POLITECNICO
MILANO 1863

Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Corso di laurea Magistrale in Ingegneria Informatica

SysTaint: Assisting Reversing of Malicious Network Communications

Tesi di laurea di:
Gabriele Viglianisi
Matr. 836130

Relatore:
Prof. Stefano Zanero

Correlatore:
Dr. Andrea Continella

A.A. 2016-2017

Contents

List of Figures

List of Tables

Listings

Abstract

Malware analysis is the complex practice of detecting and studying malicious software samples to develop countermeasures and protect end users and companies from new threats. Due to the size of the criminal industry profiting from malware there is a large volume of new malware samples appearing on the Internet every day. To be able to efficiently analyze this large amount of samples, we need to employ automatic analysis methods.

In this work, we present SysTaint, a new semi-automated analysis tool to help malware researchers rapidly obtaining information on the internal functions of a software sample that are related to a given observed behavior. In particular, we address the specific challenges in studying malware whose behavior relies on communicating with servers under the control of malicious actors.

We implemented SysTaint on top of the PANDA analysis platform, and we tested it by employing it to study four banking Trojan samples, successfully bypassing the use of encryption and locating the malware code processing the data being sent through the network.

Sommario

L'analisi dei malware è una pratica complessa, che consiste nell'acquisire ed analizzare un campione ignoto di software per determinare se esibisca comportamenti malevoli, capire meglio il modo in cui funziona, e sviluppare contromisure.

Nel corso dell'ultimo decennio una intera industria criminale si è sviluppata attorno ai malware, e toolkit che permettono a chiunque di condurre profittevoli campagne malware (per esempio ransomware, spam, frode ai danni degli inserzionisti online) vengono venduti sul mercato nero. Come risultato, nuove specie di malware appaiono su internet in grandi volumi. Secondo il rapporto del quarto semestre 2017 sulle minacce informatiche [24] di McAfee Labs, 57,6 milioni di nuovi campioni sono stati osservati nel terzo semestre del 2017.

Per tenere il passo con la comparsa di nuovi malware, i ricercatori nel campo della sicurezza informatica hanno sviluppato strumenti che permettono di analizzare grandi volumi di nuovi campioni.

Un campione di malware può essere studiato staticamente, analizzandone il codice, oppure dinamicamente, eseguendolo in un ambiente controllato ed osservando il suo comportamento. I malware, a loro volta impiegano diverse tecniche, mirate a impedire specificamente l'uso di questi due approcci, per evitare l'analisi, in modo da non essere rilevati e rallentare il lavoro dei ricercatori.

Soluzioni automatizzate come i programmi antivirus possono riconoscere automaticamente se un campione è un malware noto, cercando specifici pattern nel file del campione, oppure eseguendolo e monitorando le sue attività. L'analisi automatizzata dei malware è anche impiegata in laboratorio per aiutare l'analista nello studio di malware appartenenti a nuove famiglie. Questo tipo di analisi è comunemente eseguita tramite *programmi sandbox*, che raccolgono automaticamente dati sulle interazioni tra il campione ed il sistema operativo e ne evidenziano i pattern malevoli. C'è ancora, però, un bisogno pratico di studiare e fare ingegneria inversa sui campione di malware per poterne capire la logica interna, permettendo all'analista di contrastare le moderne tecniche con cui i malware evitano l'analisi, e individuare nuovi comportamenti malevoli. Nonostante la comunità di ricercatori abbia impiegato molto lavoro nello sviluppare tool che aiutino l'analista nel fare ingegneria inversa, questo compito rimane difficile ed esoso in termini di tempo.

Sono particolarmente difficili da studiare i malware il cui comportamento comporta e dipende dal comunicare tramite la rete con un *server di comando e controllo (C&C)*, un server sotto il controllo di attori malevoli che viene usato per raccogliere dati sottratti agli utenti e controllare un insieme di computer infetti (*botnet*) con lo scopo di usarli per effettuare frodi, diffondere altro malware tramite spam, o effettuare attacchi di *denial of service*.

In questo lavoro proponiamo un tool semi-automatico, SysTaint, che assiste gli analisti di malware eseguendo il campione in un ambiente controllato, registrando dati approfonditi sulla sua esecuzione, e dando all'analista un modo di interrogare ed esplorare questi dati, per approfondire il funzionamento del malware. Con questo approccio miriamo di superare molti degli ostacoli che le soluzioni odierne incontrano nello studiare i malware che si basano sulla comunicazione di rete.

Abbiamo implementato il nostro tool su PANDA, una piattaforma di analisi che ci permette di registrare l'esecuzione del campione, per poi farne il replay in modo deterministico e analizzandone l'esecuzione. SysTaint monitora i dati scambiati tra il campione ed il sistema operativo e ne traccia il flusso attraverso le funzioni interne del campione, registrando le funzioni coinvolte ed evidenziando automaticamente le funzioni di crittografiche. L'analista può poi esplorare i dati raccolti per individuare le funzioni ed il relativo codice coinvolto in attività specifiche. Per esempio l'analista può facilmente trovare la provenienza dei dati inviati, cifrati, attraverso la rete, localizzando i corrispondenti dati non cifrati e le funzioni che li processano.

Infine abbiamo integrato SysTaint con Cuckoo Sandbox, una diffusa soluzione sandbox open source, per rendere facile impiegare SysTaint in ambienti di analisi esistenti e per sfruttare le funzionalità già fornite da Cuckoo Sandbox.

Abbiamo testato sperimentalmente il nostro approccio applicandolo allo studio di quattro campioni di banking trojan il cui comportamento dipende dalla comunicazione con un server di comando e controllo. Studiando con SysTaint ognuno dei campioni abbiamo potuto esplorarne l'esecuzione sfruttando le informazioni sui flussi di dati e bypassare l'uso della crittografia nelle comunicazioni, ricavando informazioni utili sulle funzioni interne del malware.

La rimanente parte di questo documento è organizzata come segue:

Nel capitolo 2 introduciamo alcuni concetti necessari alla discussione dello stato dell'arte e del nostro approccio. Nel capitolo 3 introduciamo il problema dell'analisi dei malware e trattiamo l'attuale stato dell'arte e le motivazioni del nostro lavoro. Nel capitolo 4 discutiamo l'approccio proposto, poi nel capitolo 5 la sua implementazione. Nel capitolo 6 riportiamo la valutazione sperimentale della nostra implementazione, il modo in cui i campioni sono stati selezionati, e come ognuno è stato analizzato per ottenere informazioni sulle sue comunicazioni di rete. Nel capitolo 7 discutiamo le limitazioni tecniche dei questo strumento, e i limiti alla sua applicabilità. Nel capitolo 8 proponiamo alcuni possibili miglioramenti dell'approccio in lavori futuri per estendere le capacità di SysTaint e renderlo più efficace. Infine, nel capitolo 9 concludiamo la tesi con delle considerazioni sul nostro lavoro.

Chapter 1

Introduction

Malware analysis is a complex process that consists in acquiring and analyzing a given unknown software sample to determine whether it exhibits malicious behaviors, better understand the way it works, and develop countermeasures.

In the course of the last decade, a whole criminal industry developed around malware, with toolkits that allow criminals to easily run profitable malicious campaigns (e.g., ransomware, spam, ad-fraud, or denial of service) being sold on the black market. As a result, new malicious specimens keep appearing on the Internet at high rates. According to McAfee Labs' Q4 2017 Threat Report [24] 57.6 million new samples have been observed in Q3 2017.

To keep up with the volume of new malware, security researchers developed scalable automated tools to analyze large amounts of previously unseen malware samples.

A malware sample can either be studied statically, by analyzing its code, or dynamically, by running it in a controlled environment and observing its behavior. Malware, in turn, employs several techniques – aimed to specifically counter the use of these two approaches – to evade the analysis, in order to remain undetected or hinder the work of security researchers.

Automated solutions such as antivirus software can automatically recognize if a sample is a known malware by looking for specific patterns in the sample's file, or by executing it and monitoring its activities. Automated malware analysis is also employed in a laboratory environment to help the analyst in studying previously unknown malware. This kind of analysis is commonly performed through *sandbox software*, which automatically collects data about the interactions of the sample with the operating system and highlight malicious patterns. There is still, however, a practical need to manually study and reverse engineer malware samples in order to understand their internal logic, allowing the analysts to deal with modern evasion techniques and spot new malicious behaviors. Although the research community put a lot of efforts into developing static and dynamic analysis tools helping the analyst in reverse engineering, such task remains hard and time-consuming.

Particularly difficult to study are the malware whose behavior involves and relies on communicating via the network with a *command and control server (C&C)*, a server under control of malicious actors that is used to collect stolen user data and control a set of infected computers (*botnet*) with the purpose of using them to perform frauds, spread malware, or perform denial of service attacks.

In this work we propose a semi-automated tool, SysTaint, which assists malware analysts by running the analyzed sample in a controlled environment, registering in-depth data of its execution, and providing a way to query and explore this data in order to dig into the details of the sample behavior. With this approach we aim to address many of the obstacles faced by the existing dynamic analysis solutions when studying malware that rely on network communications.

We implemented our tool on top of PANDA, an analysis platform that allows us to record the sample execution, replay it deterministically, and study in details the execution during the replay phase. SysTaint monitors the data exchanged between the sample and the operating system and tracks their flow through the internal functions of the sample, logging the functions involved and automatically detecting cryptographic functions. The analyst can then explore the collected data to find functions and the related code involved in specific activities. For example, the analyst can easily find the provenance of the data being sent encrypted through the network, locating the corresponding unencrypted data and the functions processing it.

Finally, we integrated SysTaint with Cuckoo Sandbox, a widespread open source sandbox solution, in order to make it easy to deploy SysTaint in existing analysis environments and to leverage the functionalities Cuckoo Sandbox already provides.

We experimentally tested our approach by applying it to study four samples of banking Trojans whose behavior relies upon exchanging data with a command and control server. Studying each sample with SysTaint we could explore its execution making use of the data flow information and bypass the use of encryption in the network communications, obtaining useful information on the data being sent and the inner functions of the malware.

The remainder of this document is organized as follows:

In chapter 2 we introduce some concepts necessary to our discussion of the state of the art and our approach. In chapter 3 we introduce the problem of malware analysis and present some background concepts that are central to understanding the state of the art. In chapter 4 we discuss our proposed approach, then in chapter 5 its implementation. In chapter 6 we detail how we tested our implementation, how the malware samples have been collected and selected, and how each is analyzed in order to gain information on its network communications. In chapter 7 we discuss the technical limitations of the tool, and the limits to its applicability. In chapter 8 we propose some possible improvements to the approach for future works to extend SysTaint's capabilities and make it more effective. Finally, in chapter 9 we conclude the thesis with final considerations on our work.

Chapter 2

Background

In this section we briefly introduce a few topics that are central to malware analysis. For a more complete introduction to these topics refer to the book [13].

2.1 Debugging

Debugging is a technique allowing one program (the debugger) to control the execution of another program. It allows an analyst to closely follow a program's execution by pausing it and allowing the program's internal state to be inspected. It is possible to pause the program being debugged either by using a specific API that puts the processor in *single-step mode*, causing the program to pause after every instruction it executes, or by setting *breakpoints* at specific addresses, which cause the execution to pause when a certain instruction is reached, or when a given memory location is accessed.

Even though debugging an unknown and complex process is a long and tiring task, which requires re-executing a program multiple times, debuggers are invaluable in studying and understanding the behavior of a program, as they allow observing when the execution reaches a given piece of code, and how this code behaves with the current program state and input data.

2.2 Functions and call stacks

Functions, or procedures, are blocks of code that are executed with specified inputs (arguments) and return an output value, optionally modifying the internal state of the process or interacting with the underlying operating system.

An efficient way of reverse-engineering a program consists in identifying the logic and purpose of the individual functions, counting on the fact they represent a single functionality that is reused throughout the program, to gradually build an understanding of the program as a whole.

In programming languages that compile to native machine code, such as C, C++ and Rust, the compiler translates each function to a block of consecutive assembly instructions. When called, functions keep their variables in a memory region called "stack", organized like the homonym data structure. Functions expect their arguments to be in memory at a given offset from the address in the *stack pointer* or *base pointer* register, according to the *calling convention* of the compiler and processor architecture (ABI). When a function is called, the caller pushes its arguments and the return address (usually the address of the next instruction) on the stack, then jumps to the address of the called function. After that, the called function sets up its *stack frame*, a portion of the stack that will contain the function's variables, on the top of the stack frame of the calling function. When the called function returns, its stack frame is removed from the stack, and the execution jumps back to the caller, whose address is taken from the top of the stack. This mechanic is referred to as *call stack*.

The call stack can be inspected to determine which functions are running in a given moment and the values of their variables and arguments. A debugger or another program inspecting the program memory can use two different methods to understand the contents of the call stack. One relies on scanning the memory for a location containing the value of the caller's base pointer, which is unfortunately only generated by some compilers. Another relies on keeping track of each function call, in real time, and maintaining a separate data structure called *shadow stack*, which stores the addresses of each called function as well as the boundaries of each function call's stack frames. It is important to note that each thread has its own separate call stack.

As an optimization, modern compilers can replace a function call with a copy of the called function body if this is particularly short. This process is called *inlining*, and it used to avoid, during execution, the cost of the function call.

2.3 Cryptography

Cryptography is a branch of mathematics and computer science studying how to achieve secure communication between two or more parties. It allows, by using specific algorithms, encoding a message in a way that only the intended recipient can access it (*encryption*), computing practically unique identifiers associated with some data (*hashing*), and verifying that some data comes from a known party and has not been tampered with (*digital signature*). Unencrypted and encrypted data are also called respectively *plaintext* and *ciphertext*.

The majority of the communications over the Internet are nowadays encrypted to protect the privacy and security of users. Modern malware communicating with the network also employ cryptography to protect their data and communication protocols from analysis and tampering.

2.4 Operating System concepts

Virtual memory Modern operating systems make use of a processor feature called virtual memory to provide each process with his own address space, isolating its memory from the one belonging to the other processes. Before the processor switches to executing a different process, the operating system sets the corresponding value of a specific register (e.g., `CR3` in x86), so that the address space of the right process is used.

The kernel resides in a fixed part of each process' address space called kernel-space. Only the code running from kernel-space has permissions to access the kernel-space memory. User processes, running in user-space, need to interact with the operating system kernel through system calls, in order to access any resource outside of their address space.

System calls System calls (*syscalls* from here on for brevity) are a mechanism of modern operating systems that allows user programs to call specific kernel functions. A process performs a syscall by preparing the arguments in memory and specific registers, and then, unlike regular function calls, by executing a specific instruction that gives the control to the kernel. Syscalls are necessary for a process to interact with the world outside of its own address space.

Syscalls are abstracted into regular function calls by several layers of system libraries, and as such are never directly performed in common program's code. Malware programs however may, as an obfuscation measure, call directly into the low level syscall wrappers or perform the syscall directly.

Syscalls are interesting from a malware analyst point of view, because all processes need to go through them to perform any meaningful task. Save for rare malware infecting the kernel, the behavior of a syscall cannot be altered. Logging the executed syscalls is thus an effective way to capture the interactions between a process and the rest of the system.

2.5 QEMU

QEMU [1] is a widely used open source machine emulator and virtualizer. It is commonly employed as a hypervisor to run whole operating system in a *virtual machine*, or *VM* in short, comprised of a virtual CPU, RAM, and several other devices.

QEMU was originally designed to run programs built for an architecture that differs from that of the host, by translating on the fly each *basic block* of code (a series of instructions executed sequentially) executing on the virtual CPU. The translation is carried out by a component called *Tiny Code Generator* or TCG. QEMU's capability of translating code on the fly makes it an effective platform to perform program analysis tasks through dynamic instrumentation.

It is worth noting that QEMU can also take advantage of the hardware virtualization capabilities of modern processors, which allow a hypervisor like

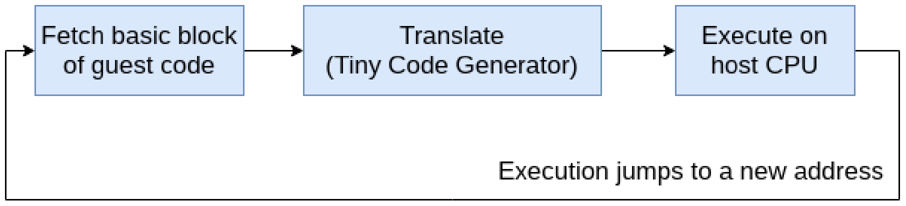

Fetch basic block of guest code
Translate (Tiny Code Generator)
Execute on host CPU
Execution jumps to a new address

Figure 2.1: In TCG mode, QEMU's fetch-translate-execute loop allows QEMU's virtual CPU to execute code compiled for a different processor architecture

QEMU to directly respond to interrupts and syscalls, without requiring on-the-fly code translation and thus running virtualized code at almost native speed on the host's CPU. However, when used in such fashion, QEMU cannot be used to add instrumentation to the code or closely monitor the code execution.

2.6 Command and control servers and botnets

Malicious actors spread malware to perform several kinds of activities, referred to as *malware campaigns*. Some common kinds of campaigns are ransomware, denial of service, ad-fraud, spam, phishing, and credit card fraud.

To perform these activities, the computers under control of a malware (*bots*) are usually organized in complex infrastructures called *botnets*. An important part of botnet infrastructures are *command and control servers* (C&C in short), central servers under control of malicious actors, which a set of infected computers contacts in order to report on its work and get tasks to execute. For example, a malware sample whose purpose is sending spam emails may contact the C&C server to retrieve a list of target mail addresses, likewise some multipurpose Trojans may ask the C&C about which malicious tasks they should perform.

