the DLL headers.

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, which provides a unified way of handling hardware and software exceptions. When an exception occurs, the exception dispatcher

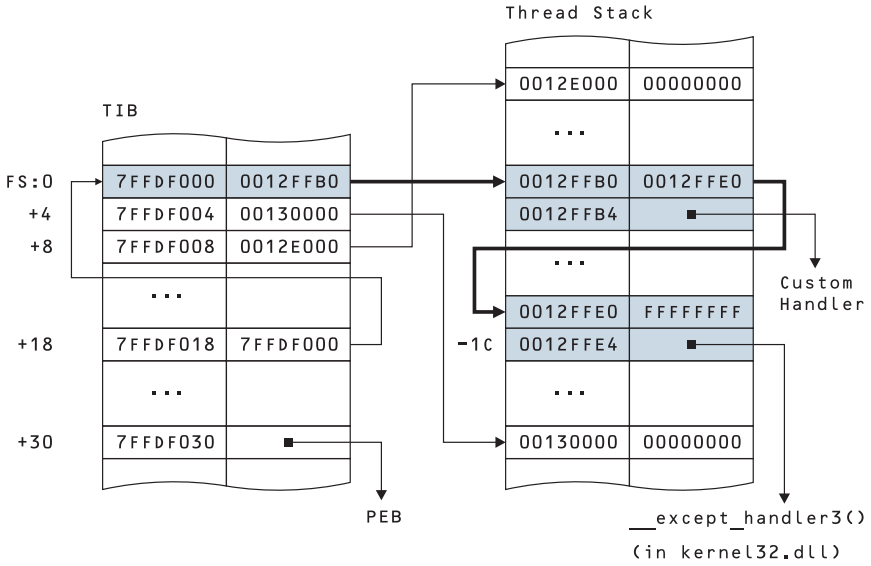

Figure 3.3: A snapshot of the TIB and the stack memory areas of a typical Windows process. The SEH chain consisting of two nodes is highlighted.

walks through a list of exception handlers for 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 3.3 shows a typical snapshot of the TIB and the stack memory areas of a process with two SEH handlers. This mechanism allows each function to easily install an exception handler that has priority over the preceding handlers by pushing a new SEH frame on the stack. 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. 3.3. 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.

Figure 3.4 shows an example of code that uses the above technique [28]. The shellcode has to first get a handle to the current SEH frame through `FS:[0]` (line 2), and then walks through the SEH chain (lines 4–8). Starting from the address pointed to by the `Handler` field of the last frame (line 9), the code then searches backwards in 64KB increments for the base address of `kernel32.dll` (lines 10–14).

Another technique to reach the last SEH frame, known as "TOPSTACK" [28], uses the stack of the exploited thread. The default exception handler is reg-

```
 1:   xor  ecx, ecx            ; ecx = 0
 2:   mov  esi, fs:[ecx]       ; esi = current_frame
 3:   not  ecx                 ; ecx = 0xffffffff
 4: find_last_frame:
 5:   lodsd                    ; eax = current_frame->Next
 6:   mov  esi, eax            ; esi = current_frame->Next
 7:   cmp  [eax], ecx          ; current_frame->Next == 0xffffffff?
 8:   jne  find_last_frame     ; if not, continue searching
 9:   mov  eax, [eax + 0x04]   ; eax = current_frame->Handler
10: find_kernel32_base:
11:   dec  eax                 ; Subtract to previous page
12:   xor  ax, ax              ; Zero lower half (64KB-align)
13:   cmp  word [eax], 0x5a4d  ; are the first 2 bytes == 'MZ'?
14:   jne  find_kernel32_base  ; if not, continue searching
```

Figure 3.4: Example of code that resolves the base address of `kernel32.dll` using backwards searching.

istered 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 (or the top, when thinking in terms of the memory layout) of the stack of the current thread 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. 3.3. In fact, the TIB contains a second pointer to the top of the stack at `FS:[0x8]`. By adding the proper offset, this pointer can also be used for accessing the default SEH handler, although this approach is less robust because some applications may have altered the default stack size.

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 for locating `kernel32.dll` 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. (e.g., as in line 9 in Fig. 3.4). 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. (e.g., as in line 13 in Fig. 3.4). 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`.

Note that a more obscure search routine may search for other characteristic byte sequences in the DLL, and thus the reads may not necessarily be made at 64KB-aligned addresses. Although the condition can be made more rigorous by requiring the execution of more than one memory reads within `kernel32.dll`, even one read operation is enough for a robust heuristic.

3.2.2 Process Memory Scanning

In the vast majority of code injection exploits, the first step of the shellcode is to resolve `kernel32.dll` and then all required API functions. However, 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, in the address space of the exploited process, 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 [29]. Egg-hunt shellcode has been used in various remote code injection exploits, while recently it has found use in malicious documents that upon loading attempt to exploit some vulnerability in the associated application, making its effective detection of critical importance.

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.

3.2.2.1 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. 3.2.1.2, 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 [29], 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-aligned within the stack and is located higher than the previous SEH frame. 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.