C&Cs are a fundamental part of malware infrastructure, and an important target for law enforcement, as locating and shutting them down is an effective way of disabling the whole botnet. Since C&Cs are high value targets, malicious actors take particular care in protecting them, often hiding them behind hacked servers used as proxies. The DNS addresses that the bots use to contact the C&C are often generated by domain-generation-algorithms, to keep the botnet working even when a *domain name registrar* deactivates the domain name. Bots belonging to a small set of malware families are also able to organize themselves in peer-to-peer networks, in a further attempt to hide and protect the original C&C.

To make the analysis as difficult as possible, the communications with the C&C servers are usually encrypted, employing both ad hoc encryption protocols and widespread solutions like HTTPS and TOR.

2.7 Summary of common anti-analysis techniques

Malware employs a wide range of techniques to stay hidden from automatic detection and hinder the work of malware analysts. Both analysts and automated analysis solutions need to be aware of these techniques to carry out their analysis. These techniques are:

Packing Packing is a technique that protects an executable program from static analysis by making it hard for the analyst to access the malware code without running it. Packing consists in compressing and encrypting the original executable, then distributing it inside of another program called *unpacker*, whose only task is restoring the original program in memory and running it.

Some advanced packing techniques involve selectively unpacking the sections of code that are about to be executed, so that the original code is never available in memory as a whole.

It is worth noting that packing is also commonly used, together with obfuscation, in commercial programs, in efforts to protect the intellectual property.

Obfuscation Obfuscation consists in transforming the executable code to hide its purpose from an analyst reading it, while maintaining its function when executed. Obfuscation is also used to impede static analysis, by purposely violating some assumptions that static analysis tools commonly make, about some specific sequences of instructions.

A widespread and basic form of obfuscation consists in hiding the usage of functions belonging to the system libraries, by indirectly calling these functions from references obtained beforehand.

Injection techniques Malware programs execute in their own process for a short time, then they usually immediately load themselves into another process' memory via process injection. Doing so allows them to hide, but also to alter the behavior of a host program. Sandbox software need to detect these techniques automatically and in real time to know which processes to analyze. Since, unfortunately, there is a wide variety of injection techniques, some more stealthy than others, this task is not always easy.

Anti-debugging measures The use of a debugger is fundamental for studying unknown programs. Some malware samples attempt to detect when they are being debugged by looking for specific debugger artifacts in their memory and purposefully crash to avoid the analysis if this is the case.

Similar checks are also employed against some dynamic analysis techniques and tracing frameworks such as Intel Pin.

Virtual machine detection Because the substantial majority of malware analysis (be it automated or manual) is performed inside of virtual machines for both convenience and hygienic reasons, some malware may include logic to

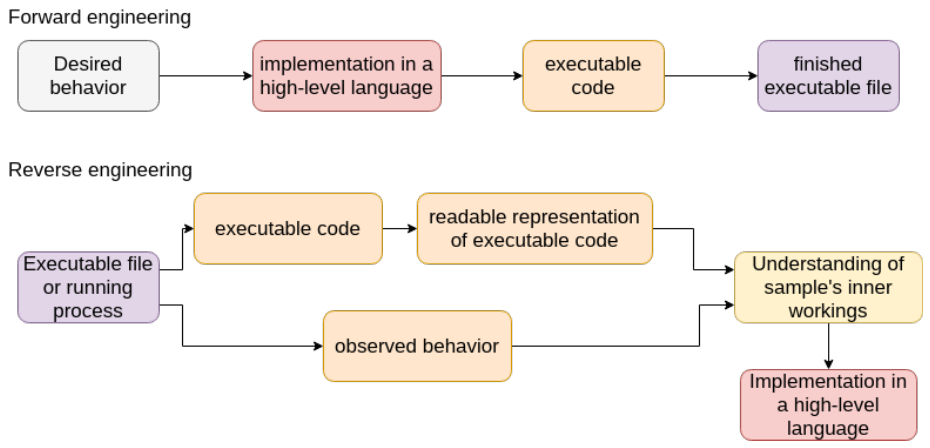

Forward engineering
Desired behavior
implementation in a high-level language
executable code
finished executable file
Reverse engineering
Executable file or running process
executable code
readable representation of executable code
observed behavior
Understanding of sample's inner workings
Implementation in a high-level language

Figure 2.2: Forward and reverse engineering

detect if they are run inside of a virtual machine. These checks usually look at the behavior of specific CPU instructions, timings, or specific virtual devices.

2.8 Reverse engineering

Reverse engineering is the process of studying the behavior and executable code of a program in an attempt to reconstruct the original high-level human-readable code and its purpose.

Although reverse engineering is a hard and mostly manual process, requiring the analyst to spend substantial amounts of time, it ultimately allows understanding the code being executed, which represents the single source of truth about a program's inner workings and logic, and capabilities. To perform reverse engineering on a program, and reconstruct the semantic of its code, an analyst usually takes advantage of both static and dynamic analysis approaches, examining the program both at rest and during the execution.

Some programs employ anti-reverse-engineering and anti-analysis techniques, as discussed in section 2.7.

2.8.1 Tools commonly used in the reverse engineering process

There is a number of tools to help the analyst in several parts of the reverse engineering process. Because of the nature of the tasks, most of these tools are either interactive or distributed as plugins for interactive reverse engineering environments like *IDA Pro* or *Radare2*.

Some of the tasks carried out by existing reverse engineering support tools (with different degrees of effectiveness) are:

Disassembly interpreting portions of a binary file as sequences of machine instruction that can be analyzed or displayed

De-compiling representing low-level assembly instructions with constructs of higher level languages like C, for example functions, variables, conditionals, and loops

Function boundaries detection recognizing functions in the disassembled code, their boundaries, stack frames and prototypes

Function identification recognizing if a function included in a program is an instance of a well-known one

Cryptographic function identification locating and/or recognizing cryptographic functions inside of a binary program

Unpacking recovering the original, unpacked binary from a sample protected by packing, so that it can be statically analyzed

De-obfuscation recovering the pseudocode describing the original functionality of a piece of code from an obfuscated sequence of instructions

String decryption recovering a list of strings that are included in a program in obfuscated form

2.9 Dynamic instrumentation

The dynamic instrumentation techniques consist in transforming the program code, inserting additional instructions that, when the code is executed, allow monitoring various aspects of the execution. It can be used for various purposes, like tracing the execution of certain functions, countering evasion techniques and performing more complex analyses like taint tracking and cryptographic function detection.

There are several ways to dynamically instrument a program, varying in speed, capabilities, and ease of deployment:

2.9.1 Virtual-machine based

A first kind of dynamic instrumentation consists in executing the code in a virtual machine and modifying it before it is executed, in a just-in-time (JIT) manner.

Two kinds of virtual machine can be employed for dynamic instrumentation: *process virtual machines*, running a given program in a virtual CPU, but giving it access to the host operating system, or *system virtual machines*, also known as *hypervisors* running a whole operating system emulating the virtual hardware.

Examples of tools based on *process virtual machines* are *Intel Pin* [2] and *Valgrind* [5]. Both these tools are used as base for more targeted analysis frameworks like *Angr* [22].

Hypervisor-based instrumentation tools are mostly based on the open source QEMU machine emulator used in TCG mode, leveraging its ability to apply additional transformations to the guest's machine code before it is executed on the host's CPU. Examples of such tools are the *Bitblaze* [8] framework, the *S2E framework* [11], and the *PANDA* [19] reverse engineering platform.

2.9.2 Function hooking

Another widespread way to instrument a program is by intercepting, through a method called *hooking*, the calls to external library functions it makes. Hooking consists in redirecting the original call to a custom handler, which usually perform some additional actions before returning to the original code. Hooking techniques are widespread in program analysis but are also widely employed by malware to alter the behavior of other programs, for instance a browser or the file manager.

Some common analysis tools employing function hooking are *Frida*, *Deviare*, *Microsoft Detours*, and the *Cuckoo Monitor*. By making use of function hooking, these tools can intercept high-level interactions with the external libraries, log them and alter their behavior.

The downside of using function hooks is that they are easy to detect for both malware and analysis tools. Moreover, a malware can avoid calling library functions altogether, by directly using lower level system functions or syscalls.

2.9.3 Kernel drivers

Kernel drivers are special programs that are executed in kernel-space and can alter some kernel functionalities. They are a powerful tool in the hands of analysts since they have full access to the operating system and are impossible for common user-space programs and malware to detect.

Kernel drivers can be used to hook system calls invocations, but not calls to high-level library functions. On Windows this is achieved by overwriting the entries in the kernel's *system service descriptor table*. Altering the behavior of syscalls via a kernel driver can be useful to hide other parts of the analysis environment from the malware samples.

An example of such tools is *zer0m0n*[34], which is a kernel driver aiming to be a kernel-space replacement of the *Cuckoo Monitor* component of *Cuckoo Sandbox*.

2.9.4 Taint analysis

Taint analysis, *dynamic tainting* or *taint tracking* is a technique, implemented via dynamic instrumentation that allows tracking how a given input influences the execution of a program. It consists in marking (*tainting*), some data D with a label, then monitoring the executed instructions to propagate the taint label to all the data obtained from D.

Taint analysis is often employed in security testing, in order to figure out which parts of the execution an attacker can influence.

2.10 Memory analysis

Memory analysis is an approach consisting in acquiring and analyzing a copy of a computer's RAM contents. This technique is used in forensics tasks where from a copy of the computer memory the analyst can reconstruct which programs and data were recently loaded. It is possible to use memory analysis to detect signs of infection and obtain the malware code after it is unpacked in memory.

Common memory analysis tools are *Volatility* [31] and *Rekall* [30].

Chapter 3

Motivation

3.1 Problem Statement

Automatically analyzing malware has become a practical necessity for security firms, which need to determine the behavior, nature, and family of a large number of malware samples to protect companies and end users.

The security industry has developed highly sophisticated solutions to automatically detect if a given software sample is malicious, both in laboratory environment, and as *antivirus software* to be deployed on the end user's systems. Similar solutions are employed to study previously unknown malware threats in a laboratory environment, with tooling that provides the researchers with information about the code and behavior of the malware, so that he can perform an accurate analysis of the threat and develop effective countermeasures.

Automated analysis solutions employ a variety of techniques that can be classified in static and dynamic. While static analysis techniques extract information by analyzing the code of the sample, dynamic analysis techniques consist in running the sample in a controlled environment to gather information on the behavior and the workings of a sample that are difficult to extract otherwise.

The most common dynamic analysis solutions are sandbox software, which are designed to process a large number of samples submitted in batches. Sandbox software run each sample in a virtual machine for a given amount of time, tracking and logging the interactions of the sample with the operating system and known system libraries.

Logging the interactions between the sample and the operating system, however, does not fully capture the behavior, logic, and capabilities of the sample. For example, the communications with external C&C servers are usually encrypted and are unintelligible from the logs a sandbox normally captures. Even when the sample does not employ encryption, it may be useful to understand how exactly certain data the malware retrieves from the system is processed and used by the sample's internal logic. To understand these behaviors and in general the inner workings of the sample, the information that can be extracted from a normal sandbox analysis are insufficient, and it is necessary to obtain more in-depth information, usually by means of manual debugging sessions.

Unfortunately, debugging session require the analysts a significant amount time, and some behaviors are particularly difficult to replicate and study by means of debugging. An example is again the communication with Internet resources and command and control servers, whose timing and outcomes can vary depending on the server status and the time of the analysis. The same factors can also create problems to other, less common, heavyweight dynamic analysis techniques, which need to pause or significantly slow down the execution of the sample, possibly causing the network communications to fail.

The study of the communications with the command and control server is a particularly important goal for malware analysts, since understanding the communication protocol may allow the analyst to obtain useful information about the botnets.

3.2 State of the art

3.2.1 Record-replay solutions

Record-replay solutions allow recording the execution of a program so that it can be analyzed at a later time. They are usually employed together with debuggers, allowing the analyst to skip forward and backward in time as he sees fit, inspect the internal state in any point in time, and to study behaviors that are difficult to replicate such as network communications and those carried out by multiple threads.

Record-replay solutions like *QIRA* [20] are expensive in terms of memory and disk space, and efficient record-replay solutions are relatively recent, and still limited to some scenarios, environments or kinds of programs. Due to their usefulness, however, their adoption is increasing. Examples of record-replay solutions are *QIRA*, *Mozilla RR* [26], *WinDBG* [25] and *PANDA*, which we describe in details later.

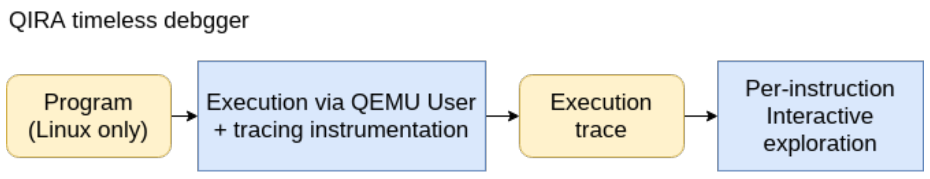

QIRA timeless debgger
Program (Linux only)
Execution via QEMU User + tracing instrumentation
Execution trace
Per-instruction Interactive exploration

Figure 3.1: High level overview of QIRA. QIRA makes use of QEMU in user-mode to instrument the execution of a Linux program. QIRA collects a detailed, per-instruction trace, which can be then indexed and interactively queried via a graphical user interface

3.2.2 Automated protocol reverse engineering

The literature contains several works addressing the problem of *automated protocol reverse engineering*, consisting in determining the specifics and the contents of an unknown communication protocol by observing it. A complete survey of these works can be found in [28] and [23].

The approaches in the literature can be broadly categorized in two groups: those working with a corpus of captured network packets, and those making use of program analysis.

We focus on this second group because, since it does not require a corpus of plaintext messages exchanged between the two parties, it is best suited for studying the communications of malware. In particular, this second approach works in the presence of encrypted network communications, which are widely used in newer malware samples.

3.2.3 Automated reverse engineering of encrypted network communications

In [7] Lutz presented a technique for automatically retrieving the unencrypted version of encrypted network communications by employing dynamic program instrumentation. His approach consists in determining when and where a buffer with decrypted data was present in the program's memory. Lutz employed various heuristics in order to locate the decryption functions that read the encrypted input, using taint analysis to reduce the number of candidate functions.

Lin et al. [6] presented a way of performing automatic protocol reverse engineering by means of dynamic program instrumentation. By observing the way a program processes the message, their approach can automatically identify the various fields of the protocol and the hierarchical relation among them. To achieve this, the authors employed taint analysis to track the transformation of each byte in the received buffer, and call stack analysis to determine the context in which each byte was processed. An algorithm is then applied to use this data to build and refine a protocol field tree.

Wang et al. [10] proposed a way of reverse engineering encrypted messages. The first step of their approach consists in finding the plaintext buffer by continuously monitoring the ratio of arithmetic operations executed, identifying a boundary between a decryption phase and a usage phase. Once the buffer with plaintext data has been detected, the format of messages is extracted with the tool described in [6].

Caballero et al. [9] proposed an approach to automate network protocol reverse engineering and allow for botnet infiltration. Their approach also allowed, in contrast to the previous works, studying sent messages by means of a technique called *buffer deconstruction*, making use of dynamic slicing[1]. Dynamic tainting was also employed to associate the fields in the deconstructed buffer

[1]Dynamic slicing is a techniques consisting in obtaining a log of all the instructions executed by a program, together with their operands, then reading it in the reverse order, to identify all the instructions involved in generating a given piece of data

with selected API calls to infer their semantic. Their technique also employs cryptographic function detection, with a simple heuristic considering the ratio of arithmetic instructions.

3.2.4 Cryptographic function detection

In addition to the aforementioned works [7, 10, 9] in the context of protocol reverse engineering, some publications expanded on the problem of detecting cryptographic functions in a program.

Gröbert [12] summarizes various static and dynamic methods, implements and tests the effectiveness and performances of the previous approaches as well as signature-based methods and some newly proposed heuristics.

Li et al. [16] propose a method to detect plaintext buffers in applications that immediately re-encrypt the plaintext buffer after use. Their proposed approach consists in detecting the avalanche effect of encryption functions.

Wang et al. [15] propose an approach to automatically break DRM implementations, allowing the analyst to retrieve the unencrypted media file. They achieve this goal by detecting the decryption functions by means of statistical analysis on the entropy of the written data in order to distinguish encrypted and compressed data.

3.2.5 Automated malware analysis

Automated malware analysis techniques allow researchers to study malware at scale, performing the analysis of a large number of samples and producing reports, automatically highlighting possible threat indicators. Automated malware analysis usually employs both static and dynamic analysis methods.

Sandbox software Automatic dynamic or behavioral analysis is carried out by means of *sandbox software*, which run each sample inside of a virtual environment, collecting data about the sample's activities, and highlighting those that are malicious or suspect. Changes to important system files or configurations and deletion of user files, for examples, are flagged as malicious behaviors. This approach was popularized by CwSandbox [4], and is now widely used in the industry, with several commercial sandbox services available.

Most of the data collected by sandbox software comes from system call tracing and from hooking known Windows API functions. The log of all the traced system and API calls is then made available to the analyst as part of the produced report, together with a log of the network activity.

Cuckoo Sandbox [29], also called *Cuckoo* in short is, at the time of writing, the most advanced and widespread open source sandbox solution. It can be easily configured and customized through modules written in the Python programming language, which control the various phases of the analysis. It makes use of an in-system component called *Cuckoo Monitor*, which is tasked to monitor the sample execution by means of function hooking. A repository of community-contributed analysis modules and signatures is also available.