3.2.2.2 System Call

Structured Exception Handling has been extensively abused for achieving arbitrary code execution in various memory corruption vulnerabilities by overwriting the `Handler` field of the current SEH frame instead of the return address, especially after the wide deployment of cookie-based stack protection mechanisms. This led to the introduction of SafeSEH, a linker option that produces a table with all the legitimate exception handlers of the image. When an exception occurs, the exception dispatcher checks if

```
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 3.5: A typical system call invocation for checking if the supplied address is valid.

the handler function to be called is present in the table, and thus prohibits the execution of any injected code through a custom or overwritten handler.

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 [29, 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 [29]. In Windows, a system call is initiated by generating a software interrupt through the `int 0x2e` instruction.

Figure 3.5 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 arbi-

trary 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 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. Condition C1 alone can happen to hold true during the execution of random code, although rarely. The shellcode should perform a mandatory check for the `STATUS_ACCESS_VIOLATION` return value, but we cannot specify a robust condition for this operation since the comparison code can be easily obfuscated. 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, the shellcode will have to search hundreds or thousands of addresses, e.g., by repeatedly calling the code in Fig. 3.5 in a loop. Hence, during the execution of an egg-hunt shellcode, 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.* Based on our analysis, 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.

3.2.3 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 [24]. 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 [20], 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.

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.

In essence, the SEH-based memory scanning detection heuristic described in Sec. 3.2.2.1 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.

CHAPTER 4

Shellcode Analysis and Classification

4.1 Shellcode Analysis and Collection Architecture

As a first step in the study of shellcodes that are being used by attackers in the wild, we will need to develop an infrastructure for collecting shellcode samples from diverse sources. Previous work on the analysis and classification of shellcode [18] has concluded that the variety of shellcode observed in the wild is extremely limited. We argue that this conclusion is largely a consequence of the limitations of the shellcode collection techniques deployed by the authors. Specifically, the authors of [18] collected network traces of communication with server honeypots, and detected and extracted shellcodes using Shield [33]. This approach, however, suffers from three significant limitations. The first limitation is related to the class of attacks observed by server honeypots: Server honeypots are completely blind to all client-side attacks. Since client-side attacks are now the biggest threat to internet security [2], this is an important limitation. The second limitation is intrinsic in attempting to extract shellcodes from network traces. It can be difficult or in some cases impossible to obtain the shellcode contained in a network trace; for instance, if the shellcode is fragmented across protocol packets, enclosed within an encrypted stream, or hidden inside obfuscated Javascript within a PDF document. The final limitation is that Shield can only detect attacks against known vulnerabilities based on a set of signatures. This further limits the variety of shellcodes that can be collected to those for which a signature has been written by a human analyst.

To overcome these limitations and attempt the largest and most diverse shellcode collection effort to date, we plan to employ a flexible, loosely

structured infrastructure that can receive shellcodes collected with arbitrary detection tools. At the center of this infrastructure is a generic shellcode behavioral analysis tool, that will be discussed in Section 4.2. This tool will offer an easy-to-use web interface for submitting shellcodes as raw binary blobs. This interface will be open to the public. Thus, anyone will be able to submit a shellcode they have collected to our analysis infrastructure. In return, submitters will obtain a report on the behavior of the submitted shellcode within our analysis sandbox, as discussed in Section 4.2. The goal of this collection effort is to assemble samples of shellcode obtained from three broad classes of sensors:

Server Honeypots. These are systems deployed on the network with the purpose of acting as network servers and being exploited by network attacks. This class of sensors can fall victim to the same class of attacks discussed in [18]. However, unlike [18] we are not limited to extracting shellcodes for known attacks from network traces. Modern honeypot technology, such as Argos [25], can use fine-grained instrumentation and techniques such as dynamic data tainting to perform host-level, 0-day detection of exploitation attempts.

Client Honeypots. These are systems that act like network clients, and attempt to detect malicious content such as malicious web sites or documents by crawling the web or processing the content of SPAM emails. For detection, these systems can rely on a variety of technologies [11, 27, 22, 32], but can also employ techniques similar to the ones employed by server honeypots such as Argos.

Network-level Detectors. These are detectors that operate at the network level, and are able to recognize shellcode in the raw network traffic. This is typically based on some form of emulation, as is the case for Nemu and for its extensions discussed in Chapters 3 and 6. Clearly, we plan to integrate the shellcode detection tools developed and deployed within i-Code into our shellcode collection and analysis effort. Thus, shellcodes detected on the network by the i-Code tools will be automatically submitted for analysis.

To maximize the amount and diversity of the shellcodes that we collect, rather than simply rely on user submissions, we plan to collaborate with the operators and developers of server and client honeypots deployed within and without the i-Code consortium, to automate the submission of shellcodes from the largest possible number of honeypots.

4.2 Behavioral Analysis and Unpacking

Once a shellcode is submitted, it will be subjected to dynamic analysis, with the goal of observing the intended behavior of the attack. For this, the first step is to wrap the raw submitted code into an executable format. Then, the shellcode can be executed inside an instrumented and sandboxed virtual environment. For this second step we plan to employ a modified and extended version of the Anubis malware analysis tool [6, 1, 5]. Anubis is a dynamic malware analysis system based on an instrumented Qemu [7] emulator. It is offered as an open service through a public website, where users can submit binaries for analysis, and receive a report that describes the behavior of the sample in a human-readable way. We plan to follow the same approach for shellcode analysis. For this, the behavioral report will be enriched with shellcode-specific information. For instance, the report will include a high-level description of the method used by the shellcode to consolidate the attacker's presence on an exploited system, such as downloading and installing a binary or creating a user for the attacker.

In addition to a behavioral report similar to the one produced by Anubis, this tool will detect the shellcode's unpacking behavior and dump the unpacked code. This step is essential for analysing the phylogeny of shellcode by means of the classification techniques discussed in the following Section.

4.3 Shellcode Classification

Once the behavioral analysis phase has dumped an unpacked version of a shellcode's code, we can use static techniques to further analyse this code. In particular, we are interested in studying the variety of the shellcodes available in the wild, and their phylogeny. To what extent are similar shellcodes employed in exploits of different vulnerabilities? Do attackers write their own shellcodes or mostly rely on exploitation tools such as metasploit[1]? Do client and server-side exploits employ different shellcodes? To be able to answer such questions, we require a way to perform unsupervised classification (or clustering) of the collected shellcodes. For this, we plan to employ techniques based on the static analysis of the unpacked code, similar to the ones employed by Ma et al. [18]. In particular, we plan to refine the code comparison techniques based on the CFG-based code fingerprinting from Kruegel et al. [17].

[1] http://www.metasploit.com/

CHAPTER 5

Behaviour-based Detection of Malcode

5.1 Motivation

Given the importance and the security impact of malware, it is not surprising that there exists a significant body of research, both in the scientific community and the commercial world, on ways to protect machines from becoming infected and on techniques to detect and contain malware programs once they are on the host. The most popular approach to identify malware is based on signatures. These signatures are typically byte strings (or instruction sequences) that are characteristic for a particular malware instance or a family of malicious code [30]. Unfortunately, code obfuscation and polymorphism have long proved to be effective tools in the arsenal of a malware author to evade signature-based detection.

To address the limitations of signature-based detection techniques, behavior-based detection was introduced as a novel approach to identify malicious code [9, 16]. Behavior-based detectors do not examine the (static) content of a binary, but rather focus on the (dynamic) actions that the program performs, or might perform. The idea is that even when the syntactic layout of the program is different, the semantics of the code should remain unchanged between polymorphic variants of the same malware (or even, between different members of the same malware family).

Arguably the most popular way to characterize the behavior of programs is based on some kind of analysis of the system calls (or Win32 API functions) that a program invokes; or that it can invoke, in case the code is examined statically. Various models range from looking at sequences of system calls [21], over bags of system calls [14], to system call patterns based

on data flow dependencies [19, 16]. Yet other techniques look at individual system calls, but take into account argument information [15].

In most cases, authors achieved good results with system-call-based malware detectors, and they reported high success rates with low numbers of false positives. However, a closer examination of the presented results reveals that most experiments are performed on a relatively small scale. In particular, this is true for the analysis of false positives. That is, authors collect traces only for a small set of benign applications. In addition, these programs are exercised in a limited fashion, often using synthetic inputs or launching simple test cases. As a result, it is not clear whether the observed system call traces produced by these benign applications are representative for the diverse set of applications that is used by actual users. Moreover, in most cases, the experiments are performed on a single machine. It is not clear how the detection results will generalize to a larger installation base. Thus, the reported number of false positives might be underestimated.

5.2 Data Collection

During the first part of the project, we set out to analyze the diversity of system call information and the robustness of simple detection techniques when looking at data that is collected on a larger scale. To this end, we developed a light-weight system call collection module that was installed on ten machines that were used by people to carry out their normal activities. Over a period of several weeks, these modules collected more than 1.5 billion system calls that were invoked by 362 thousand operating system processes. In total, we observed 242 different applications.

Our analysis of the collected data shows that there is a large diversity in the system calls that benign programs invoke. This makes it difficult for system-call-based detectors to establish a concise model of normality that allows for a clean separation between acceptable and malicious behavior. In other words, the fact that a previously unseen system call sequence is observed is not a good indication that the process is malicious. It might just be that an existing application is used in an unexpected fashion, or that a new, benign application is installed.

In this section, we discuss our efforts to collect a large and diverse set of system call traces. Our requirements are geared towards imposing the least impact on the users whose machines are part of the data collection effort. Thus, the data collection framework must have minimal impact on the performance of those machines, must operate with and without network connectivity, must ensure that private information does not leave the user's machines, and must make almost no assumptions about the run-time environment. For example, requiring that users make use of virtual machines would significantly restrict the practical applicability of our data collection.

Program	*System call*	*Arguments*		*Return value*
		In	*Out*	
SVCHOST.EXE	NtCreateFile	131208,'..\ACGENRAL.DLL',...	2600	0
SVCHOST.EXE	NtQueryInformationFile	2600,6,0		-144573084
SVCHOST.EXE	NtClose	1004		0
CLIENT.EXE	NtClose	1560		0
CLIENT.EXE	NtCreateNamedPipeFile	2148532480,'..\NamedPipe',...	288	0
CLIENT.EXE	NtOpenFile	1074790528,'..\NamedPipe',...	264	0
firefox.exe	NtReleaseSemaphore	404		0
firefox.exe	NtReadFile	780		0

Figure 5.1: Sample system-call log. Due to formatting constraints, some values are abbreviated and the timestamp, process ID, and parent process ID fields are not shown.

Additionally, the data collection framework must be capable of extracting a rich set of attributes for each event (i.e., system call) of interest. Unfortunately, none of the existing system call tracing tools satisfy these requirements, so we built and deployed our own data collection framework.

Our system consists of software agents, which, once installed on user's machines, automatically collect, anonymize, and upload system call logs, and a central data repository, which receives logs from each machine and normalizes the data in preparation for further analysis. The software agents can be installed by users on their own machines and are mindful of system load, available disk space, and network connectivity. Furthermore, users can enable and disable the collection agent as they wish.

Data description. We are interested in performing different types of analyses on the collected data. Thus, the data elements collected for each system call must allow analyses along many dimensions. For each system call we collect its arguments, its result (return) code, the process ID, the process name, and the parent process ID. Each log entry is a tuple (see an example in Figure 5.1):

$$\langle timestamp, program, pid, ppid, system\ \ call, args, result \rangle$$

This data allows us to perform our analyses within a single process, across multiple executions of the same program, or across multiple programs.

5.2.1 Raw Data Collection

The software agent that collects data is a real-time component running on each user's machine. This agent consists of a data collector and a data anonymizer. We implemented our agent for Microsoft Windows, as it is the OS targeted the most by malware. The description in the remainder of this section provides details specific to the Microsoft Windows platform. The data collector is a Microsoft Windows kernel module that traces system call events and annotates them with additional process information. The data anonymizer transforms the collected system call data according to privacy

rules and uploads it to the remote, central data repository.

Kernel collector. The main goal of this component is to collect system call and process information *across the entire system*. In order to intercept and log system call information, the kernel data collector hooks the SSDT table [13]. The kernel collector logs information for 79 different system calls in five categories: 25 related to files, 23 related to registries, 25 to processes and threads, one related to networking, and five related to memory sections. We select the same subset of 79 syscalls already used in Anubis [1] which covers the relevant operations that manipulate persistent OS resources as well.