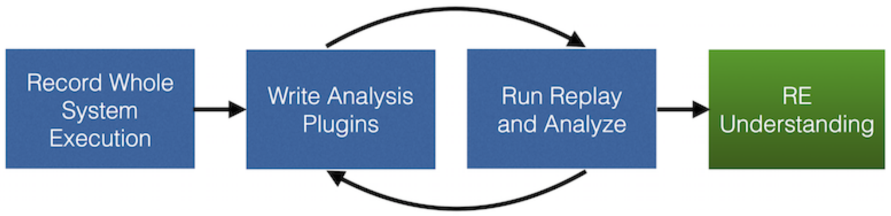


Figure 3.2: PANDA is employed by recording the execution, then writing plugins to extract the required information. The recording can be re-analyzed several times, targeting different kinds of information, in order to progressively build an understanding of the recorded execution. Image source: PANDA manual

Memory analysis Memory analysis is often employed in malware analysis, with frameworks like *Volatility* and *Rekall* including modules to automatically detect signs of infection, for example the presence of function hooks. Additionally, memory snapshots can be scanned for patterns belonging to known malware by using pattern-matching tools like *Yara* [33]. Both these detection techniques can be automatically executed as part of a sandbox analysis, by applying them to a memory snapshot taken at the end of the analysis. Some commercial sandboxes, like *VxStream Sandbox*[32] also make use of memory analysis to automatically inspect the code being executed.

Teller [18] proposes employing a differential memory analysis approach consisting in configuring a sandbox to trigger a full-system memory dump when certain conditions verifies, then analyzing the changes between the different dumps.

3.2.6 PANDA analysis framework

PANDA is an open source tool and platform for full-system (i.e., operating on the entire virtual machine instead of a single process) reverse engineering. It was presented by Dolan-Gavitt et al. in [19].

PANDA is implemented as a fork of QEMU, it offers an advanced record-replay feature, and a plugin API that makes it easy to use it as framework to perform several kinds of analysis.

PANDA's record-replay functionality allows recording a whole execution with a minimal disk space usage and then replay it with added instrumentation. This enables employing heavyweight instrumentation like the one needed for taint analysis without negatively affecting the execution of the sample.

SysTaint is built on top of PANDA and our approach borrows heavily from the ideas implemented in PANDA and its library of included plugins. Refer to chapter 5 for a description and references to the specific ideas.

3.3 Motivation

Even though automated malware analysis, in the form of sandbox software, is already widely employed in observing the behavior of a malware sample, it can only offer very limited help in the reverse engineering process. On the other hand, the existing approaches for automated malware reverse engineering, and in particular the ones aimed at the reverse engineering of the communication protocols, are not easy to apply in practice due to the heavyweight instrumentation required and the lack of publicly available open source implementations. Because of these limitations, manual debugging sessions remain the most widespread way of gathering information on the behavior and inner state of a sample to help in understanding and reverse engineering its inner workings.

The newly available record-replay approach offered by the PANDA analysis platform allows addressing some of the mentioned concerns, allowing the development of analysis solutions that are easier to apply in practice and can access in-depth information about the execution without disturbing it. To the best of our knowledge, there have been no previous attempts at building a malware analysis solution leveraging full-system record-replay.

3.4 Goals

The goal of this work is the development of a tool, SysTaint, which can help the analyst in the study and reverse engineering of recent and real-world malware samples.

We aim to make it easier to study malware samples, by replacing manual debugging sessions with a semi-automated data collection. In particular, we want to ease the study of malware communicating with C&C servers and employing encryption.

We want our approach to be able to:

- record the execution, then replay and re-analyze it at a later time, thus removing the reliance of the analysis on the presence and behavior of external servers
- uncover the contents of encrypted network communications
- follow the data flows that the sample uses in network communications, locating their usage and transformations by the sample's internal functions

Specifically, we also aim at improving state-of-the-art solutions by

- provide the analyst with useful information about the data exchanged through network communications, with techniques similar to the ones employed in *Dispatcher* [9]. Although we do not aim to automatically extract the details of the communication protocol, we aim to provide rich semantic information on the contents of network communication and allow the analyst to leverage the same information to study other behaviors of the sample. The information the analyst obtains can be put in relation to specific functions in the sample's code. In addition, we aim to make this tool as practical to use as possible and reduce the required resources and setup.

- allowing to inspect the data processed by the sample at various point in time, as done by time-travelling analysis tools like QIRA [20]. Differently from QIRA, we aim to support Windows programs, target longer executions and complex multiple-processes scenario, and provide the analyst with information on the data flows and the detected encryption function.

3.5 Challenges

Deriving semantics from low-level data The main challenge of any reverse engineering task is to recognize high level behaviors from large amounts of low level instructions and data that appear meaningless when not considered in their context. For this reason, to present the analyst with meaningful information about what is happening in the program, it is important to automatically infer as much high-level information as possible.

As SysTaint is based on PANDA, which works as a hypervisor, it observes the execution of the programs by monitoring the instructions and the memory accesses being executed. Determining what these instructions and memory accesses represent in the context of the current program and operating system is a complex task called *virtual machine introspection* (*VMI*).

Some information that are easy to access at run-time, for example the current thread identifier, normally available through the `GetCurrent ThreadId` Win32 API call, are harder to access by means of VMI, requiring knowledge of OS internals.

Further complicating matters, the internals of the Microsoft Windows operating system are only partially documented, and Windows' source code is not publicly available. Most of the information required to apply virtual machine introspection on Windows is therefore derived by researchers by means of debugging and by leveraging the debugging symbols Microsoft releases publicly. Projects like *Volatility* and *Rekall*, fortunately, already incorporate most of this knowledge.

Reconstructing the malware activity from a partial view of the system Despite, from the hypervisor standpoint, we have full visibility into system memory, some information may be harder to access by means of virtual machine introspection than they are from inside of virtual machine, querying the operating system. Some information, moreover, like the state of the files on the disk at a given moment, is not captured in the PANDA recording, and other information may be incomplete, as in the case of memory pages that the operating system swapped out to disk.

Timing When studying the behavior of a malware sample, timing plays an important role since, from the moment the sample is first executed, it can start several other processes and inject itself into some of them. An infected process' memory layout and code can also change significantly during the analysis because of the use of packing and some specific injection techniques.

Some information may only be in memory for a certain time-window, as for example temporary buffers containing unencrypted data, the unpacked code of the malware, or some dynamically loaded libraries. When using memory analysis or virtual machine introspection to obtain information, it is important to access memory at the right points in time.

Isolating the behavior of the sample Another challenge is distinguishing the behavior of the sample under analysis from the benign activities of the operating system and the other programs with which it interacts. This is particularly evident when analyzing the network activity logs, as several unrelated communications from different processes can overlap.

Moreover, malware code is often executed from inside an infected benign process (for example, *Internet Explorer*). It is necessary, in this case to distinguish the actions of the malware from the ones of the host process.

Performance and memory considerations Depending on the specific analysis being run, dynamic analysis can have a significant CPU and RAM overhead. This is the case with taint tracking, which requires instrumenting almost every assembly instructions, and requires potentially a very large amount of *shadow memory* to keep track of which memory locations are tainted.

When dynamic analysis is used to collect data, disk space is also a concern, and there is often a trade-off between disk space and the amount of details to capture. Some malware samples can produce a large amount of recorded activity as part of their normal operation – for example a ransomware encrypting a large number of files. Worse, some malware specimens purposefully perform a huge amount of unrelated activity as an attempt to produce too much activity to be recorded and analyzed and evade sandbox analysis.

Malware employing encryption As part of the malware authors' efforts to hinder the analysis, the communications between malware and their command and control server are usually encrypted, either by using an encrypted channel, for example HTTPS, or by using an unencrypted channel like HTTP and encrypting the payloads.

Malware employing evasion techniques As described in section 2.7, malware try to evade the analysis by detecting some artifacts of the virtual environment or analysis software. Additionally, even some strategy as simple as waiting a long-enough time before acting, if not detected, can effectively hide the malware activities from the behavioral analysis.

Unfortunately countering these strategies is hard and is part of an ongoing cat-and-mouse game between malware authors and analysts. As a consequence, the presence of malware samples able to evade a given automated analysis must be assumed.

Chapter 4

Approach

4.1 Overview

Our approach aims at providing the analysts with a tool to easily follow the data flows and identify the sample's functions that process any given data of interest. For instance, such tool can track the data read from a file, or received over the network, and help the analyst understand how this data is processed and used. Similarly, this tool can be used to understand what data a sample is sending through the network and how this data was obtained and transformed.

To do this, we collect in-depth data about a subset of the functions called by the malware sample and make use of taint analysis and cryptographic function detection to build a data-flow graph that makes it possible to interactively query and navigate the collected data.

It is possible to broadly divide our approach in the following steps:

1. **Recording the execution of the sample** A recording of the malware sample performing the activities we wish to study is obtained by executing it in a virtualized environment, either manually or through an existing sandbox software.

2. **Selecting the activities of interest** The analyst selects some specific activities he wishes to inspect and use as starting point for exploring the sample's behavior. For example, a given API call or network exchanges.

3. **Preliminary analyses** The tool performs some preliminary analyses on the recording, first to identify, among the processes in the recording, the ones involved in the behaviors we are interested in, then to detect the usage of cryptographic and compression functions in these processes.

4. **Data-flow analysis and data collection** The tool runs a more heavy-weight analysis, performing the data-flow analysis and collecting the input and output data of a subset of the internal functions of the sample. The collected data is stored in a file.

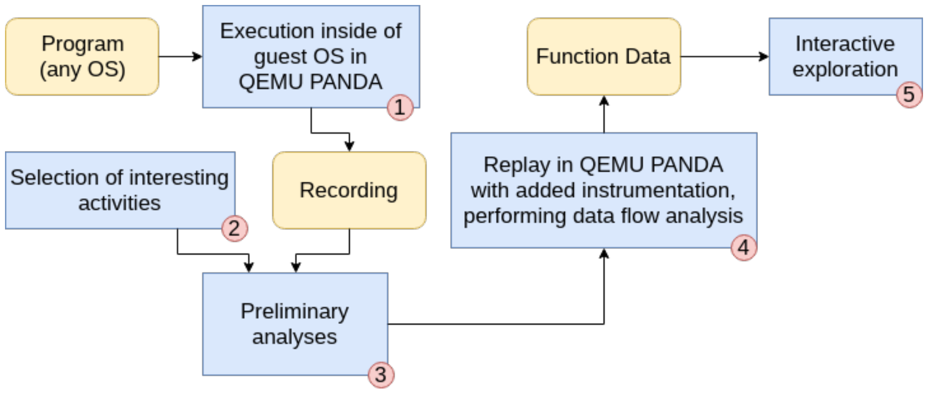

Program (any OS)
Execution inside of guest OS in QEMU PANDA
1
Function Data
Interactive exploration
5
Selection of interesting activities
2
Recording
Replay in QEMU PANDA with added instrumentation, performing data flow analysis
4
Preliminary analyses
3

Figure 4.1: Overview of the approach employed in SysTaint

5. **Interactive data exploration** The analyst interactively explores the malware execution by querying the collected data to find, annotate and study interesting functions and data.

4.2 Recording the execution of the sample

The first step of the approach consists in running the malware in a virtual machine for a given amount of time and recording its execution. All the subsequent analysis steps are performed on the obtained recording, so to not affect the execution of the sample or the communications with external servers. Our recording includes the activities of the whole guest virtual machine and is not limited to a single process.

Since the recording contains the activities of all processes, we can delay the detection of malicious processes to the later *preliminary analysis* phase. This is an advantage with respect to regular sandbox analysis, which needs instead to choose the processes to analyze in real time, relying on the ability to detect when the malware infects other processes. As the injection techniques are various and complex, the current state-of-the-art sandbox software is not always able to reliably detect them, and, as a result, it may completely ignore some malware processes.

Since, during the later phases, the instrumentation and data collection is performed by the hypervisor, there are no analysis-specific memory artifacts the malware can detect to evade the analysis. It is however worth noting that malware can still indirectly infer it is being analyzed by detecting the virtual environment.

It is also possible to record the execution of the sample as part of a regular sandbox analysis to take advantage of the existing analysis infrastructure, the data it collects, and its anti-evasion countermeasures. Since taking a recording does not affect the sample's execution, except for an acceptable slowdown, it is also possible to configure an existing sandbox setup to transparently and automatically take a recording of all the analysis it normally performs, so that researchers can later analyze or share them if needed.

To minimize the time needed for the analysis, the recording should comprise as few instructions as possible, ideally only the ones pertaining the activities of the sample one wishes to study. To reduce the number of instructions in the recording, the analyst should disable the other background processes running in the virtual machine.

4.3 Selecting the activities of interest

The second step requires the analyst to manually identify some data or behavior he wishes to inspect. In the preliminary analyses phase, we use this data or behavior to identify the processes of interest, excluding the unrelated ones from the analysis. Then, in the interactive analysis phase, we use them as starting point for the interactive exploration.

The analyst can extract the required information by filtering and inspecting the available logs, for example selecting and extracting some encrypted payload from the network log or choosing an entry from the behavioral log provided by a sandbox software's instrumentation (if present during the recording). If a behavioral log is not available, it is also possible, in alternative, to extract a list of the system calls executed by all processes, with the related data and call stack, by running the analysis described in the *data-flow analysis and data collection* phase, but without restricting it to specific processes nor enabling taint analysis.

4.4 Preliminary analyses

The preliminary analyses phase consists in three sub-steps.

1. Collecting information about the processes in the recording to later display the memory addresses of data and functions to the analyst in a more meaningful way, as well as obtaining the addresses of known functions in Windows's APIs.

2. If a set of processes of interest has not yet been identified, it is possible to find the processes and functions processing a given previously identified piece of data by scanning the recorded memory accesses for reading or writing operations on such data.

3. Detecting the use of cryptographic functions, so that we can easily locate them and their outputs during the later phases.

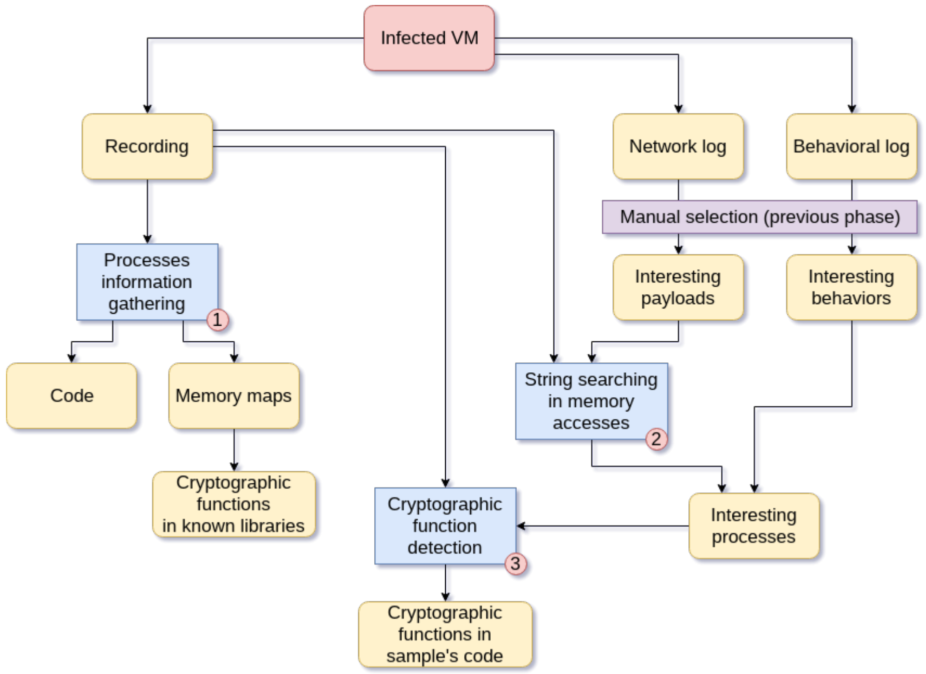

Infected VM
Recording
Network log
Behavioral log
Manual selection (previous phase)
Processes information gathering
1
Interesting payloads
Interesting behaviors
Code
Memory maps
String searching in memory accesses
2
Cryptographic functions in known libraries
Cryptographic function detection
3
Interesting processes
Cryptographic functions in sample's code

Figure 4.2: Information obtained during the preliminary analyses

4.4.1 Collecting information on the processes

During this phase, we replay the recording a first time, using virtual machine introspection to automatically collect information about every process in the recording. In particular, we analyze each process' memory right before its last recorded instruction. By inspecting each process' memory layout immediately before its last instruction, we are assuming that the sample does not unload any of its code or libraries from memory before exiting. This assumption holds for the malware samples we tested, but it is violated by more complex unpacking techniques. In that case it is possible to repeat this step of the analysis and take a snapshot of the process memory and code when the sample is performing some particular action of interest.

For each process, we collect the contents of all regions in its address space, and the list of the functions exported by every loaded library. This data is useful for the analyst in the interactive analysis phase, for instance to inspect the code at a given address, or to distinguish memory belonging to a process' *stack* or *heap* or mapped to a file. Finally, the list of exported functions is used to resolve or look-up the address of known functions in order to recognize them.

4.4.2 String searching

A quick way to find the processes and the functions dealing with a particular piece of data is by monitoring the memory accesses and logging the occurrences of some known string of data, for example a ciphertext extracted from the network logs.

There are two methods to search for a string being written or read from memory. The first one is based on [14] and around the concept of *tap points*, locations in the execution identified by process ID, program counter, and caller address. The data being read and written at each tap point is seen as a stream of bytes, which is monitored for the occurrence of the searched strings. The second approach is instead based on the per-function-call buffering of the data being written and read. When the function call returns, the buffers it has read or written are scanned for the searched strings.

The two approaches have pros and cons, the approach based on tap points is faster, and can uncover the patterns with which a certain data is read inside of a given function, but may miss some matches when the data is not read in order, or when the string we are searching for is read by parts, in different points of a given function.

If the payload is longer than a couple of words, it is preferable splitting it into smaller chunks before searching for it so that it can still be found if parts of it are read by different functions. Moreover, by inspecting the patterns with which the chunks are read or written to memory, it is possible to infer some information: whether the data is being read in one or multiple function calls, and whether a function is simply copying the data to a different location or transforming it.

Despite this technique is useful in finding relevant processes and functions, it is limited to searching exact patterns and, although the analyst can use it iteratively for every transformation of the data, doing so is cumbersome and error prone, and it is preferable to instead rely on the taint tracking information.