A challenge arises from the fact that the kernel collector does not necessarily observe the start of a new process. One reason is that the user can disable and re-enable the software agent at any point. Another reason is that the kernel collector is started as the last kernel module in the system boot process. This means that the kernel collector might observe system calls that refer to previously acquired resource handles, but without having any information about which resources those handles point to. As a special case, some resource handles (e.g., handles to the registry roots) are automatically provided to a process by the OS at process-creation time. Consequently, if we log only the parameters for each individual system call that we observe, we lose information about previously (or automatically) acquired resources. To address this problem, we query the open handler table for each process we have not seen before. This allows the kernel collector to retrieve the open objects already associated with a new process. We store the path names of these objects for later use when we intercept a system call (such as `NtOpenKey`) that references a pre-existing handle.

Log anonymizer. To protect the privacy of our users, we obfuscate or simply remove arguments of various system calls before sending the log to the data repository. The obfuscation consists of replacing part or whole of a sensitive argument value with a randomly-generated value. Every time a value repeats, it is replaced with the same randomly-generated value, so that we can recover correlations between system call arguments. We consider as sensitive all arguments whose values specify non-system paths (e.g., paths under `C:\Documents and Settings` are sensitive), all registry keys below the user-root registry key (`HKLM`), and all IP addresses. Furthermore, we remove all buffers read, written, sent, or received, thus both providing privacy protection and reducing the communication to the data repository. The data repository indexes the logs by the primary MAC address of each machine.

Impact on performance. We designed the software agent to minimize the overhead on users' activities. The kernel module collects information

only for a small subset of 79 system calls. Log are saved locally and processed out of band before being sent to the server, when network connectivity is available. Users can turn data collection on and off, based on their needs. Local logs are uploaded to the repository when they reach 10 MB in size and logging is automatically stopped if available disk space drops below the 100 MB threshold. Each 10 MB portion of the system call log is compressed using ZIP compression, for an 95% average reduction in size (from 10 MB to 500 KB). Given these techniques, we are confident that users were able to use their computers with the data collector present as they would normally do, and thus the collected system call logs are representative of day-to-day usage.

5.2.2 Data Normalization

The purpose of this component is to process the raw system call logs and extract the fully qualified names of the accessed resources as well as the access type. For files and directories, the fully qualified name is the absolute path, while for registry keys, it is the full path from one of the root keys.

To compute fully qualified resource names, we track for each process the set of resources open at any given time, via the corresponding set of OS handles. When a resource (file or registry key) is accessed relative to another resource (either opened by the process or opened by the OS automatically for the process), we combine the resource names to obtain a fully qualified name. Symlinks are handled observing the actual open operation of the target (linked) file. To handle the hardlinks we include the locations (paths) of all hard-linked aliases of a file.

Computing the access type (e.g., read, write, or execute) requires tracking the access operations performed on a resource. This is more tricky than expected. When a resource is acquired by a program (e.g, a file is opened), the program specifies a desired level of access. This information, however, is not sufficiently precise for our needs. This is because, often, programs open files and registry keys at an access level beyond their needs. For example, a program might open a file with `FULL_ACCESS` (i.e., both read and write access), but afterward, it only reads from the file. Since we are interested in the actual access type, we track all of the operations on a resource, and only when the resource is released (on `NtClose`), we compute the access type as a union of all operations at all on the resource. If the program performs no operations on a resource, then we use the initially-requested access (provided at resource acquisition) as actual access. Such an heuristic is used for the memory-mapped files. With such files, we might not see any read/write operation at the system call level, although the file is accessed.

In Microsoft Windows, there is no single system call that starts a new process from a given executable file. In order to retrieve the execution path and file name, the normalizer needs to recognize the `NtOpenFile` system

Machine	*Data* (GB)	*System calls* ($\times 10^6$)	*Processes* ($\times 10^3$)	*Applications*
1	18.0	285	55.1	90
2	4.5	70	22.4	87
3	5.6	89	17.7	46
4	32.0	491	110.9	41
5	34.0	514	125.6	42
6	14.0	7	2.8	73
7	1.3	19	3.7	49
8	1.2	18	3.0	22
9	1.6	27	8.5	47
10	2.3	36	12.9	26
Total	114.5	1,556	362.6	242

Table 5.1: Characteristics of our data set.

calls that belong to the process-creation task. When a process is created, the OS executes a set of system calls to allocate resources, load the binary executable, and start the new process: `NtOpenFile`, `NtCreateSection` with desired executable access, and `NtCreateThread`. Consequently, we automatically identify occurrences of this pattern and extract the executable path and file name.

5.2.3 Experimental Data Set

We deployed our data collection framework on ten different machines, each belonging to a different user, all running Microsoft Windows XP. The users have different levels of computing expertise and different computer usage patterns. Based on use, the machines can be classified as follows: two were development systems, one was an office system, one was a production system, four were home PCs, and one was a computer-lab machine.

Overall we collected 114.5 GB of data, consisting of 1.556 billion of system calls, from 362,600 processes and 242 distinct applications. 5.1 provides detailed information for each machine.

Our system collected data from each machine at an average rate of 8.2 MB/minute, with highly used machines producing logs at 40 MB/minute and idle machines producing 1.5 MB/minute. In 5.2, we report the logging time for the ten different machines. For each machine, we show the machine's usage profile, the size of data collected, the total time during which data was actually collected, the time period between the first log entry and the last log entry, and the average data rate. For example, the fourth row indicates that machine 4 was a production server that generated 32 GB of system call logs, over a period of 3 days, during which data collection was

Machine	*Usage*	*Data*	*Time*		*Data rate*
			Logged	*Total*	
		(GB)	(hours)	(days)	(MB/minute)
1	office	18.0	12	3	8
2	home	4.5	4	3	6.25
3	home	5.6	3	4	7.77
4	prod.	32.0	12	3	14
5	prod.	34.0	12	3	15
6	lab	14.0	8	3	11
7	home	1.3	3	2	4
8	home	1.2	3	2	4
9	dev.	1.6	2	2	6
10	dev.	2.3	2	3	6.4

Table 5.2: Data rates during collection.

active for 12 hours. Our training data included every OS task that happened during the monitoring period, including software installation and updates.

5.3 System Design

Now that we collected a large amount of examples of benign behavior, we can compare them with the known malicious behavior exhibited by malware programs under Anubis. To goal is to try to determine if there is any significant difference that can be used to distinguish the ones from the others.

In particular, we plan to run two sets of experiments. First we want to evaluate known *program-centric* detection models (based, for example, on sequences of system calls) to see how they perform on a real and large dataset. Second, we want to implement a *system-centric* approach to see if it can achieve a better detection rate without raising false positives.

In the rest of the Section we describe the design of the system-centric approach we have in mind.

5.3.1 System-Centric Approach to Detect Malicious Behavior

In this section, we propose the design of model that attempts to abstract from individual program runs and that generalizes how, in general, benign programs interact with the operating system. For capturing these interactions, we focus on the file system and the registry activity of Microsoft Windows processes. More precisely, we record the files and the registry entries that Windows processes read, write, and execute (in case of files only). It is possible to integrate other kinds of interactions into our model (in particular, the network), but we leave this for future work.

Our model is based on a large number of runs of a diverse set of applications, and it combines the observations into a single model that reflects the activities of *all* programs that are observed. For this to work, we leverage the fact that we see "convergence." That is, even when we build a model from a subset of the observed processes, the activity of the remaining processes fits this model very well. Thus, by looking at program activity from a system-centric view—that is, by analyzing how benign programs interact with the OS—we can build a model that captures well the activity of these programs. Of course, this would not be sufficient by itself. To be useful, our model must also be able to identify a reasonably large fraction of malware. Note that it is also possible that program-centric models converge at one point. However, the results presented in the previous section indicate that a large amount of data is needed before this point is reached; more than we collected in our experiments, and definitely more than previous research has used to demonstrate that system-call-sequence-based detection works.

5.3.2 Creating Access Activity Models

To capture normal (benign) interactions with the file system and the Windows registry, we propose *access activity models*. An access activity model specifies a set of labels for operating system resources. In our case, the OS resources are directories in the file system and sub-keys in the registry (sub-keys are the equivalent of directories in the file system). In the following, we refer to directories and sub-keys as folders.

Note that we do not specify labels directly on files or registry entries. The reason for this was that the resulting models are significantly smaller when looking at folders only. As a result, the model generation process is faster and "converges" quicker (i.e., less training data is required to build stable models). Moreover, in almost all cases, the labels for the folder entries (files or registry keys) would be similar to the label for that folder itself. Thus, the sacrifice in precision was minimal.

A label L is a set of access tokens $\{t_0, t_1, \ldots, t_n\}$. Each token t is a pair $\langle a, op \rangle$. The first component a represents the application that has performed the access, the second component op represents the operation itself (that is, the type of access).

In our current system, we refer to applications by name. In principle, this could be exploited by a malware process that decides to reuse the name of an existing application (that has certain privileges). In the future, we could replace application names by names that include the full path, the hash of the code that is being executed, or any other mechanism that allows us to determine the identity of the application that a process belongs to. In addition to specific application names, we use the star character ($*$) as a wildcard to match any application.

The possible values for the operation component of an access token are `read`, `write`, and `execute` for file-system resources (directories), and `read` and `write` for registry sub-keys.

Initial access activity model. An initial access activity model precisely reflects all resource accesses that appear in the system-call traces of all benign processes that we monitored (we call this data set the training data). Note that for this, we merge accesses to resources that are found in different traces and even on different Windows installations. In other words, we build a "virtual" file system and registry that contains the union of the resources accessed in all traces.

Whenever an application `proc` opens or reads from an existing file `foo` in directory `C:\path\dir`, we insert the directory `dir` into our "virtual" file system, including all directories on the `path` to `dir`. When a prefix of the directories along `path` already exist in our virtual file system, then these directories are re-used. All directories that are not already present (including `dir`) are added to the virtual file system tree. Then, we add the access token $\langle$`proc`, `read`$\rangle$ to the label associated with `dir`.

When a process creates or deletes a file in a directory `dir`, or when it writes to a file, then we use the operation `write` for the access token. Similar considerations apply for read and write operations that are performed on the registry. Finally, whenever a binary is executed (loaded by the OS loader), then we add a token with `execute` to the directory that stores this binary.

For example, consider that file `C:\dir\foo` is read by pA on machine A, and that file `C:\dir\sub\bar` is written by pB on another machine B. Then, the resulting virtual file system tree would have `C:\` as its root node. From there, we have a link to the directory `dir`, which in turn has a link to `sub`. The label associated with `dir` is $\langle pA, read\rangle$, and the label associated with `sub` is $\langle pB, write\rangle$.

Pre-processing. Before model generation can proceed, there are two additional pre-processing steps that are necessary. First, we need to remove a small set of benign processes that either read or execute files in many folders. The problem is that these applications appear in many labels and could lead to an access activity model that is less tight (restrictive) than desirable. We found that such applications fall into three categories: Microsoft Windows services (such as Windows Explorer or the command shell) that are used to browse the file system and launch applications; desktop indexing programs; and anti-virus software. The number of different applications that belong to these categories is likely small enough so that a manually-created white list could cover them. In our system, we remove all applications that read or execute files in more than ten percent of the directories. We found a total of 15 applications that fit this profile: nine Windows core services, two desktop indexing applications, and six anti-virus (AV) programs. Identify-

ing such applications automatically is reasonable, because we assume that our training data does not contain malicious code. However, the number of white-listed applications is so small that the entries can be easily verified manually.

The second pre-processing step is needed to identify applications that start processes with different names. We consider that two processes with different names belong to the same application when their executables are located in the same directory. We have found 14 applications that start multiple processes with different names. These include well-known applications such as MS Office, Messenger, Skype, and RealPlayer. Of course, all Windows programs that are located in `C:\Windows\system32` are also aggregated (into a single meta-application that we refer to as `win_core`). Merging processes that have different names but that ultimately belong to the same application is useful to create tighter access activity models.

Model generalization. Based on the initial access activity model, we perform a generalization step. This is needed because we clearly cannot assume that the training data contains all possible programs that can be installed on a Windows system, nor do we want to assume that we see all possible resource accesses of the applications that we observed. Also, the initial model does not contain labels for all folders (recall that the access is only recorded for the folder that contains the accessed entity).

The generalization step performs a post-order traversal of both the virtual file system tree and the virtual registry tree. Whenever the algorithm visits a node, it performs the following four steps:

Step 1: First, the algorithm checks the children of the current node to determine whether access tokens can be *propagated* upward in the tree. Intuitively, the idea is that whenever we inspect a folder (node) and observe that all its sub-folders are accessed by a single application only, we assume that the current folder also belongs to this application.

More formally, the upward propagation rule works as follows: For each operation *op*, we examine the labels of all child nodes and extract the access tokens that are related to *op*. This yields a set of access tokens $\{t_1, \ldots, t_n\}$. We then inspect the applications involved in these accesses (i.e., the first component of each token t_i). When we find that all accesses were performed by a *single* application `proc`, we add the access token $\langle proc, op \rangle$ to the current label.