Some example of using the string searching technique can be found in chapter 6.

4.4.3 Locating cryptographic functions

Locating cryptographic, compression and encoding functions by means of a preliminary analysis has several advantages, the first of which being able to highlight and track their input and outputs during the later phases, making it easy to locate the unencrypted data.

Cryptographic functions are also treated differently from the others during the later data collection phase. In particular, they are treated as nodes in the data-flow graph, and their output is tainted with a new label. Doing so has also the purpose to compensate for some practical limitations of our taint tracking implementation, which may not be able to track complex cryptographic or compression functions correctly. Having the cryptographic functions in the data-flow graph also makes it easier for the analyst to follow the flow of data through the application and understand how it is transformed.

As described in subsection 3.2.4, the literature contains several approaches to perform cryptographic function detection. We decided to evaluate two approaches, namely the heuristic based on the ratio of arithmetic operations from [9] for its simplicity, and a custom heuristic based on several per-function metrics, detailed in the next paragraphs.

It is worth noting that, since we work on the recording and know the ciphertext in advance, it is also possible to locate the functions processing the ciphertext, and by extension the plaintext, by searching for the memory accesses writing or reading the ciphertext.

In order to apply the heuristics, we replay the execution, collecting several per-function-call metrics:

- the size of the function (in basic blocks)
- the total number of executed basic blocks
- the presence of loops (inferred from the previous two metrics)
- the number of arithmetic and total instructions executed
- the size, entropy, number of ASCII characters of each read and written buffer

Custom heuristic

Our custom heuristic aims to detect both cryptographic and compression functions, and it is based on the following observations:

- cryptographic/compression functions read or write high entropy data
- the sub-tree of the call tree having as root a cryptographic/compression function call is shallow
- cryptographic/compression functions spend most of the time in loops

Each function call is evaluated by itself, and the results are then aggregated based on the address of the function. If more than two calls of a given functions fail to satisfy the heuristic, the whole function is discarded. This allows us to discard functions like `memcpy`[1] that may happen to copy high-entropy data.

It is important to note that read/written buffers are always assigned to the current (at the top of the stack) function call. As a result, the above heuristic is aimed to find low-level cryptographic primitives.

As the last step of applying this heuristic, we inspect the callers of the identified primitives in order to find, if present, parent functions that process the buffer as a whole. We go through the list of detected primitives, and we

[1]`memcpy` is a function of the C standard libraries copying a buffer from a location to another

replace a primitive with its caller C if the primitive is only called by C and if the sub-trees of the call tree having as roots the instances of C are also shallow.

It is worth noting that, since we collect data at function-level granularity, our heuristics cannot detect the smaller obfuscation functions that have been inlined in the caller's body.

4.5 Data-flow analysis and data collection

During the data collection step, the recording is replayed with taint analysis enabled, collecting and logging *events*. An *event* is defined as a series of instructions executed on a given thread between some start and end times.
Events can either be:

Syscalls which cover the activity of kernel-space code from a *sysenter* to a *sysexit* instruction.

Encoding functions which cover the activity of a known data-transformation function, from their call to return.

Common functions which cover the activity of a function taking as input tainted data.

Each event is identified by a progressive ID, *label*, the address of its entry point, start and end times, its process and thread IDs, a call stack, an optional *tag*, a set of read and written buffers with their associated taint labels. We call *active events* the events that have started but not yet ended.

If the start of an event is detected on a thread where there is already an active event, the new event can either be ignored, or "nested" to the previous one, according to specific rules. If the event is nested, it is pushed, on top of the previous event, on a per-thread stack of active events. When an event terminates, it is removed from the stack. We define *current event* for a given thread the topmost event in the per-thread event stack.

If on a thread there is a *current event* (i.e., the event stack is not empty), all memory accesses happening on that thread and reading or writing to the user-space portion of the address space are attributed to the *current event* and logged. All reads or writes to the kernel-space portion of the address space are discarded, since, despite *syscall events* cover code executing in kernel-space, we are only interested in the data entering or exiting the regions of memory under the control of the user-space malware code we are studying.

Events are nested or ignored according to these rules: 1. If there is no *current event*, any event can be set as such 2. If the *current event* is a *common function*, any new event can be nested on top of it 3. If the *current event* is a *syscall*, no event can be nested on top of it - i.e., any nested *syscall* is discarded 4. If the *current event* is an *encoding function*, only *syscalls* can be nested on top of it

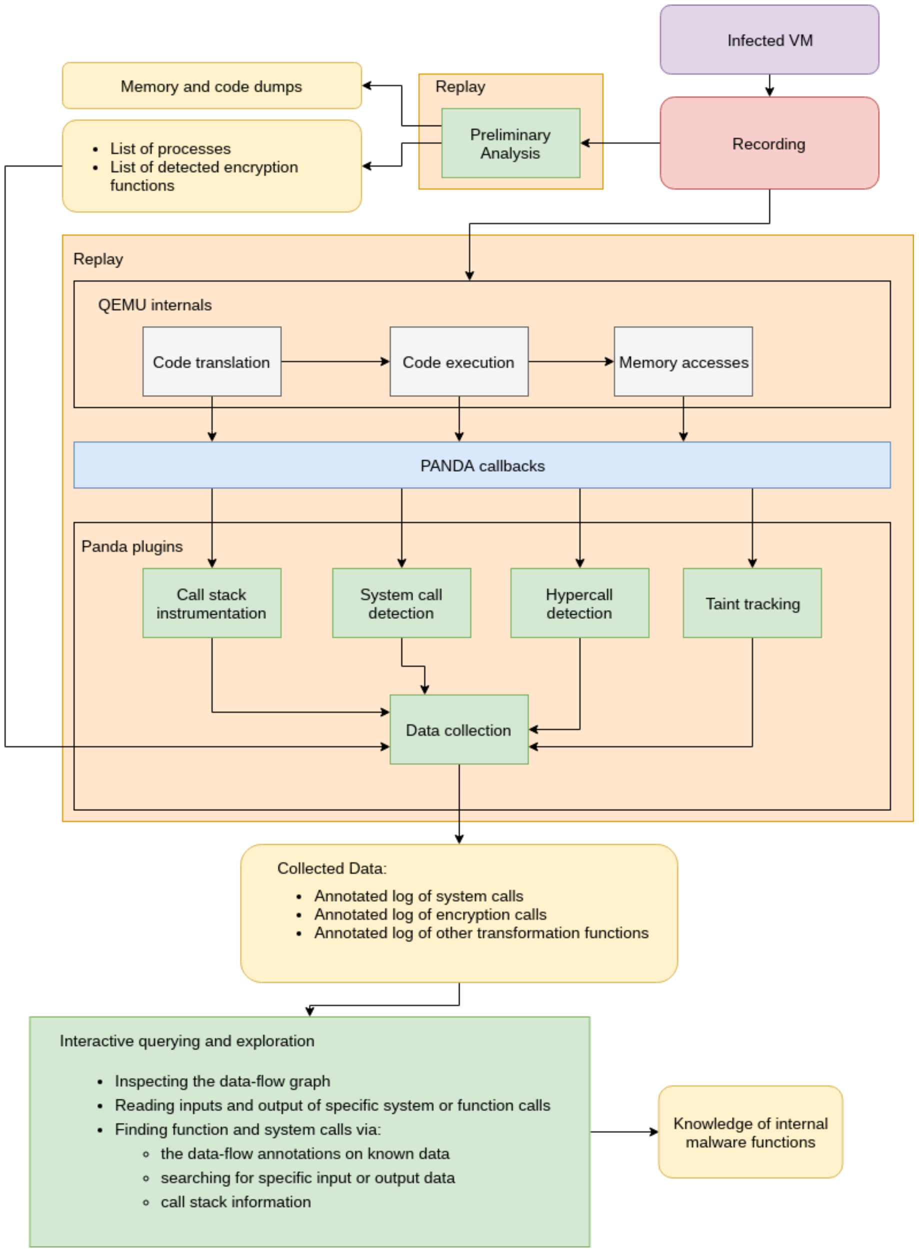

Infected VM
Recording
Replay
Preliminary Analysis
Memory and code dumps
List of processes
List of detected encryption functions
Replay
QEMU internals
Code translation
Code execution
Memory accesses
PANDA callbacks
Panda plugins
Call stack instrumentation
System call detection
Hypercall detection
Taint tracking
Data collection
Collected Data:
Annotated log of system calls
Annotated log of encryption calls
Annotated log of other transformation functions
Interactive querying and exploration
Inspecting the data-flow graph
Reading inputs and output of specific system or function calls
Finding function and system calls via:
the data-flow annotations on known data
searching for specific input or output data
call stack information
Knowledge of internal malware functions

Figure 4.3: Detailed overview of SysTaint's data collection phase

4.5.1 Taint tracking

When a memory access is detected on a given thread, we check if on that thread there is a *current event* active event. If there is, all memory accesses by its thread on the user space portion of the process' address space are attributed to that event, and logged.

If the memory access is a *read*, the taint labels corresponding to memory locations being read are fetched and logged together with the read data. If the memory access is a *write* and the event is either a *syscall* or an *encoding call*, the memory location being written is tainted with the ID of the current event.

This approach is simple and does not require any knowledge of the arguments of syscalls or encoding functions. Despite its simplicity, this approach works well in our experiments.

The taint information is also used, during the analysis, to decide which function calls to log as *common functions*, since, to save on disk space and avoid collecting uninteresting information, by default only the functions processing tainted data are logged. *Common functions* do not taint any data in order to save memory and not to overload the data-flow graph and the taint tracking implementation. The logged *common functions* can be queried to understand how the tracked data is transformed and to manually uncover undetected encryption functions.

Taint policy

Our taint policy is a variation of the one implemented in [17]. It is structured around the concept of *taint locations* to which a set of *taint labels* can be associated. *Taint locations* can either be memory locations, identified by a physical memory address, or register locations, identified by a register and an offset. A taint location always represents a whole byte of data, as a result, our taint tracking implementation works fine for coarsely tracking the movements and transformation of data, but it is not fit to be employed to accurately determine which bits an attacker is able to influence.

The taint status of each *taint location* is updated by instrumentation that is added to the assembly instructions moving, transforming or combining data, and follows this set of rules:

- when a taint location's content is overwritten with untainted data, its taint labels are cleared
- when a taint location's content is overwritten with tainted data, its taint labels are overwritten by the source's taint labels
- when the contents of two taint locations are combined by means of an arithmetic instruction, their set of labels is merged
- certain operations used to clear the registers (for example `xor eax eax`) are correctly handled, clearing the target register's label set.

Additionally, when the configuration option *taint on pointer dereference* is enabled, the load operations moving data from a memory address A to a register

B also cause the contents of the register to be tainted with the labels assigned to the address. For example, in the instruction `mov eax [esi]`, moving the memory contents pointed by `esi` to the register `eax`, causes `eax` to be tainted by the union of the labels assigned to the register `esi` and to the data in memory pointed by the address in `esi`. This option allows data dependencies to be correctly propagated in the cases when the output bytes are not directly obtained by operations on the inputs bytes. This happens with functions employing substitution tables, as the tainted input bytes are used as offset to load untainted bytes from a substitution table. Conversions to base64 or from a charset to another are common examples of this kind of function.

To correctly follow the data-flow through this kind of functions it is necessary to either enable *taint on pointer dereference* or to detect these functions beforehand and treat them as encryption/compression functions. The reason for not always enabling *taint on pointer dereference* is that our test shows it can create problems in some situations, leading to a large amount of addresses being tainted.

In order to discard irrelevant dependencies between functions, which is particularly important when the option *taint on pointer dereference* is enabled, we have manually excluded the registers `EBP` and `ESP` from tainting. This allows the taint tracking implementation to avoid detecting irrelevant data-flow dependencies between a function A, and a function B called immediately after, which accesses data using the stack pointer.

4.5.2 Interaction with sandbox software

To leverage the existing domain knowledge encoded in the current sandbox products, we designed our tool so that it can take inputs from additional instrumentation running alongside the sample to study, inside the virtual machine. This in-system instrumentation can interact with our tool through *hypercalls* (special instructions resulting in calls to the hypervisor), which are executed during the recording phase and become part of the recorded execution. The in-system instrumentation uses hypercalls to notify SysTaint when something happens, for example when the sample calls one of the high-level API function it monitors. The hypercalls also carry an identifier, which is used to cross-reference the entries in the sandbox's behavioral log from SysTaint's events and vice versa.

Integrating SysTaint with existing sandbox software is useful for two reasons. On a hand, it allows us to leverage the existing knowledge of the Windows APIs (for example knowing which functions are important and what are they arguments). On the other hand, it allows augmenting the information collected by the sandbox software, allowing the analyst to use the entries in the sandbox's behavioral log as starting point to explore the sample's execution using SysTaint, leveraging the data flow and the extended call stack information.

The events notified by the sandbox software, *external events*, are treated as *tags* and put in separate per-thread stacks. When an event (for example a *syscall*) starts on a given thread, it is assigned the topmost tag of the tag stack of that thread. We decided to treat higher-level APIs as tags and not as events because, differently from *syscalls*, they can be bypassed or have their behavior altered via inline hooks (both by the sample and by the inline instrumentation).

Note that making use of a sandbox software is entirely optional and may even be disadvantageous in cases when the malware is able to detect it. In our experiments, the sandbox software was only helpful in the analysis of the older malware samples.

4.5.3 Syscall handling and tainting

Syscall events cover the instructions executed in kernel-space between a *sysenter* and a *sysexit* instruction, respectively moving the control to the kernel and back to the user-space code. Syscall events allow logging and taint-labelling all data that enters a process' own virtual memory, acting as tap/sink points for taint analysis purposes.

The fact *syscalls events* cover kernel code plays well with our simplistic tainting policy. Because we taint all writes made during an event in user space memory, writes to some internal program state may cause irrelevant data dependencies to be shown between calls that, for example, happened to increment the same counter. Syscall events, on the other hand, have their internal state on the kernel side of the address space, which is excluded from taint-analysis.

The flip side of using syscalls as *events* is that syscall, by themselves, give little semantic information to the analyst, especially considering that the arguments to the syscalls are mostly opaque identifiers. To understand what is happening at a higher level, the analyst either need to refer to the assigned tags (if available), or to follow the dependency graph to find syscalls producing those identifiers. As an example of this last method, from a `NtReadFile` syscall reading some data of interest, it is possible, following the taint label assigned to the file descriptor, to find the `NtOpenFile` syscall, among whose reads there is the path of the file it opened.

4.6 Interactive data exploration

Once SysTaint completes the data-collection phase, the analyst can query the collected data interactively to gather information about specific behaviors, obtain information on the functions involved in them, find the data they process, its provenance and how it is used in successive syscalls.

The collected data takes the form of a list of *events*, each logged together with their input and output buffers and their call stacks. The input buffers can have taint labels attached, indicating the list of previous events that generated or transformed them. The logged events are either *system calls*, *encoding calls* or *common functions.* These events can optionally have a tag indicating that they happened during a call to a high-level API.

When exploring the execution, the analyst first looks for a set of events to use as starting point, which he can locate in several ways:

- by examining and filtering the list of recorded events, for example taking the encryption call or the syscall of the kind `NtDeviceIoControlFile` (which are responsible for sending and receiving data through a socket) that has read the most data
- by searching for known strings in the buffers the events read or wrote, for example some payload extracted from the network log
- by tag, making use of the list of high-level API calls (provided by the sandbox in our implementation)
- by parent functions, looking for the address of a specific function in the call stack of the logged events

Once an event E to use as starting point has been found, there are several ways to explore the execution.

- by following the data-flow graph backward, to find the functions producing or transforming the inputs of E
- by following the data-flow graph forward, finding the functions using the outputs of E
- by looking for events with the same functions in the call stack
- by looking for *common functions* writing to the addresses corresponding to the input buffers of E

By moving from an event to another as described, the analyst is able to go, for example, from the event sending an encrypted buffer through the network, to the event encrypting the buffer. From a buffer, it is possible to find the functions that wrote it, or the syscalls that originally obtained the data.

Since the collected data mostly consists in detailed and low-level information, a querying interface is tasked to annotate the data with all known information when presenting it to the analyst. For example, the querying interface uses the information obtained during the preliminary phases about the memory regions, the loaded libraries, and the addresses of known functions to give more meaningful names to functions and addresses.

Understanding how data flows between the functions, by finding which system calls retrieved certain data and which functions used it, can help the analyst build a knowledge of the various functions in the malware. By using SysTaint in conjunction with an interactive disassembler like *IDA Pro*, the analyst can use the knowledge and addresses obtained by SysTaint to annotate the functions with his findings, as well as inspect their code. On the other hand, given the address of a function from the disassembler, the analyst should be able, by querying the data collected by SysTaint, to find its arguments and the buffers it read and wrote in all its invocations.

For examples of using the collected data to study recent malware samples refer to chapter 6.

Chapter 5

Implementation Details

5.1 System architecture

SysTaint is implemented as a collection of plugins for the PANDA analysis framework and scripts written in the Python programming language for analyzing the collected data. For simplicity of implementation, all plugins have been written to only support Windows7 SP1 on the x86 architecture as guest OS.

The main plugin, *SysTaint*, performs the actual data collection and taint tracking, producing the output file for the interactive querying. Other plugins, like *procinfodump*, *fnmemlogger* and *stringsearch2*, are used during the preliminary analysis. Finally, plugins, like *callstack_instr*, *sysevent* and *tcgtaint* provide common functionalities.

A brief description of each plugin is presented here. Note that many of the plugins and ideas described here are based on existing PANDA plugins. The differences with the original versions is described in detail in the following sections.

- **callstack_instr** maintains a shadow stack by monitoring calls and returns, assigning a unique ID to the functions called.
- **stringsearch2** reads a list of strings, chunks them, then monitors memory accesses for matches
- **fnmemlogger** collects statistics about the functions called during the execution to be used for cryptographic function detection
- **procinfodump** collects information about each process, its address space layout, and the loaded libraries
- **sysevent** monitors the execution of hypercalls. It starts the recording phase when the first hypercall is detected and collects information about the processes issuing them.