Step 2: The upward propagation rule of Step 1 is used to identify parts of the file system or the registry that belong to a single application. However, this is problematic when considering *container* folders. A container is typically a directory that holds many "private" folders of different applications. A private folder is a folder that is accessed by a single application only (including all its sub-folders). A well-known example of a container is the

directory `C:\Program Files`, which stores the directories of many Windows programs.

Since a container holds folders owned by many different applications, its label would deny access to all sub-folders that were not seen during training. This might be more restrictive than necessary. In particular, we would like to ensure that whenever an application accesses a previously-unseen folder in a container, this should be allowed. Intuitively, the reason is that this access follows an expected "pattern," but the specific folder has not been seen during training. To handle these cases, we introduce a special flag that can be set to mark a folder as a container.

The following rule is used to mark a folder as a container: Similar to before, we examine the labels of all child nodes and extract the access tokens that are related to each operation *op*. We then inspect the set of access tokens that is extracted $\{t_1, \ldots, t_n\}$. When the applications in these accesses are different, but there is *no wildcard* present in any access token, then the folder is marked as container.

Step 3: Next, the access tokens in the label associated with the current node are *merged*. To this end, the algorithm first finds all access tokens that share the same operation *op* (second component). Then, it checks their application names (first components). When all tokens share the same application name, they are all identical, and we keep a single copy. When the application names are different, or one token contains the wildcard, then the tokens are replaced by a single token in the form $\langle *, op \rangle$. Merging is useful to generalize cases in which we have seen multiple applications that perform identical operations in a particular folder, and we assume that other applications (which we have not seen) are also permitted similar access.

Step 4: Finally, the algorithm adds access tokens that were likely missed because of the fact that the training data is not complete. More precisely, for each access token that is related to a *write* operation, we check whether there exists a corresponding *read* token. That is, for all applications that have written to a folder, we check whether they have also performed read operations. If no such token can be found, we add it to the label. The rationale for this step is that an application that can write to resources in a folder can very likely also perform read operations. While it is possible to configure files and directories for write-only access, this is very rare. On the other hand, adding read tokens allows us to avoid false positives in the more frequent case where we have simply not seen (legitimate) read operations in the training data.

When the generalization algorithm completes, all nodes in the virtual file system and the registry tree have a (possibly empty) label associated with them.

Note that, for building the access activity model, we do not require any knowledge about malicious processes. That is, the model is solely built

from generalizing observed, good behavior. This is an advantage compared to the n-gram model introduced in the previous section, which requires training traces captured from malware runs to identify those n-grams that are unique to benign applications.

CHAPTER 6

A Scaleable I/O Architecture

Network-level attack detection systems have a hard time keeping up with link rates. We have already seen that this is true for Nemu (Chapter 3), but also for other intrusion detection techniques, it becomes increasingly hard to keep up with network speeds.

In i-Code, we develop an I/O architecture to help speed up network-based intrusion detection systems, by reducing the OS bottlenecks in accessing and processing network traffic. Specifically, our design will reduce overhead due to copying, context switching, signalling, and cache invalidations.

Our design builds on Streamline [12], but we aim to make it more usable for general IDS applications. To do so we need to implement a proper TCP/IP stack on top of the Streamline buffer management system. In the remainder of this section, we sketch the design as well as how it will be extended in i-Code.

6.1 Bottlenecks in network processing

The bottleneck in system software has moved from the CPU to the memory system, especially for I/O-intensive tasks such as networking. Operating systems have not structurally changed to reflect this reality, with the result that architectural decisions made in the past hinder applications today. They waste CPU cycles copying data between kernel subsystems and across memory protection boundaries. They waste cycles switching tasks too frequently. To make matters worse, they waste cycles refreshing caches as a result of all this copying and context switching.

An application binary interface (ABI) at the abstraction level of Posix calls incurs many mode switches between userspace and kernel mode by handling packets one at a time; multi-user OS access control imposes copy semantics across memory protection domains even on dedicated (i.e., single user) servers. These inefficiencies result from fundamental architecture choices and can only be resolved through comprehensive OS restructuring. Failure to resolve the issues systematically has led to application-specific solutions, such as disk caches duplicated in userspace (in server-side script engines) or kernelspace application servers. This road is far from ideal, as it increases code complexity, memory- and CPU utilization, and reduces robustness.

An *I/O architecture* is the communication fabric linking applications, OS kernel and hardware, that extends from library interfaces in userspace down to peripheral devices. It crosscuts the classical OS layering. The present I/O architecture not only impedes performance on conventional hardware, it also obstructs the use of heterogeneous hardware. For high-speed network processing, i-Code proposes an I/O architecture for commodity operating systems that avoids common I/O bottlenecks and enables clean integration of arbitrary hardware.

On monolithic operating systems such as Linux, streaming I/O applications encounter one or more of the common bottlenecks presented in Figure 6.1. Transport overhead accrues where data is forwarded, at the crossings between hard- and software compartments. Computation issues can occur anywhere; these are the result of a poor match of application to available hardware. We now discuss the six bottlenecks.

1. **System Calls** Commonly, processes communicate with the kernel through system calls that require a mode-transition and copy operation for each block.

2. **IPC** System call overhead affects inter process communication (IPC) most. Traffic between applications is copied twice and per-call block size is constrained (often to 4 KB), causing frequent task-switching.

3. **Group Communication** Multiprocess access to the same data is seen in group communication (which subsumes 2-party IPC) and when auxiliary tasks such as traffic monitors are enabled. As in the IPC case, access to shared data requires a copy for each process and frequent task-switching.

4. **Kernel Subsystems** Between kernel subsystems copying is required when interfaces are incompatible. A classic example is having to copy between the disk cache and network queues while pinning of memory pages suffices in principle.

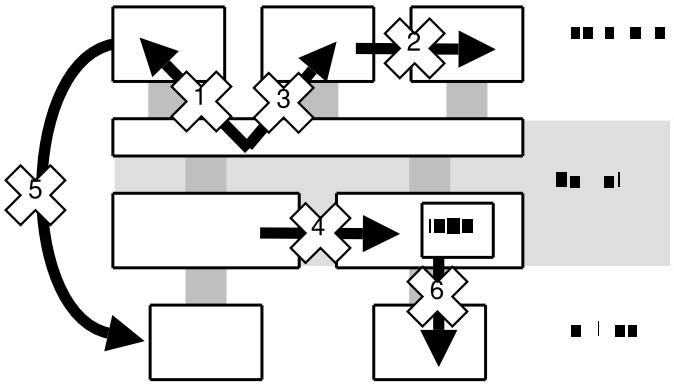

Figure 6.1: I/O Bottlenecks in a monolithic OS—the numbers are explained in the text.

5. **Direct I/O** Data traverses the kernel even when it performs no operation. High-speed devices (e.g., DAG cards [10]) present libraries that bypass this bottleneck, but these require superuser privileges and exclusive device access and they replace generic I/O primitives with vendor-specific APIs. An OS approach combines standard kernel control for resource multiplexing and device configuration with generic kernel bypass interfaces on the datapath.

6. **Fixed Logic** Applications sometimes encounter the above bottlenecks unnecessarily, because OSes force all to structure I/O logic the same way. A fileserver can save two copies by moving fast-path logic to the kernel; a DNS daemon can reduce latency by bypassing the kernel completely.

6.2 i-Code architecture

The i-Code I/O architecture avoids common bottlenecks by reconfiguring datapath logic at application load time to match workload and exploit special-purpose hardware. In general, the two extremes in coping with heterogeneous hardware and software configurations are (a) not to deal with it at all (static code), and (b) fully recompile all the code with specific optimizations for each specific configuration. We propose a mid-way point between static code and full recompilation, because both are impractical. The first cannot anticipate all computer architectures and applications. The second requires a single tool-chain capable of programming all available devices on each end host.

Instead, we construct application-tailored *I/O paths* at runtime from sets of precompiled processing and buffering elements. It avoids bottlenecks by

optimizing the mapping of applications onto the physical computer architecture: a *tailoring* algorithm selects the set of elements that (1) satisfies the application, (2) maximizes the use of specialized hardware and (3) minimizes data movement, in that order.

The two components of the I/O architecture that are crucial for performance are processing and buffering.

For processing, we reuse the well-known streams and filters model [26]. Streams and filters are the concepts behind the well-liked UNIX pipes. Unfortunately, the architecture behind pipes has so much overhead that pipes, for all their elegance, are not used in practice for high-speed applications. We refine them to avoid unnecessary cost from context switching, copying and cache misses. Specifically, it moves processing close to the data and minimizes data copying and task switching between stages.

Besides processing, efficient buffering management is crucial for I/O performance and will only grown in importance as long as memory latency keeps falling behind CPU speed increases. We propose a buffer management system where all live data is kept in coarse-grain ring buffers, and buffers are shared long-term between protection domains. Moreover, actual data transformation is replaced with updates to metadata structures, whereever possible.

In our design, we reuse proven interfaces where possible and base our system on Unix I/O [26]. We only deviate when functional or performance constraints demand it (for instance, we will see that we add a zero-copy `peek()` call to the POSIX file API to avoid unnecessary copying). Crucially, we follow the design principles of Unix: in particular, that "everything is a file," i.e., that all resources live in the same filepath namespace and expose the same file interface.

6.3 Buffering

A buffer management system controls the movement of data through a computer system. Its design determines the number of copies, data-cache misses and task switches per block. To maximize end-to-end throughput, these technical buffering details must be managed centrally, concealed from individual data clients, such as applications and filters.

6.3.1 POSIX File I/O

We explain the buffer management system design with a focus on copy -, context switch -, and cache miss avoidance. Crucially, we base our design on ring buffers.

As mentioned earlier, we choose the classic Unix file API for the I/O streams (`open()`, `read()`, etc.). The reason is that the file API is appro-

priate for sequential access typical for network intrusion systems and well known.

Traditionally, this interface is implemented as part of the ABI, but we will make it available in all address spaces and without a mode transition: all calls are local *function* calls that operate on locally accessible memory buffers (unless dictated otherwise by data access policy). Since POSIX `read()` has expensive copy semantics, we extend the interface with a `peek(int, char **, int)` call, a `read`-like function that does not copy. Specifically, with `peek()`, a client receives a direct pointer into the data stream.

Buffers are large contiguous memory regions capable of holding many blocks of data. To transport data across memory protection domains with minimal overhead, we share these regions among domains. Previous work has shown that modifying virtual memory mapping is cheaper than copying. We will increase these savings by reusing the same mappings for the duration of an I/O path.

6.3.2 Ring buffers

Traditionally, operating systems allocate blocks on-demand and use pointer queues to group blocks into streams. In contrast, we build static data rings, or *DBufs* from large memory regions. A *shared* ring has previously been shown to reduce copying cost between the kernel and userspace processes [8].

Static, shared rings hold a number of advantages over I/O based on dynamically allocated blocks: they amortize allocation and virtual memory management operations over the lifetime of streams and render sequential access cheap within streams because blocks are ordered in memory.

Data and index buffers The I/O architecture receives network packets in the circular DBufs. In addition, it places a pointer to this packet in a second circular buffer, known as the index buffer, or IBuf. Applications use the pointers in IBuf to find packets in DBuf.

The reason for using *two* buffers is that while an IBuf is specific to a flow, the DBuf is *shared*. If the application opens two streams, there will be just one DBuf and two IBufs. If the streams are "overlapping" (i.e., some packets in flow_a are also in flow_b), only one copy of each packet will be in DBuf. However, if a packet is in both flows, a pointer to it is placed in both IBufs. In other words, we do not copy packets to individual flows. Moreover, the buffers are memory mapped, so we do not copy between kernel and userspace either.

In addition, the indices in IBufs are small and easily fit in the cache. Pushing data from one filter to another is as simple as moving the Ibuf entry. There is no need to touch the data at all. This is good for cache behaviour.

6.3.3 Signal batching

Copying and cache usage are not the only factors with a large impact on performance. Signalling is another one. Our design specifically aims to reduce overhead due to context switching and signalling.

Signalling occurs when one filter in the architecture goes and tells another filter that there is new data available. For instance, when a component X has data for component Y, it places the data in a buffer that can be accessed by Y (in our case in a shared data buffer) and then signals the other party.

In traditional operating systems, this notification (or signalling) occurs for every data block that one filter sends to another filter. Doing so incurs a lot of overhead, due to context switches, TLB and cache pollution, etc.

In our design, we deliberately refrain from sending signals for every data block. Instead, signals are batched. The batching is flexible. One possibility is to use a procedure similar to the Linux NAPI patch for interaction with a network card, which transitions from signalling to polling as the load goes up. The difference in our architecture is that the signal batching permeates the entire architecture.

6.4 TCP/IP on top of shared rings

The processing and buffer management abstraction sketched above are ideal for fast I/O. Copying, context switching and signalling are avoided to boost performance. The file API provides convenient access to data buffers, and streams and filters allow one to build application conveniently. Unfortunately, it is not very suitable for legacy applications that build on sockets.

One of our goals is to support both novel and legacy network applications. To do so, our I/O architecture needs a full-fledged TCP/IP stack. TCP/IP stacks are hugely complicated. They deal with fragmentation, checksumming error control, and the whole set of complicated procedures to do with TCP's time-outs, retransmissions, window management, reassembly, slowstart, fast recovery, etc.

Mapping this complicated stack on a system of streams and filters on a shared ring buffer management system is no trivial task. The way to approach this is by building on the similarities between the file APIs and sockets, and using a well-tested, reliable network stack, rather than writing one from scratch. In our case, we plan to opt for a compact stack, such as that of LWIP.

CHAPTER 7

Console