- **tcgtaint** contains the taint-tracking implementation, which it exposes to the other plugins.

During the recording process, PANDA can also integrate with the popular Cuckoo Sandbox, in order to be able to deploy it as a drop-in replacement for QEMU in existing Cuckoo sandbox setup, and to be able to leverage and augment Cuckoo's behavioral logs. To achieve this integration, a customized version of *Cuckoo Monitor* was employed.

5.1.1 PANDA plugins architecture

PANDA plugins are dynamic libraries, written in C++ and loaded at run-time by the main PANDA/QEMU process. After they are loaded, PANDA plugins can register callbacks that PANDA calls at certain points of the execution, for example after a basic block of code has been translated, or when a memory write is about to happen. The main use of PANDA plugin is to instrument the execution during the replay phase and collect data.

It is possible to use PANDA plugins in both the recording phase and in the replay phase. It is important to note, however, that in the replay phase PANDA plugins cannot alter the execution of the programs inside the virtual machine since, by design, during replay most of the virtual hardware is turned off and PANDA can only replay the hardware interactions it recorded.

5.2 System details

5.2.1 Integration with Cuckoo Sandbox

Integrating Cuckoo Sandbox and PANDA, so that the analysis cuckoo performs are automatically recorded, is easy and consists in configuring Cuckoo to use PANDA as virtual machine provider. We made minor changes to Cuckoo so that it is possible to specify, in a configuration file, the settings of the virtual machine and the PANDA plugins to use during recording.

We made some changes to the Windows version of *Cuckoo Monitor*, the analysis component of Cuckoo Sandbox which is run alongside the sample to analyze and employs function hooking to monitor its execution, so that it communicates with SysTaint via *hypercalls* every time it intercepts and logs a call to an interesting Windows API. Before and after the call to the original function, Cuckoo Monitor performs a hypercall to signal the call and the return of the intercepted function. These hypercalls also include a unique ID that allows referring to the Cuckoo log's entries from SysTaint events and vice versa. The modified Cuckoo Monitor generates these unique IDs before each call, using a per-thread counter.

```
uint32_t current_call_counter = call_counter++;
uint64_t unique_id = (get_current_thread_id() * 1000000) +
    current_call_counter;

```

```
do_hypercall(HYPERCALL_FN_ENTER, unique_id);
{{ call_old(hook) }}
do_hypercall(HYPERCALL_FN_EXIT, unique_id);
```

Listing 5.1: Hypercall notification code added to Cuckoo Monitor for SysTaint integration

Hypercalls are made by means of *inline asm*, moving some values inside of the registers then executing a `cpuid` instruction.

```
void do_hypercall(int enter, uint32_t unique_id){
    uint32_t v_eax = 0xFFAAFFCC;
    uint32_t v_ebx = enter;
    uint32_t v_ecx = unique_id;
    uint32_t v_edx = 0;

    asm __volatile__
        ("mov  %0, %%eax \t\n\
            mov  %1, %%ebx \t\n\
            mov  %2, %%ecx \t\n\
            mov  %3, %%edx \t\n\
            cpuid \t\n"
        : /* no output registers */
        : "g" (v_eax), "g" (v_ebx), "g" (v_ecx), "g" (v_edx)
    /* input operands */
        : "eax", "ebx", "ecx", "edx" /* clobbered registers */
        );
    return;
}
```

Listing 5.2: The do_hypercall function

The `eax` register contains a magic value in order to distinguish this hypercall from regular `cpuid` calls, `ebx` contains a constant indicating whether the hypercall indicates the external event is beginning or finishing, `ecx` contains the unique identifier of the external event.

On the other side, the *sysevent* PANDA plugin listens for such hypercalls, abstracts them to External Events, then passes them to the other plugins.

When the analysis is started through Cuckoo, the *sysevent* plugin monitors the hypercalls and asks PANDA to start recording when the first hypercall is received, to exclude from the recording the preparation phase of the Cuckoo analysis.

5.2.2 Capture process

The first part of the analysis consists in capturing a recording of the sample to analyze by executing it inside of a PANDA virtual machine.

The procedure for obtaining a recording is simple: bringing the virtual machine to some initial state, starting the recording, executing the malware sample, waiting until some meaningful network traffic is produced, and finally stopping the virtual machine.

It is also possible make use of a purposely set up Cuckoo sandbox, configured to use the modified Cuckoo Monitor and use PANDA as hypervisor, to automatically run the sample inside of the virtual machine. The procedure in this case is similar, but requires saving the initial state, with the Cuckoo agent already started, as a QEMU *savevm snapshot* (consisting in a snapshot of memory, devices, and disks stored inside of the *QCOW2* image file). When the analysis begins, Cuckoo takes care of restoring the VM, launching the sample, and letting it execute for a given amount of time. The *sysevent* plugin automatically starts the recording process when it receives the first hypercall from the modified Cuckoo Monitor.

It is important to note that, although for speed and convenience the analyst can prepare the virtual machine by using QEMU in KVM mode, the QEMU snapshot of the running machine must be created by booting the virtual machine by using PANDA in TCG mode.

5.2.3 Gathering information about the running processes

This step aims at collection information about the various processes of interest, obtaining the code from the executable regions of each process' memory, as well as the addresses of known functions from Windows's libraries.

The first step consists in gathering statistics about all the processes that have been active during the recording session by using the *asidstory* PANDA plugin. This information includes, for each process, the address space identifier, the process identifier, the process name, and the time of the process' first and last instructions.

The executable code and the information about the address space are collected, before each process' last instruction by using memory forensics techniques, making use of the *Rekall* framework, and some functions ported from *Volatility*.

To this purpose, we wrote a *procinfodump* PANDA plugin that embeds a Python interpreter through *pybind11*[1] and exposes the memory of the guest VM to Rekall as an address space object.

On startup, *procinfodump* loads the list process identifiers and time of their last recorded instruction, expressed in number of instructions from the start of the recording. When one of the target points is reached, the plugin executes a Python script that makes use of Rekall to collect the following information:

- The list of regions composing its address space, each with start address, length, permissions and name of the mapped file (if any). This information is obtained by parsing the *VAD tree* [3]

- The list of stacks and heaps

[1]`https://github.com/pybind/pybind11`

- A dump of all known memory regions
- A list of all exported functions for all DLLs in the process' address space

5.2.4 Shadow stack

The shadow stack implementation is based on the plugin *callstack_instr* included in PANDA, which works by monitoring the execution of `call` instructions, and the execution of a new basic block.

We extended the original implementation so that it assigns an ID to the individual function calls, allowing the client code to distinguish the completion of individual calls.

The implementation has also been extended to support separate per-threads shadow stacks. Each stack is identified by its thread id.

The current thread ID is obtained by reading the thread-information block (*TIB*) from memory, as it is not possible to call the dedicated Win32 API function during the replay. The TIB is a user-space structure holding information about the current thread. When the processor is executing user-space code, a pointer to the TIB is available in the `FS` segment register.

```
CPUArchState *env = (CPUArchState*)cpu->env_ptr;
uint32_t tib_address = env->segs[R_FS].base;
uint32_t curr_thread_id_address = tib_address + 0x24;
uint32_t curr_thread_id;
panda_virtual_memory_read(cpu, curr_thread_id_address, (
    uint8_t*)(&curr_thread_id), 4);
return curr_thread_id;
```

Listing 5.3: Obtaining the current thread identifier from PANDA

The shadow stack is implemented as follows (as inherited from the original plugin):

- When a basic block finishes executing, the block is disassembled in order to check if the last instruction was a `call`, if it is, its address is recorded as return address.
- When a basic block is about to be executed, its address is checked against the stored return addresses of the last 10 functions on the stack. If there is a match, the matching call is assumed to have returned, and all the calls on top of it are marked as calls with missed returns and removed from the stack.

It is worth noting that in x86, it is possible to call a function without using the `call` instruction; however, a `call` instruction is what all compilers generate.

Matching the current address against several known return addresses makes the implementation robust for the cases in which the `call` instruction is abused, for example as an anti-analysis measure, or as a way to access the current program counter.

The same method has been applied to the other stack implementations, namely the tag stack, and the *common function* stack.

5.2.5 Syscall tracing

To perform syscall tracing we do not use the existing PANDA plugin, but a close variation thereof, in order to have run-time information about the prototypes of the syscalls being executed.

If the syscall prototype is available, the syscall arguments are read from the syscall stack and logged in the *Syscall Event*, together with the syscall number.

The remaining part of the implementation is exactly as in the original plugin and works as follows: the presence of the `sysenter` instruction is detected at basic block translation time, its address is recorded, and PANDA is asked to trigger a callback when that instruction is executed. When the callback fires, the current program counter is checked against the recorded ones, and if there is a match, the syscall is assumed to have started and the return address is stored, analogously to what happens for the shadow stack.

5.2.6 Encryption functions heuristics

The *fnmemlogger* plugin is tasked to gather statistics about every called function. These statistics are then used to decide if a given function is an encryption/decryption one. To keep the analysis time low, only the first 10 calls of any given functions are analyzed.

Function statistics gathering

The *fnmemlogger* plugin makes use of *callstack_instr* in order to track the calls and returns of functions in each thread, and to provide an identifier, *callID*, for each call.

It keeps a map of *CallInfo* records for each *callID*, and a counter, for each function address, of how many times it has been called.

When a callback fires, for example because of a memory access, *fnmemlogger* retrieves from *callstack_instr* the current *callID*. If there exist a record for that *callID*, that memory access contributes to the statistics of that record, in the other case, nothing is logged.

The statistics gathering works as follows:

- When a basic block is first translated, it is disassembled, and the number of arithmetic and total instructions is stored, associating it to the virtual address of the block.
- When a function is called, if it has been called less than 10 times, a *CallInfo* record is created for the current *callID*.
- After a basic block has been executed, the count of many times the active function executed each of its basic blocks is updated, as well as the count of arithmetic and total instructions executed.
- Every memory access performed by a thread is added to the *readset* or *writeset* of the current *CallInfo*

- When a function call returns, its sets of read and written memory location are partitioned in contiguous buffers and summarized with their length, entropy, count of bytes in the ASCII range, and the number of null bytes.

  The number of blocks executed, the maximum number of executions throughout the blocks, the number of distinct blocks executed are computed.

  The collected information is logged to file, and the call is removed from the list of tracked calls.

5.2.7 Data-flow analysis

For flexibility and performance reasons we decided to employ an improved version of the taint tracking implementation from QTrace [17] instead of the LLVM[2]-based taint tracking implementation included in PANDA.

The taint instrumentation is applied to the code during QEMU's code translation and, in particular, inside of TCG's frontend code. Calls to custom QEMU helpers are added to the intermediate TCG code, after every TCG instruction, to propagate the taint information.

Most of the taint engine is implemented in the *tcgtaint* PANDA plugin, whose functions are called by the aforementioned helpers. Other than increasing modularity and simplifying the build process, having the taint implementation as a PANDA plugin also allows exposing the taint engine to other plugins via PANDA's plugin-plugin interface.

The shadow memory is implemented as a hash map having as key the physical address of each tainted byte. We have decided on using the *sparsepp*[3] implementation to handle the large number of records without degrading performances.

A number of improvements have been made upon the original QTrace implementation in order port it to QEMU2, add support for more x86 instructions and for the *taint on pointer dereference* option.

Limitations In addition to the ones due to the approach, and described in subsection 4.5.1, our implementation has some important limitations. The most prominent is due to the considerable number of complex and high level x86 instructions, mostly in the SSE and AVX extensions to the x86 instruction set.

Due to their complexity, most of these instructions are supported in QEMU by means of helpers, which can only be instrumented by changing the code of each one. Due to the effort required, we decided to leave these instructions non-instrumented.

A partial support for the simpler MMX instructions, which are not implemented via helpers has however been added. This allowed tracking instructions like `movq`[4] whose usage we detected during our experiments.

[2]https://llvm.org/

[3]`https://github.com/greg7mdp/sparsepp`

[4]The `moveq` instruction is used to moves 64bit of data at a time in a 32bit system

Performance considerations Instrumenting the code by adding calls to QEMU helpers has a significant cost in term of performance. However, our experiments proved this cost to be lower than the one required for LLVM translation and recompilation in the LLVM-based taint instrumentation plugin included in PANDA.

The memory usage of *tcgtaint* was, in our experiments, also significantly lower than the LLVM counterpart, allowing the analysis to be run on a common laptop. This difference, however, may have been also due to a different tainting policy we adopted when switching to our taint-tracking implementation, excluding some registers like `EBP`, and `ESP` from the analysis to avoid the detection of irrelevant dependencies between functions.

5.2.8 Considerations on parallelizing the analysis

Although QEMU makes use of several threads, all of guest's code is executed in QEMU's CPU thread. PANDA's added instrumentation, which gets executed via callbacks, is likewise run in the same thread. This means the analysis, which are strongly CPU-bound, cannot automatically take advantage of multiple processing cores.

There are two possible approaches for running the analysis in parallel, splitting the work among several PANDA instances. One consists in splitting the recording in chunks and analyzing them individually, the other consists in running the analysis on the whole recording, but limiting each to a specific part of it, for example to a subset of the processes in the guest VM.

Working on chunks of the recording implies, unfortunately, a loss of information for many PANDA plugins. The *Callstack_instr* plugin is only able to show in the call stack the functions that began after it started monitoring the execution, and the *SysTaint* plugin is only able to show dependencies from syscalls started after it began monitoring the execution and propagating taint information.

The only kind of analysis that can easily be run in parallel is string searching (albeit with a loss of information in the reported call stacks), as strings are read and written within short time spans.

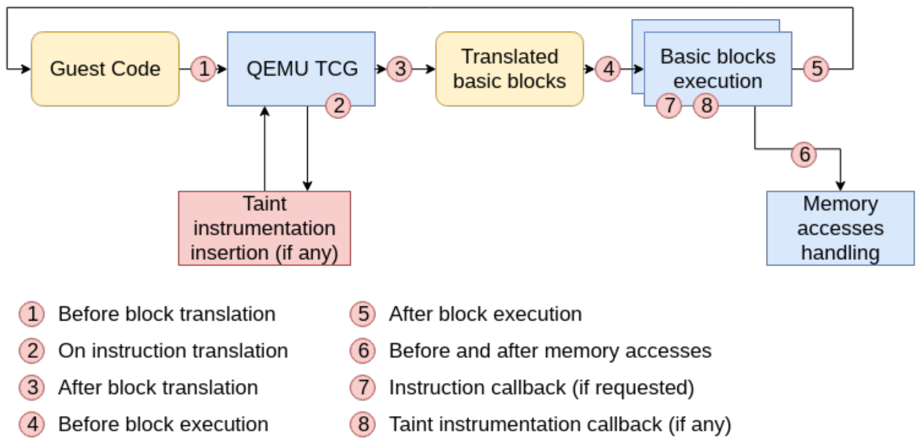

Guest Code
QEMU TCG
Translated basic blocks
Basic blocks execution
Taint instrumentation insertion (if any)
Memory accesses handling
1 Before block translation
2 On instruction translation
3 After block translation
4 Before block execution
5 After block execution
6 Before and after memory accesses
7 Instruction callback (if requested)
8 Taint instrumentation callback (if any)

Figure 5.1: Instrumentation points available to PANDA plugins

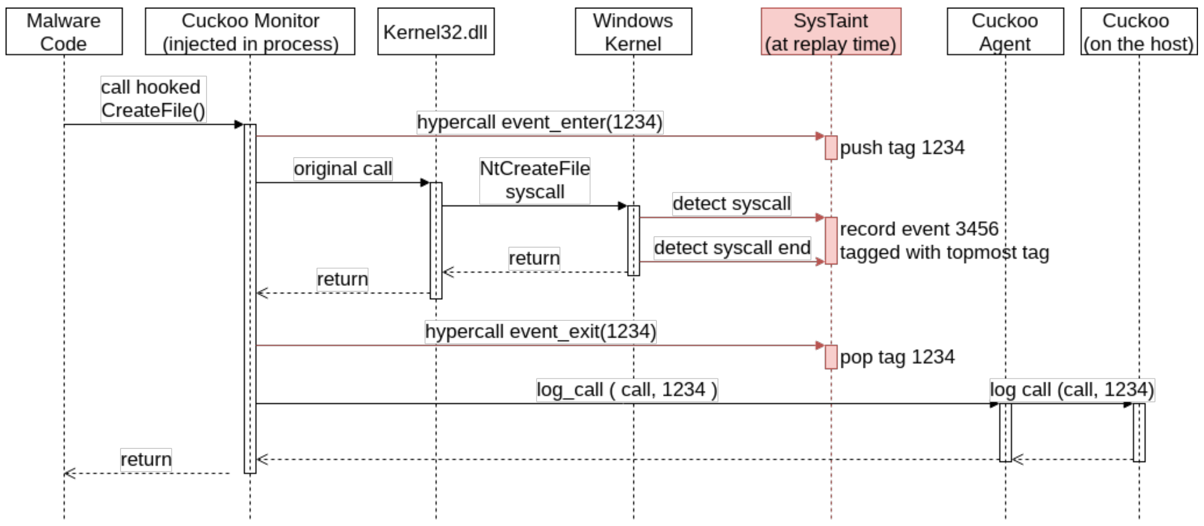


Figure 5.2: Low level details of the Cuckoo-SysTaint integration

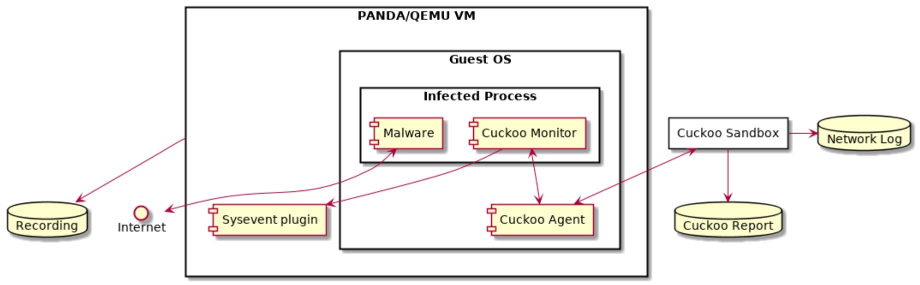


Figure 5.3: Components during the capture process

Chapter 6

Experimental Validation

6.1 Goals and challenges

In the following experiments we aim to assess the effectiveness of SysTaint in helping security researchers in the process of studying a malware sample. Since using SysTaint requires manual inputs and involves interactively exploring the collected data, we tested it against a small number of real-world malware samples. Reverse engineering all the functionalities of the samples in consideration is outside of the outside of the scope of this work, we focus instead on three points:

- Checking the effectiveness of our heuristics in detecting the cryptographic functions used by the malware process
- Checking that it is possible to retrieve the unencrypted messages the malware sent and received from the network
- Checking that the contents of the plaintext messages to be sent through the network are correctly annotated with their provenance

The main difficulty in performing these experiments with real-world malware samples lies in the absence of a ground truth in the form of source code or documentation. This is especially true with regards to communications with the C&C server, which the publicly available reports rarely describe in detail. The lack of a ground truth makes it particularly hard to check the correctness of the data automatically gathered by our tool, especially if the sample employs a custom binary format or obfuscation techniques. To address the lack of ground truth, we included in our tests a sample of *Zeus*, a *banking Trojan* whose source code is publicly available[1].