The i-Code console will allow to see events generated by external sensors and collected within a centralized database. The i-Code console will provide the user with an easy and convenient way to browse, find, and analyze events; therefore the design of this console will be purposely simple and plain. The main screen will be composed by four parts: the header, the dashboard, the list of collected events and the footer. Figure 7.1 shows the main screen as it will appear after startup.

Header The header will allow to set and manage filters. Filters will be created by typing *filter tags*, that are key-value pairs (e.g., dst-ip = ...) into the text field—which will support autocompletion—placed at the upper right corner. Complex filters will be created by arranging the filter tags in rows and columns in the upper-left portion of the header. Filter tags in rows will be conjuncted while filter tags in columns will be disjuncted. This filtering approach will allow a good degree of flexibility: the user will be able to create, omit or move filter tags to compose the desired filter. Figure 7.2 shows an example of a filter, which can be read as "show all the events having destination IP equal to 123.056.123.042 *and* destination port equal to 80 *or* 443."

As described in Section 2.3, the console will have to satisfy some important requirements in order to make it a powerful tool to analyze security notifications and reports. First of all, the information provided by the system will be presented both in text (*Event List*) and graphical (*Dashboard*) forms; this will allow the user to both visually assess the overall situation and, if needed, go deeper using the details shown by the table. Moreover, the

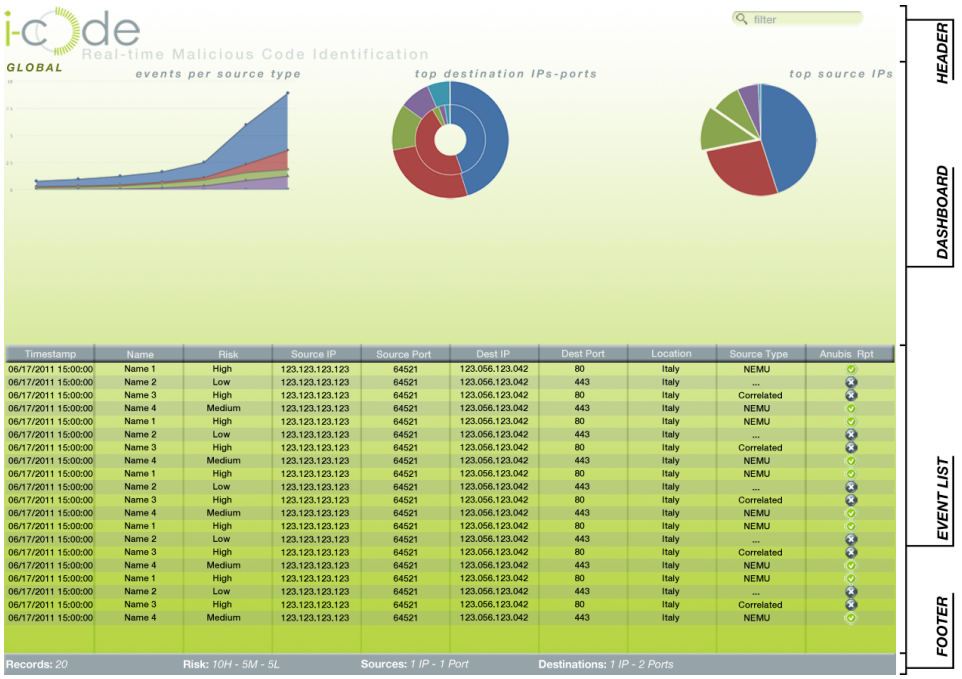

Figure 7.1: Mockup of the i-Code console's main screen, composed by four parts: header, dashboard, event list and footer.

Footer section can help by summarizing in a comprehensible and readable way what the user is being shown by the other sections.

Extendibility is another issue addressed by the presented mockup: the console will make use of a centralized database containing all the events gathered from the external sensors and it will be possible at any time to add new sensors and have their events shown in the *Event List* section. This addition will be completely transparent to the administrator, after configuring the newly added sensor. The user will also be able to discern what (type of) sensor sent the event by just looking at the *Source Type* column and, if needed, filtering by such property using the described filtering system.

Note that the above column will report events also from a particular kind of source, which will be a correlation engine. Correlated events will appear in the *Event List* just as any other events and will have the same properties of the events that triggered them. They will however be generated by customizable rules triggered by incoming events and dispatched to the main database by the correlation engine.

The i-Code will be built with all these guidelines in mind and will therefore be an invaluable aid for the administrators in locating and analyzing the security events that occurred within the network boundaries. It will in fact

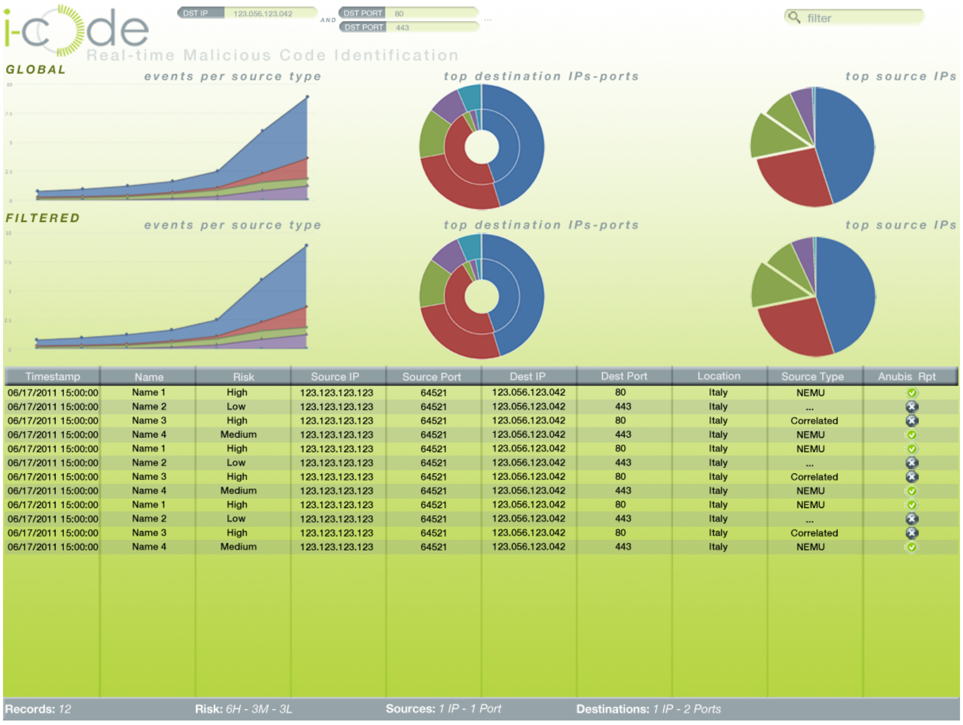

Figure 7.2: Mockup of the i-Code console showing the events matching a custom filter (in this dummy example, all the events having destination IP equal to 123.056.123.042 and destination port equal to 80 or 443). Note that a filter is created by arranging filter tags in the header section and, once set, the dashboard shows three additional graphs reflecting this smaller portion of data.

handle all the data management and visualization issues to provide the user a comfortable way to watch the events flow or filter them in any useful way, using simple tags and logical operators. It will also be developed using web based technologies that are compatible with most of the browsers commonly used (e.g., JavaScript, plain HTML) so that compatibility and dependency issues will not arise during later phases of the project.

CHAPTER 8

Conclusions

In this deliverable, we described the design for the i-Code console. By combining a variety of advanced detection and analysis techniques, the console will enable administrators to correlate events. Moreover, the design is deliberately kept extensible, so that new modules can be added in the future.

The design is based on integration and innovation. Each of the constituent components requires innovation: ambitious research in order to realise the design. Examples include a radical re-orientation of analysis techniques (from malware to shellcode), or an order-of-magnitude speedup for payload execution (for Nemu). The i-Code consortium is quite confident that these research goals will be achieved.

In addition, all these separate components will be explicitly integrated in a unifying console. Only the unification of the different techniques will help administrators to dig deeper into security incidents and respond to them in a timely fashion.

In the remainder of the iCode we will work towards realising the design described in this document.

Bibliography

[1] Anubis. 2007. http://anubis.seclab.tuwien.ac.at.

[2] Sans: Top cyber security risks, 2009.

[3] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of the 20th IFIP International Information Security Conference (IFIP/SEC)*, June 2005.

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

[5] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[6] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR) Annual Conference*, 2006.

[7] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, 2005.

[8] H. Bos, W. de Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis. FFPF: Fairly Fast Packet Filters. In *Proceedings of OSDI'04*, pages 347–363, San Francisco, CA, December 2004.

[9] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA, May 8–11, 2005. IEEE Computer Society.

[10] J. Cleary, S. Donnelly, I. Graham, A. McGregor, and M. Pearson. Design principles for accurate passive measurement. In *Proceedings of PAM*, Hamilton, New Zealand, Apr. 2000.

[11] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. 2010.

[12] W. de Bruijn, H. Bos, and H. Bal. Application-tailored i/o with streamline. *ACM Transacations on Computer Systems (TOCS))*., 29:6:1–6:33, May 2011.

[13] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional, 2005.

[14] D. Kang, D. Fuller, and V. Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *6th IEEE Systems Man and Cybernetics Information Assurance Workshop (IAW)*, 2005.

[15] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, Vancouver, BC, Canada, August 2006.

[16] C. Kolbitsch, P. Milani, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium (Security'09)*, pages 351–366, Montréal, Canada, Aug. 2009. USENIX Association.

[17] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Symp. on Recent Advances in Intrusion Detection (RAID)*, 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 ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 53–64, 2006.

[19] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell. A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection (RAID'08)*, pages 78–97, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] 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.

[21] S. Mukkamala, A. Sung, D. Xu, and P. Chavez. Static analyzer for vicious executables (SAVE). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 326–334, Tucson, AZ, USA, Dec. 2004.

[22] J. Nazario. PhoneyC: A Virtual Client Honeypot. 2009.

[23] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, December 2010.

[24] 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.

[25] G. Portokalidis, A. Slowinska, and H. Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, Leuven, Belgium, April 2006.

[26] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[27] C. Seifert and R. Steenson. Capture-HPC. https://projects.honeynet.org/capture-hpc, 2008.

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

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

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

[31] 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.

[32] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *In Proceedings of the European Workshop on System Security (EuroSec)*, 2011.

[33] H. J. Wang, H. J. Wang, C. Guo, C. Guo, D. R. Simon, D. R. Simon, A. Zugenmaier, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *In ACM SIGCOMM*, pages 193–204, 2004.

[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.