[1] The Zeus source code can currently be retrieved at `https://github.com/Visgean/Zeus/`

6.2 Collection and filtering of candidate samples

For our experiments, we selected four malware samples. The first is the aforementioned *Zeus*. The second is *Citadel*, a close derivative of Zeus whose toolkit, publicly leaked in 2012, we used to build a bot configured to report to a server under our control, in the same manner as we did for Zeus. The third and fourth samples are *Dridex* and *Emotet*, two modular trojans whose samples we downloaded from VirusTotal[2].

We collected an initial set of candidate samples as follows: 1. We took a list of malware families known for exchanging complex data from [27] 2. We looked up analyses of samples from these families on the Internet and discarded the ones known for employing complex evasion techniques (e.g., *Nymaim*, *Trickbot*). 3. We searched VirusTotal and VxStream sandbox for recent samples which were tagged by the users as belonging to one of the selected families. 4. By looking at the report of the analysis by VxStream, we discarded the samples that were not in PE/exe x86 format, to ensure they were compatible with our x86 version of Windows, and those that showed no network activity 5. We downloaded the selected samples from VirusTotal

We chose the *Dridex* and *Emotet* samples among the 47 samples in our dataset because they were the only ones that in our tests produced meaningful communications with some external server, as distinguishable from the network logs, in 10 minutes of execution. We purposely chose a 10 minutes time limit to reduce the analysis time to a minimum.

We checked that a sample produced what we consider a meaningful network activity by manually inspecting its network log by means of the open source program *Wireshark*[3] and ensuring that:

1. The sample was able to establish a TCP connection with an external server
2. The server did not immediately close the connection after receiving few bytes of data
3. If sample made an HTTP request, that the server did not reply with a standard error page
4. The sample did not repeat the same connection procedure with other addresses immediately after the initial one
5. The sample did not immediately try to connect to a port specific to the SMTP protocol in order to send spam

The four samples we obtained are enough for our purposes. However, it is interesting to consider the possible reasons only two of the samples in our initial

[2]VirusTotal `https://www.virustotal.com/` is a web service known for its ability of automatically analyzing a given sample using several different antivirus software and showing a report. VirusTotal also allows selected companies and institutions to download samples given their hash or a specific query.

[3]Wireshark `https://www.wireshark.org/` is a widespread open-source software for inspecting network logs

dataset were able to connect to their C&C, even when let execute for a longer (30 minutes) time window: the samples that did not show any network traffic may have either detected the virtual environment (which cannot be hidden completely), employed arbitrary long timeouts as a measure to evade the analysis, or expected to be launched in a specific way (for example from a Microsoft Word macro). The behavior of the samples that showed network traffic, but failed to connect their C&C, is instead likely due to their C&C being brought down.

6.3 Experiments

6.3.1 Capture and analysis environment

We prepared a Windows7 SP1 x86 virtual machine with 512mb of RAM and 20GB of hard disk. To make it easier for malware to execute, we disabled the firewall, user account control and Windows Defender components of the operating system. Since many malware families are designed to infect the user's computer through malicious code embedded in a Microsoft Word document, and may check for the presence of Microsoft Word in the system, we also installed a copy of Microsoft Office 2007. Network and Internet access was provided to the virtual machine via a *tap device* connected to a virtual bridge, which allowed it to communicate with the host and the Internet.

We aliased the common part of our PANDA invocation to a `preplay` command. We employ the same alias in this document to keep the examples short. The settings are the same for the recording and replay phases, with the exception of the `replay` flag being added and the network device mode being switched from *tap* to *user*.

```
alias preplay="qemu-system-i386 -m 512M \
-hda ~/vms/win7_cuckoo.qcow2 \
-netdev user,id=net1 \
-device rtl8139,netdev=net1,mac=02:29:07:29:9F:41 \
-os windows-32-7 -replay sysrec"
```

6.3.2 Testing encoding function detection

To test our taint tracking implementation and cryptographic function detection, we wrote a small C++ console application making use of cryptographic APIs from various libraries.

For each tested function, we evaluated whether:

- our custom heuristic, *heuristic 1*, can detect it in its entirety or one of its primitives
- the ratio of arithmetic instructions heuristic, *heuristic 2*, can detect one of its primitives
- our taint tracking implementation can correctly detect a dependency of the output buffer on the input buffer with the option *taint on pointer dereference* **enabled**

function	heuristic 1	heuristic 2	taint analyzable
wcsncpy	no	no	yes
utf8Encode	no	no	yes
base64Encode	no	no	dereference
strlen	no	no	no
CRC32	no	no	no
xor with previous byte	yes	no	yes
deflate	no	no	yes
RC4	yes	no	dereference
CryptEncrypt AES	primitives	–	no
TinyAES	yes	primitives	dereference
OpenSSL AES	yes	no	no
OpenSSL SHA256	yes	yes	yes

Table 6.1: Results of the taint-tracking and encryption detection tests

- our taint tracking implementation can correctly detect a dependency of the output buffer on the input buffer with the option *taint on pointer dereference* **disabled**

The test includes both simple transformation functions like base64encode (to test that the data-flow tracking works correctly) and encryption functions.

As we discussed in 5.2.7, our current taint-tracking implementation can fail for functions performing complex bitwise transformations and is only able to track functions employing substitution tables when the *taint on pointer dereference* option is enabled.

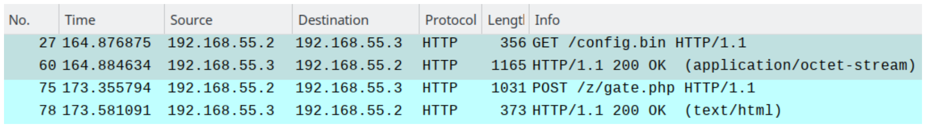
Our custom heuristic proved effective, detecting all the tested cryptographic functions. However, we were not able to obtain the expected results from the heuristic based on the ratio of bitwise arithmetic and regular instructions described in [12] and [9].

It is possible to improve the results mentioned above by refining the heuristics and adding more functions to the test suite.

6.3.3 Analysis of Zeus

Zeus is a banking Trojan whose complete source was publicly leaked in 2011, causing it to become the base for other malware families developed later on. We built the Zeus toolkit from the publicly available source code and then used it to obtain an executable sample configured to report to a server under our control, installed on a second virtual machine attached to the same virtual bridge.

We ran our Zeus sample via Cuckoo sandbox in the testing environment previously described and obtained a recording covering approximately 10 minutes of execution and comprising 32,811 million instructions. By inspecting Cuckoo Sandbox's report, we can notice the presence of the HTTP calls reporting to our servers, and infer that Cuckoo managed to correctly track the sample's actions, following the malware as it infected other processes and communicated with its



No.	Time	Source	Destination	Protocol	Length	Info
27	164.876875	192.168.55.2	192.168.55.3	HTTP	356	GET /config.bin HTTP/1.1
60	164.884634	192.168.55.3	192.168.55.2	HTTP	1165	HTTP/1.1 200 OK (application/octet-stream)
75	173.355794	192.168.55.2	192.168.55.3	HTTP	1031	POST /z/gate.php HTTP/1.1
78	173.581091	192.168.55.3	192.168.55.2	HTTP	373	HTTP/1.1 200 OK (text/html)

Figure 6.1: The relevant HTTP requests made by the Zeus sample. The first one sends to no data to the server and retrieves a `config.bin` file, the second one sends an encrypted report to the `gate.php` endpoint.

C&C. We had configured the recording process so that the *sysevent* plugin automatically generates a "pids.txt" file containing, for each process monitored by Cuckoo, the time of the first and last received hypercall, the process identifier, the process name, and the address space identifier.

The next step consists in gathering information about each process of interested. To this purpose we ran the *procinfodump* PANDA plugin, which takes as input the previously generated pids.txt file. *Procinfodump* works by replaying the execution and, when the point in time corresponding to the last hypercall of a process is reached, executing a *Rekall* script that records the layout and contents of the address space of each process, together with the list of functions exported by all DLLs the process loaded in memory.

```
preplay -panda "procinfodump:file=pids.txt"
```

By quickly scanning the Cuckoo behavioral log, filtering the network calls, and aggregating them by process, we then identified the processes communicating through the network.

```
lsass.exe 0
uqim.exe 1 WSAStartup
dwm.exe 1 WSAStartup
taskhost.exe 1 WSAStartup
conhost.exe 1 WSAStartup
rundll32.exe 1 WSAStartup
SearchProtocolHost.exe 1 WSAStartup
cmd.exe 1 WSAStartup
SearchProtocolHost.exe 1 WSAStartup
bot.exe 2 WSAStartup
explorer.exe 374 getaddrinfo,shutdown,GetBestInterfaceEx,
    HttpOpenRequestA,WSARecv,WSAStartup,WSASocketW,getsockname,
    GetAdaptersAddresses,InternetCloseHandle,WSASend,
    InternetConnectA,InternetQueryOptionA,listen,InternetReadFile,
    ObtainUserAgentString,HttpQueryInfoA,closesocket,
    InternetSetOptionA,GetAddrInfoW,setsockopt,HttpSendRequestA,
    socket,bind,InternetOpenA,InternetCrackUrlA
```

Listing 6.1: Processes in the Zeus recording, with number and kind of network calls

From this data we can infer that Zeus infected the explorer.exe process, and performed its activities from there. Scanning the behavioral log, we extracted the entry corresponding to the HTTP request to `/gate.php` that sent data to our server.

```
{
    "category": "network",
    "status": 1,
    "stacktrace": [],
    "api": "HttpSendRequestA",
    "return_value": 1,
    "arguments": {
        "request_handle": "0x00cc000c",
        "lpszHeaders": "",
        "dwOptionalLength": 3897,
        "headers": "",
        "post_data": "P\u0007\u00e4\u00e0@\u00few\u0018\u00a1g04\
    u00e6\u001c\u00d04\u00e4W\u00f6\u00e5~\u008e\u00bf9K\u00e0\
    u001d\u00bf\u00c3J(\u00d2BR\u00daT\u00146\u00c9\u00ea\u00a2\
    u008f\u0003JG\u001dew\u00e3&\u00a5M'S\u00d94SX [...]",
        "lpOptional": "0x05c10658",
        "dwHeadersLength": 0
    },
    "time": 1511959763.1944,
    "tid": 2680,
    "flags": {},
    "uniqhash": 2680025796
},
```

Listing 6.2: Entry in cuckoo behavioral log corresponding to the encrypted payload being sent to `/gate.php`

We then performed the automated detection of cryptographic functions, whose first step consists in running the *fnmemlogger* plugin to gather statistics about every function called by the sample. A Python script is then used to analyze the output of *fnmemlogger* and use heuristics to extract a set of functions classified as cryptographic.

```
preplay -panda "fn_memlogger:asids=227745792"
```

In the explorer.exe process that we identified earlier, our heuristic detected as cryptographic 6 functions that belong to memory regions which are executable but not mapped to any file, and thus likely to be regions where the sample injected the malicious code.

```
0x23da81d privateXRW_0x23c0000+0x1a81d
0x37b0050 privateXRW_0x37b0000+0x50
0x23d94d1 privateXRW_0x23c0000+0x194d1
0x23e481e privateXRW_0x23c0000+0x2481e
0x23daa69 privateXRW_0x23c0000+0x1aa69
0x23da8ea privateXRW_0x23c0000+0x1a8ea
```

Listing 6.3: Addresses of cryptographic functions detected in the code Zeus injected in explorer.exe. On the right, the same addresses are shown with as offset with respect to their memory region. “privateXRW” means the region is not mapped to a file and it is executable, readable and writable

By manually inspecting the code of these functions with an interactive disassembler we could identify `0x23da8ea` as the RC4[4] encryption function, and `23da81d` as a CRC32[5] implementation.

To these six functions we added a list of known cryptographic functions in the Windows' libraries. To obtain their addresses, we looked up their names on the list of functions exported by the loaded libraries that we obtained earlier with the *procinfodump* plugin. We then wrote the list of the addresses of known and detected cryptographic function to a configuration file, which is then fed to the *systaint* plugin, performing the data collection phase of the analysis.

```
preplay -panda 'systaint:cfg=config.json'
```

We then parsed and inspected the logged data by means of python scripts. We employed the Jupyter[6] interactive python environment to quickly run commands and check the results.

We then queried the data collected by SysTaint to find the syscalls involved in the HTTP request to `/gate.php` that we have identified before. We could find, in particular, the syscalls responsible for sending the content body of the HTTP request through the network, by filtering the logged syscalls by tag (the "uniqhash" we noted before) and sorting them by the size of the largest buffer they read.

```
id, largest buffer read, name
17135175, 3865, NtDeviceIoControlFile
17134992, 243, NtDeviceIoControlFile
17142006, 214, NtCreateFile
17141426, 138, NtOpenKeyEx
17131926, 134, NtOpenKeyEx
```

Listing 6.4: Syscalls belonging to the `HttpSendRequestA` api call that read the largest buffers

Once we found the syscall of interest, we could inspect the memory buffers it read, along with the taint label indicating their provenance.

```
1
READ 0x5c10658(Heap+0x658), len=1
>> 'P'
DEPS: 17109175[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)],17109113[0
    x23daa69(privateXRW_0x23c0000+0x1aa69)],2982974[NtReadFile]

[...]

8
READ 0x5c10676(Heap+0x676), len=2
>> '(\xd2'
```

[4]RC4 or Rivest Cipher 4 is a fast encryption function, now considered outdated, which is easy to implement in software

[5]CRC32 is the 32bit variant of the cyclic redundancy check algorithm, which is used to compute a 32-bit number (checksum) from a given chunk of data to quickly check for transmission errors

[6]`http://jupyter.org/`

```
DEPS: 17109175[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)],17109113[0
    x23daa69(privateXRW_0x23c0000+0x1aa69)],17093091[0x23da8ea(
    privateXRW_0x23c0000+0x1a8ea)],17087951[0x23da8ea(
    privateXRW_0x23c0000+0x1a8ea)],2467394[NtQueryDirectoryFile
    ],2982974[NtReadFile],17092734[NtReadFile]

9
READ 0x5c10678(Heap+0x678), len=3865
>> 'BR\xdaT\x146\xc9\xea\xa2\x8f\x03[...]'
DEPS: 17109175[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)],17109113[0
    x23daa69(privateXRW_0x23c0000+0x1aa69)],17109023[CPGetHashParam
    ],17108746[CPHashData],17093091[0x23da8ea(privateXRW_0x23c0000
    +0x1a8ea)],17087951[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)
    ],17062235[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)],2165326[
    UuidCreate],1852044[0x68aa69(Stack+0x1aa69)],unregistered,
    unregistered,2467000[NtCreateFile],2467394[NtQueryDirectoryFile
    ],unregistered,2982974[NtReadFile],17092734[NtReadFile]

[...]

14
READ 0x5dbf404(Stack+0x3f404), len=8
>> '\x03\x01\x00\x00\x83\x10>\x02'

15
READ 0x5dbf478(Stack+0x3f478), len=8
>> '9\x0f\x00\x00X\x06\xc1\x05'
```

Listing 6.5: Buffers read from the NtDeviceIoControlFile call sending the encrypted body of the request. Each buffer is annotated with a list of taint labels, which are the identifiers of functions and system calls that handled that data. After each label, between square brackets there is the resolved name of the associated function or system call. Since we configured SysTaint not to overwrite existing taint labels on encryption/encoding events, encrypted data is tainted with several labels.

We noticed that all the data being sent was processed by the encryption function call with label 17109175, which was detected earlier by our heuristic and that we manually identified as RC4. Repeating the process with its input buffers, we noticed more encrypted data, as reported below:

```
0x23da8ea(privateXRW_0x23c0000+0x1a8ea) []

0
READ 0x5c10658(Heap+0x658), len=20
>> '\x0c\xd0\x19\xe2]S\xa1\x0e`a@\xd3\xde\xef|W\x1ai\xfb;'
DEPS: 17109113[0x23daa69(privateXRW_0x23c0000+0x1aa69)],2982974[
    NtReadFile]

[...]

3
READ 0x5c10678(Heap+0x678), len=3865
>> '"pN\x84\xbe\xa4\xd0\xa5\x18\xeb\'\xfc\x83d\xad\xb8\xa9\x8e\x8e\
    x8e\x8e\x8e\x8e\x8e\x96\x96\x96\x96\x8e\x8e\x8e\x8e\xc[...]'
DEPS: 17109113[0x23daa69(privateXRW_0x23c0000+0x1aa69)],17109023[
    CPGetHashParam][...],2467394[NtQueryDirectoryFile],unregistered
    ,2982974[NtReadFile],17092734[NtReadFile]

4
READ 0x5dbf74c(Stack+0x3f74c), len=8
>> '\x82\xf6=\x02`\xf7\xdb\x05'

[...]
```

Listing 6.6: Obfuscated inputs to RC4 in Zeus

Among the dependencies of these buffers there was no function call writing a significant amount of data. Since SysTaint is configured to record all the functions handling tainted data, and the buffer contents are tainted, we are sure the function producing this data has been recorded has been recorded, however, since it has not been detected as an encryption call, it does not have a taint label of its own, and does not explicitly appear as a dependency. By scanning the collected data for the last function writing to the buffer at 0x5c10678, we were able to find the function call we were looking for (which has address `0x23df4ff`) and the buffer with plaintext information.

```
0x23df4ff(privateXRW_0x23c0000+0x1f4ff) []

0
READ 0x5c10658(Heap+0x658), len=20
>> '\x0c\xdc\xc9\xfb\xbf\x0e\xf2\xafn\x01!\x93\r1\x93+Ms\x92\xc0'
DEPS: 17109113[0x23daa69(privateXRW_0x23c0000+0x1aa69)],2982974[
    NtReadFile]

[...]

4
READ 0x5c10698(Heap+0x698), len=24
>> 'JOHN-PC_E532648A058CCAAC'
DEPS: 17062235[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)]

[...]

13
READ 0x5c10738(Heap+0x738), len=20
>> 'C:\\Windows\\system32\\'
DEPS: 17093091[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)],17087951[0
    x23da8ea(privateXRW_0x23c0000+0x1a8ea)],17092734[NtReadFile]

[...]

16
READ 0x5c10763(Heap+0x763), len=12
>> 'John-PC\\John'
DEPS: 17093091[0x23da8ea(privateXRW_0x23c0000+0x1a8ea)],17087951[0
    x23da8ea(privateXRW_0x23c0000+0x1a8ea)],17092734[NtReadFile]

[...]
```

Listing 6.7: Plaintext inputs to the undetected obfuscation function in Zeus

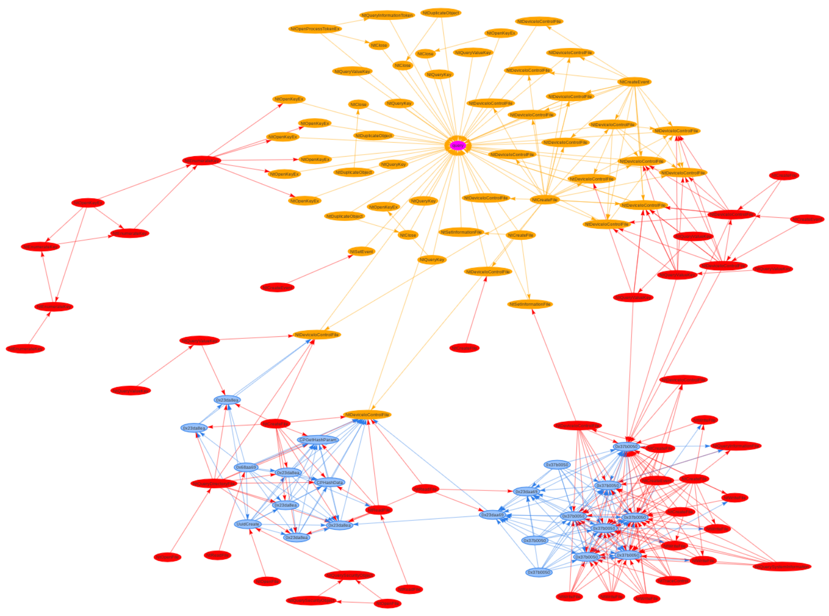


Figure 6.2: Visual outline of the dependency graph of the HTTP call to gate.php in Zeus via `HttpSendRequestA`. Edges represent data dependencies, yellow nodes represent the syscalls intercepted during the call to the function `HttpSendRequestA`, red nodes represent syscalls recorded before the call, finally blue nodes represent the detected encryption calls.

Looking at the plaintext data, we can distinguish a list of processes, some information about the PC, and a list of HTTP cookies. By focusing on the dependencies of the buffers, we can notice a second invocation of RC4, and a `NtReadFile` call. Looking up its tag in the Cuckoo behavioral log (or following the dependency chain of NtReadFile until the NtCreateFile reading the filename is reached), we can find that the original buffer is read from `C:\Users\John\AppData\Roaming\Poweg\kuywl.tmp`. We can then infer that the data being sent has been read from a file (written by another process), decrypted, re-encrypted and sent.

If we inspect the function `0x23df4ff` that encrypted the plaintext data before RC4, by reading the previously obtained memory dump with a disassembler, we find a complex function, which contains a small block of code doing a `xor`-based obfuscation function. Reading the Zeus source code, we can recognize this function as `BinStorage::_pack`, inside of which the call to `visualEncrypt` (which obfuscate a string by performing the xor of each byte with the previous one) has been inlined.

We can also obtain a visual representation of the data-flow leading to the data being sent.

By closely inspecting this dependency graph using a small GUI tool we wrote, and the buffers each syscall reads, it is possible to distinguish the `NtDeviceIo ControlFile` syscall sending the header of the HTTP request, having as input the IP address of the host, which is taken from a registry key holding the configuration, as well as the previously examined and the `NtDeviceIo ControlFile` syscall sending the body of the POST request.

It is worth noting that finding the plaintext buffers (and by extension, encryption functions) is quicker and easier if we know in advance a part of the plain text. In Zeus, for example, it is possible to locate the same plaintext buffer we have seen earlier by looking for large buffers containing the name of the PC ("John-PC" in our case).

6.3.4 Analysis of Citadel

The second sample we analyzed is Citadel, which, as Zeus, we built from a leaked toolkit for which, however, the source code is not available. Citadel is a close descendant of Zeus, so we can expect many similarities in the way it works.

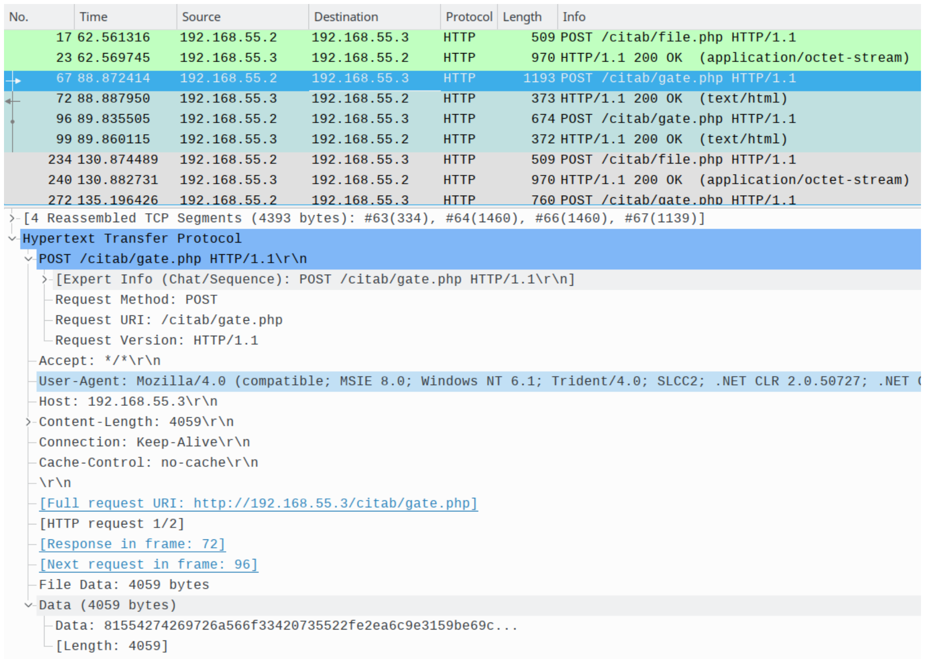
Looking at the network log with Wireshark, we could distinguish two HTTP calls that repeat periodically with a short interval due to the timeouts we set when we configured it. The first one is a call to a `file.php` to download a `config.dll` file, the second one is a call to `gate.php`, which is the endpoint our sample uses for reporting to the server. We exported the body of these requests and responses to file, then searched Cuckoo's behavioral log for their contents, obtaining the ID of the process (also explorer.exe), and a list of call IDs we can inspect with SysTaint in the later phases.

As we did with Zeus, we ran *fnmemlogger* and the encryption function detection scripts, which identified 11 functions in the executable memory regions of the sample, then ran *systaint* to obtain the data for the interactive querying phase.

We then checked the request to `gate.php`, which was encrypted in the same way as it was in the Zeus sample and had similar contents, and the request to `file.php` which instead contained the name of the file to retrieve (`config.dll`), the constant string "C1F20D2340B519056A7D89B7DF4B0FFF", some short integer identifiers, and the MD5 hash of the same contents.

6.3.5 Analysis of Dridex

Differently from Zeus and Citadel, the Dridex sample did not produce network activity when analyzed through Cuckoo, however, we could observe network activity when starting the sample by double clicking on the executable. We can speculate that this behavior is due to Dridex being able to detect the Cuckoo processes or their instrumentation. We therefore obtained a recording by starting the sample manually, and obtained a network log through Wireshark. Since the recording was not performed through Cuckoo, in this analysis we could not



```
No.    Time         Source         Destination    Protocol Length Info
    17 62.561316    192.168.55.2   192.168.55.3   HTTP         509 POST /citab/file.php HTTP/1.1
    23 62.569745    192.168.55.3   192.168.55.2   HTTP         970 HTTP/1.1 200 OK  (application/octet-stream)
    67 88.872414    192.168.55.2   192.168.55.3   HTTP        1193 POST /citab/gate.php HTTP/1.1
    72 88.887950    192.168.55.3   192.168.55.2   HTTP         373 HTTP/1.1 200 OK  (text/html)
    96 89.835505    192.168.55.2   192.168.55.3   HTTP         674 POST /citab/gate.php HTTP/1.1
    99 89.860115    192.168.55.3   192.168.55.2   HTTP         372 HTTP/1.1 200 OK  (text/html)
   234 130.874489   192.168.55.2   192.168.55.3   HTTP         509 POST /citab/file.php HTTP/1.1
   240 130.882731   192.168.55.3   192.168.55.2   HTTP         970 HTTP/1.1 200 OK  (application/octet-stream)
   272 135.196426   192.168.55.2   192.168.55.3   HTTP         760 POST /citab/gate.php HTTP/1.1
```

```
> [4 Reassembled TCP Segments (4393 bytes): #63(334), #64(1460), #66(1460), #67(1139)]
v Hypertext Transfer Protocol
  v POST /citab/gate.php HTTP/1.1\r\n
    > [Expert Info (Chat/Sequence): POST /citab/gate.php HTTP/1.1\r\n]
      Request Method: POST
      Request URI: /citab/gate.php
      Request Version: HTTP/1.1
    Accept: */*\r\n
    User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET
    Host: 192.168.55.3\r\n
  > Content-Length: 4059\r\n
    Connection: Keep-Alive\r\n
    Cache-Control: no-cache\r\n
    \r\n
    [Full request URI: http://192.168.55.3/citab/gate.php]
    [HTTP request 1/2]
    [Response in frame: 72]
    [Next request in frame: 96]
    File Data: 4059 bytes
  v Data (4059 bytes)
      Data: 81554274269726a566f33420735522fe2ea6c9e3159be69c...
      [Length: 4059]
```

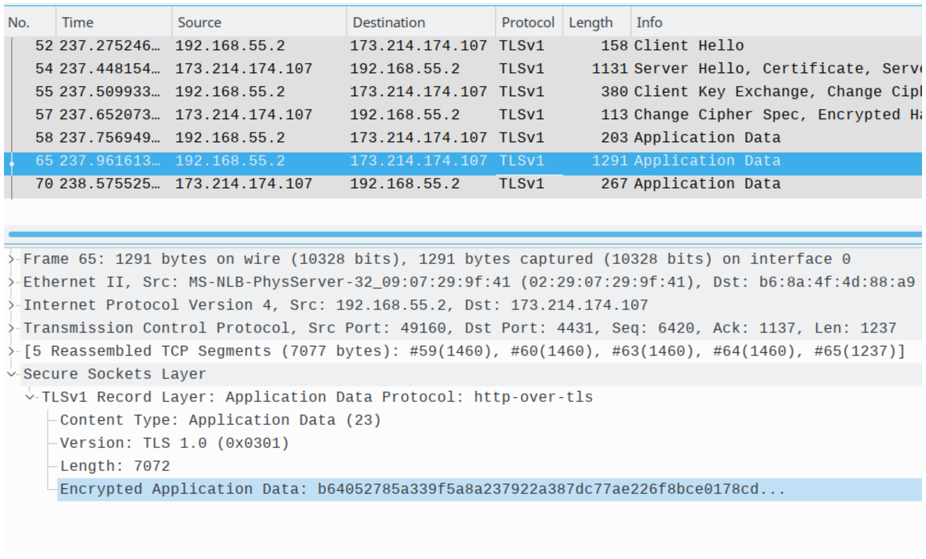
Figure 6.3: The relevant HTTP requests made by Citadel. In the bottom panel, the details of the request to `gate.php` containing the encrypted payload as body

make use of the behavioral log Cuckoo produces and in particular the contained list of high-level APIs called by each process.

Since we do not know anything about the activities of the processes in the execution, the quickest course of action is starting from the network traffic and use the *stringsearch2* plugin to detect the process that generated it. As we did with the other samples, we first gathered information about the running processes by means of the *procinfodump* plugin. Since there were no hypercalls by Cuckoo Monitor, a pids.txt with the list of processes and the time of their last known API call was not automatically generated during the recording by the *sysevent* plugin. We could however generate a pids.txt file by using another plugin, *asidstory*, included in PANDA, which outputs basic information for every process in the recording, including the time of their last executed instruction.

Network traffic analysis Our Dridex sample did not resolve any address through DNS, but made four different TLS connections to different IP addresses. By inspecting the connections with Wireshark, it is possible to see the details of the TLS connections, with the encrypted payload being transmitted. The first of these connections had no reply sent back from the server. The second one contained reasonably long requests and response, so we decided to focus on it.

From Wireshark, we then extracted the two payloads sent and the one re-



```
No.  Time           Source           Destination      Protocol Length Info
  52 237.275246…  192.168.55.2     173.214.174.107  TLSv1      158 Client Hello
  54 237.448154…  173.214.174.107  192.168.55.2     TLSv1     1131 Server Hello, Certificate, Serv
  55 237.509933…  192.168.55.2     173.214.174.107  TLSv1      380 Client Key Exchange, Change Cip
  57 237.652073…  173.214.174.107  192.168.55.2     TLSv1      113 Change Cipher Spec, Encrypted H
  58 237.756949…  192.168.55.2     173.214.174.107  TLSv1      203 Application Data
  65 237.961613…  192.168.55.2     173.214.174.107  TLSv1     1291 Application Data
  70 238.575525…  173.214.174.107  192.168.55.2     TLSv1      267 Application Data

> Frame 65: 1291 bytes on wire (10328 bits), 1291 bytes captured (10328 bits) on interface 0
> Ethernet II, Src: MS-NLB-PhysServer-32_09:07:29:9f:41 (02:29:07:29:9f:41), Dst: b6:8a:4f:4d:88:a9
> Internet Protocol Version 4, Src: 192.168.55.2, Dst: 173.214.174.107
> Transmission Control Protocol, Src Port: 49160, Dst Port: 4431, Seq: 6420, Ack: 1137, Len: 1237
> [5 Reassembled TCP Segments (7077 bytes): #59(1460), #60(1460), #63(1460), #64(1460), #65(1237)]
v Secure Sockets Layer
  v TLSv1 Record Layer: Application Data Protocol: http-over-tls
      Content Type: Application Data (23)
      Version: TLS 1.0 (0x0301)
      Length: 7072
      Encrypted Application Data: b64052785a339f5a8a237922a387dc77ae226f8bce0178cd...
```

Figure 6.4: Contents of the second TLS conversation between Dridex and an external server

ceived, putting them in a "http" folder. We then used the *stringsearch2* plugin to look for memory accesses involving these payloads, setting a chunk length of 4 bytes.

```
preplay -panda "stringsearch2:filelist=http,chunklen=4"
```

The *stringsearch2* plugin produces a file containing a list of matches. Each match contains the chunk it refers to, the point in time when the match was detected, and the call stack at that moment. By manually inspecting the matches stringsearch2 found, we could quickly find the relevant process ("svchost8"), as well as some functions in the sample's own code handling the encrypted data. We noticed that some of the functions found belong to `bcryptprimitives.dll`, which contains code for some Windows encryption APIs.

Once we had the list of processes, we ran *fnmemlogger* and the encryption function detection scripts. The script reported, in addition to the already mentioned Windows cryptographic APIs, 33 functions in the sample's own code. After running the full analysis, we can search the collected data for the function producing the two sent TLS payload and reading the received one.

The first payload is 144 bytes long. Looking for functions producing its contents, we find "bcryptprimitives.dll+0x4b09", which appears to encrypt the header of a HTTP post request.

The second payload is 7072 bytes long and is also encrypted by "bcryptprimitives.dll+0x4b09". That function reads data from a long buffer, which is unfortunately further encrypted. We can infer Dridex is sending its payload encrypted, inside the body of a HTTP request, the same way Zeus and Citadel did.

This buffer does not show any functions previously detected as cryptographic among its dependencies, but SysTaint detected it as composed of several smaller buffers having different dependencies.

```
[...]

18
READ 0x3e1761(Heap+0x21761), len=3
>> '\x06\x94_'
DEPS: 2877057[0x755d4b09(bcryptprimitives.dll+0x4b09)],2828033[
    CPVerifySignature],2827857[CryptDecodeObjectEx],2827757[
    CryptDecodeObjectEx],2745264[NtReadFile]

[...]

42
READ 0x45528d(Heap+0x9528d), len=32
>> '\xaci\xcbDE\xd6u\xc5\xfe\x1ee\x9f\n\xe8\x1c\x88ll\xd5\xc5\\C\
    xcf\xcf9)\xc8(\x9eW\xf7\xfd'
DEPS: 408768[NtQueryValueKey]

43
READ 0x4552ad(Heap+0x952ad), len=2
>> 'mo'

44
READ 0x4552af(Heap+0x952af), len=14
>> '\xa7\xb3Z\xb5z\xa6\xa6\xd2?\xeb\xfeZ\x1ft'
DEPS: 411577[NtQueryValueKey]

45
READ 0x4552bd(Heap+0x952bd), len=2
>> '\xd8\xaf'

46
READ 0x4552bf(Heap+0x952bf), len=22
>> '/\xd7q\xdc\x97G@\xbe\x19\xa7\xd3\xab\xb1\xda;\x05u\x12\xd5\xc8\
    xd9m'
DEPS: 430074[NtQueryValueKey]
```

Listing 6.8: Encrypted body of the HTTP request sent by Dridex

The function encrypting this payload was not automatically detected. However, we could find it by either searching for the functions writing one of the buffers detected above. By inspecting its inputs, we can find a password, presumably used for the encryption and the complete plaintext buffer containing a list of installed programs, the output of the command "whoami.exe /all" and several other information about the system.

Applying the same procedure for the response TLS payload, we were able to access the corresponding plaintext buffer. To our surprise, the reply was an HTTP 403 ("Forbidden") error message from the Nginx webserver. This may mean that our Dridex sample was ultimately not able to reach its C&C server.

6.3.6 Analysis of Emotet

We recorded the execution of the Emotet sample via Cuckoo without encountering any particular problem. However, proceeding with the analysis, we noticed that Cuckoo was not able to detect and track all the processes where the malware injected itself, and that the relevant processes were absent from its logs. As a result we had to proceed as we did with Dridex.

Our Emotet sample makes several HTTP POST requests to an address belonging to an American hosting provider. The HTTP calls are not themselves encrypted via TLS but, as in the previously analyzed cases, carried an encrypted payload. The first HTTP call downloaded a 1147 KB payload, which we supposed was an updated version of the malware executable. The second HTTP call retrieved a smaller 4692 bytes payload, likely a configuration file. After receiving the two payloads, the sample attempted to contact several SMTP servers to send spam emails.

By running *stringsearch2* to identify the processes of interest, we discovered that our implementation was not able to detect the memory accesses moving the received data between the TCP buffers in kernel space and the buffers in user space. As a consequence, the *systaint* plugin was not able to detect the syscalls receiving the network payloads and include them in the taint analysis. This in turn caused all the functions processing the received data not to be logged. We are still unsure why this was the case, but we decided to leave the problem for a later time.

To inspect the data the sample sent in the first HTTP call, we proceeded as with the previous samples. The data being sent as body of the HTTP was 340 bytes long and was encrypted by `CryptEncrypt`, from Window's cryptographic APIs. Inspecting the inputs to `CryptEncrypt`, we found more encrypted data, which is a sign of a second, undetected, layer of encryption. We found the second encryption function by getting the last function that wrote in the buffer read by CryptEncrypt.

We then inspected the second payload retrieved from the network, which we supposed contained some configuration data. Not having the network data properly tainted, we had to rely on the *stringsearch2* plugin to find user-space functions reading the data we were looking for.

The *stringsearch2* plugin indicates that the data is processed by the undetected user-space function `0x1cd2000`, which is in a private memory region not mapped to any file. Dumping that memory region we found a DLL, which we identified, by running `strings`, as *openssl-1.1.0f*.

We added this function to the list of known encryption functions and re-ran the *systaint* plugin, so that *systaint* could record and taint its output. After *systaint* finished, we inspected the data in input to that function, but noticed it was not a plaintext buffer. We could, however, quickly find another encryption function and the plaintext buffer by taking the functions depending on the one we just found. In that plaintext buffer we could distinguish a list of email addresses and an email template.

6.4 Considerations on the experimental results

6.4.1 Taint tracking

The taint tracking tests performed on our sample application showed good results for all the simpler data-movement functions and for some complex cryptographic functions. As expected, it was not able to reliably track data across functions performing complex bitwise operations as AES, and more surprisingly also the much simpler CRC32.

Despite these imperfections, our taint tracking implementation was very effective in identifying the dependencies of the buffers we inspected while analyzing the various samples.

6.4.2 Cryptographic function detection

The automated detection of cryptographic functions allowed us to easily deal with at least one level of encryption in all the analyzed samples, speeding up our analysis.

However, in all samples, it failed to identify some encryption function in the data-flow path we were interested in, forcing us to find the functions of interest by scanning the list of logged functions looking for the ones producing a given sequence of bytes or writing in a given memory area.

In the case of Zeus and Citadel, this was due to the “visualEncrypt” function being inlined in the calling function, which was rightfully not identified as encryption function. With Dridex the case appears similar, with the encrypted data being written by a function that does not, by itself, perform encryption, but obfuscates the data. We could not determine the reason in the case of Emotet, since during the analysis, by employing *stringsearch2*, we identified a wrapper for the encryption function and not the encryption function itself. The fact *stringsearch2* did not detect the plaintext being read by the encryption function itself may be due the encryption function not reading the input bytes in sequential order.

6.4.3 Insights on the usage of network communications

In all cases it was possible to navigate the execution and locate the plaintext buffers, whose contents were correctly annotated with their provenance.

It is worth noting, however, that not all the contents of the inspected buffers were annotated. This is expected, and we can classify the data portions without provenance annotations as: 1. constant strings copied from the memory regions where the malware program unpacked itself 2. field separators and other constant parts of the protocol 3. small integers, with a specific meaning, that were not obtained by direct transformation of tainted data. Although understanding the contents of the fields in this third category is valuable to understand the protocol, unfortunately, to keep the resource requirements acceptable, SysTaint limits the use of taint tracking to the data obtained via syscalls or produced by cryptographic functions and does not use taint tracking on the data being produced by the other functions executed.

6.4.4 Performance

The longest part of the analysis is the execution of the *systaint* plugin, because of the heavyweight taint instrumentation. We did not focus on optimizing the analysis time, but believe that it is possible to significantly reduced it by employing some simple measures, like disabling taint instrumentation on the processes that are not being tracked.

sample	instructions	execution time	analysis time
Zeus	32,811 M	10 minutes	7 hours
Citadel	12.414 M	4 minutes	2 hours
Dridex	6,853 M	10 minutes	2 hours
Emotet	21,270 M	4 minutes	4 hours

Table 6.2: Time spent for the *SysTaint* analysis of each sample. Notice the analysis time of Dridex is, in proportion, lower than that of the other samples because of the absence of Cuckoo Agent and Cuckoo Monitor from the recording

Memory usage is a bigger concern, as it depends on the amount of addresses that have been tainted. This is, in particular, an issue when the option *taint on pointer dereference*, is enabled, which may lead in some particular cases to a large amount of addresses being tainted and a very high memory usage. These cases can be easily handled manually or automatically by blacklisting the affected label. Even with this option enabled, however, in our tests the ram usage stayed under 8GB.

Chapter 7

Limitations

7.1 Tracking data not obtained through syscalls

By design, SysTaint only applies taint labels to the data obtained through syscalls or output by cryptographic functions, to keep the RAM and disk usage acceptable. Unfortunately, as discussed in subsection 6.4.3 this makes it impossible to automatically track the provenance of data that is not directly derived from the output of a syscall or a previously detected encryption function.

7.2 Code employing virtual machines

The described approach assumes that the code under analysis follows some of the semantics defined by the C language. In particular, we assume that the code of the sample is partitioned in functions that we can identify by their address in memory.

These assumptions hold for both the components of the operating system and for the majority of the malware families, which are compiled to native x86 code, however, they are violated by code that is meant to be interpreted at runtime or executed by a virtual machine. This is the case with higher level languages like Java, C# or Python, and for some complex packing techniques. Additionally, since we rely on the current thread identifier to maintain the shadow stack and distinguishing concurrent activities, *coroutines* and other facilities to achieve concurrency without the use of the operating system threads are also a concern.

When the sample employs interpreters, virtual machines, or coroutines, SysTaint is still able to track the dependencies between system calls, but, not being able to identify the code being executed, SysTaint is not useful to reverse engineer it.

7.3 Limits to the applicability

Our approach shares some of the limitations of sandbox analysis. In particular, it requires the sample to manifest its behaviors in a limited time window. As we discussed in 6.2 in some cases this may be a problem and may require patching the sample to go around its anti-analysis measures.

An excessive number of API calls in the recording, which are sometimes generated on purpose as an evasion technique, can also be an obstacle to the analysis, as it causes both the logs and the analysis time to grow long. As described in [21] Nymaim is an example of malware employing *Win32 API Hammering* as an evasion technique.

Malware specimens that only contact the C&C server after performing some time or CPU-consuming activities may also be more difficult to analyze, as a large number of instructions have to be instrumented and processed during the analysis. Although these parts of the recording can in theory be excluded from the analysis, as described in subsection 5.2.8 this would involve stopping taint-propagation and thus losing data-flow information on the data obtained or copied during the excluded part.

Chapter 8

Future works

8.1 Narrowing the semantic gap

SysTaint keeps track of which functions or system call processed a certain piece of data, builds a dependency graph and allows the analyst to manually query it. It is possible, as a future development, to use the output of SysTaint to automatically detect and highlight interesting data relationships and behaviors, helping the analyst find the relevant parts of the sample's behavior, especially when studying complex scenarios involving multiple processes. Moreover, automatically inferring semantic information about the functions in the sample's code from the data they access, or from the system functionalities they use, would greatly help the analyst to reverse engineer the sample's code.

8.2 Automated decryption of network traffic

Aiding an analyst in the mostly manual process of studying how a malware communicates with its C&C server has been the main motivation of this study. However, since SysTaint is easy to deploy alongside existing sandbox solutions, it is possible to employ it to run an automatic analysis with the purpose of detecting and reporting the contents of encrypted communications. To do so it is necessary to first locate the relevant encrypted traffic in the network traces and then analyze the replay in order to find the points in the execution when it is encrypted or decrypted, in a manner similar to the manual process we demonstrated in section 6.3

8.3 Integration with other software

Allowing the collected information to be accessed from commonly used reverse engineering environments like *IDA Pro* and *Radare2* would allow interesting features, as being able to rapidly check the data (with its provenance) a given function accesses.

One of the possibilities is using the data collected by SysTaint to expose an interactive debugging interface. The usual per-instruction debugging interface could be offered by using the Unicorn Engine[1] to emulate the execution of the individual functions in the sample using the data that SysTaint collected from any of the function's recorded invocations.

A different approach to achieve "time-traveling debugging" could be adding checkpointing and/or reverse-execution capabilities to the PANDA platform. This would allow jumping to a specific point in time, interactively setting hardware breakpoints, and making use of QEMU's integrated GDB stub, using the data collected by SysTaint as an index.

8.4 Automated protocol reverse engineering

It could be useful to integrate in SysTaint some techniques specific to automated protocol reverse engineering as described in [6, 10, 9]. This would allow, in addition to help the analyst understand the source and the usage of the data the malware sent and received, automatically inferring the format of the data in the buffer.

To apply some of the mentioned techniques, it is necessary to apply different taint labels to each byte in the received messages and to make use of dynamic slicing, logging all executed instructions and their operands. To keep the amount of collected data manageable and keep the tool practical to use to study multiple processes and long executions, we decided not to provide these functionalities. It may however be possible to efficiently generate the required data from the function-grained data SysTaint already collects, by emulating the individual functions involved by means of the Unicorn Engine.

[1]`https://github.com/unicorn-engine/unicorn`

Chapter 9

Conclusions

The purpose of this work was building a tool to help the analyst in the malware reverse engineering process, addressing, in particular, the problems arising when debugging and studying malware communicating with external network resources. We proposed an approach that consists in collecting in-depth data about the execution of the sample, enriched with data-flow and call stack information, so that an analyst can interactively query it to learn useful information about the sample's internal functions and data flow.

We implemented this tool, SysTaint, on top of the PANDA analysis platform. We tested SysTaint against four real-world malware samples and it proved effective in extracting useful information about their internal functions, find the unencrypted data being sent through the network and understand its provenance.

Future research can improve our approach by developing ways to automatically infer semantic information from the collected data, and by integrating the existing research on automated protocol reverse engineering and function identification. We are confident that the approach we presented can help the analysts in their reverse-engineering efforts, and that future developments can refine SysTaint into a valuable addition to the malware analyst's toolbox.

Bibliography

[1] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track.* Vol. 41. 2005, p. 46.

[2] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Acm sigplan notices.* Vol. 40. 6. ACM. 2005, pp. 190–200.

[3] Brendan Dolan-Gavitt. "The VAD tree: A process-eye view of physical memory". In: *Digital Investigation* 4 (2007), pp. 62–64.

[4] T. Holz, F. Freiling, and C. Willems. "Toward Automated Dynamic Malware Analysis Using CWSandbox". In: *IEEE Security & Privacy* 5 (Mar. 2007), pp. 32–39. ISSN: 1540-7993. DOI: 10.1109/MSP.2007.45. URL: doi.ieeecomputersociety.org/10.1109/MSP.2007.45.

[5] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices.* Vol. 42. 6. ACM. 2007, pp. 89–100.

[6] Zhiqiang Lin et al. "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution." In: *NDSS.* Vol. 8. 2008, pp. 1–15.

[7] Noé Lutz. "Towards revealing attacker's intent by automatically decrypting network traffic". In: (2008).

[8] Dawn Song et al. "BitBlaze: A new approach to computer security via binary analysis". In: *International Conference on Information Systems Security.* Springer. 2008, pp. 1–25.

[9] Juan Caballero et al. "Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering". In: *Proceedings of the 16th ACM conference on Computer and communications security.* ACM. 2009, pp. 621–634.

[10] Zhi Wang et al. "ReFormat: Automatic reverse engineering of encrypted messages". In: *European Symposium on Research in Computer Security.* Springer. 2009, pp. 200–215.

[11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems". In: *ACM SIGPLAN Notices* 46.3 (2011), pp. 265–278.

[12] Felix Gröbert, Carsten Willems, and Thorsten Holz. "Automated identification of cryptographic primitives in binary programs". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2011, pp. 41–60.

[13] Michael Sikorski and Andrew Honig. *Practical Malware Analysis*. No Starch Press, Feb. 2012. ISBN: 978-1-59327-290-6.

[14] Brendan Dolan-Gavitt et al. "Tappan zee (north) bridge: mining memory accesses for introspection". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 839–850.

[15] Ruoyu Wang et al. "Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services." In: *USENIX Security Symposium*. 2013, pp. 687–702.

[16] X. Li, X. Wang, and W. Chang. "CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution". In: *IEEE Transactions on Dependable and Secure Computing* 11.2 (Apr. 2014), pp. 101–114. ISSN: 1545-5971. DOI: `10.1109/TDSC.2012.83`.

[17] Roberto Paleari. *Introducing QTrace, a "zero knowledge" system call tracer*. Mar. 2014. URL: `http://roberto.greyhats.it/2014/03/qtrace-part1.html` (visited on 03/11/2018).

[18] Tomer Teller and Adi Hayon. "Enhancing automated malware analysis machines with memory analysis". 2014.

[19] Brendan Dolan-Gavitt et al. "Repeatable reverse engineering with PANDA". In: *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. ACM. 2015, p. 4.

[20] George Hotz. *qira*. 2016. URL: `http://qira.me/` (visited on).

[21] Joe Security. *Automated Malware Analysis - Nymaim - evading Sandboxes with API hammering*. Apr. 2016. URL: `https://www.joesecurity.org/blog/3660886847485093803` (visited on 03/11/2018).

[22] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[23] Julien Duchêne et al. "State of the art of network protocol reverse engineering tools". In: *Journal of Computer Virology and Hacking Techniques* (2017), pp. 1–16.

[24] *McAfee Labs Threat Report December 2017*. Dec. 2017. URL: `https://www.mcafee.com/uk/resources/reports/rp-quarterly-threats-dec-2017.pdf` (visited on 03/11/2018).

[25] Microsoft. *Time Travel Debugging in WinDbg.* Sept. 2017. URL: `https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview` (visited on).

[26] Robert O'Callahan et al. "Engineering record and replay for deployability". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference.* USENIX Association. 2017, pp. 377–389.

[27] Jarosław Jedynak. *Mtracker - our take on malware tracking - CERT Polska.* Jan. 2018. URL: `https://www.cert.pl/en/news/single/mtracker-our-take-malware-tracking/` (visited on 03/02/2018).

[28] Baraka D. Sija et al. "A Survey of Automatic Protocol Reverse Engineering Approaches, Methods, and Tools on the Inputs and Outputs View". In: *Security and Communication Networks* (2018).

[29] *Cuckoo Sandbox.* URL: `https://cuckoosandbox.org/` (visited on 03/11/2018).

[30] *Rekall.* URL: `http://www.rekall-forensic.com/` (visited on 03/11/2018).

[31] *The Volatility Foundation - Open Source Memory Forensics.* URL: `http://www.volatilityfoundation.org/` (visited on 03/11/2018).

[32] *VxStream Sandbox.* URL: `https://www.payload-security.com/products/vxstream-sandbox`.

[33] *YARA - The pattern matching swiss knife for malware researchers.* URL: `https://virustotal.github.io/yara/` (visited on).

[34] *Zer0m0n.* URL: `https://github.com/conix-security/zer0m0n` (visited on 03/11/2018).