FAKULTÄT FÜR MATHEMATIK UND INFORMATIK

DER FERNUNIVERSITÄT IN HAGEN

Master's Thesis in Computer Science

Integrated Honeypot Based Malware Collection and Analysis

Martin Brunner

FAKULTÄT FÜR MATHEMATIK UND INFORMATIK

DER FERNUNIVERSITÄT IN HAGEN

Master's Thesis in Computer Science

Integrated Honeypot Based Malware Collection and Analysis

Integrierte Honeypot basierte Malware Sammlung und Analyse

Author:	Martin Brunner, BSc
Supervisor:	Prof. Dr.-Ing. habil. Herwig Unger
Advisor:	Dipl.-Inf. Daniel Berg
Advisor:	Dr. Sascha Todt
Date:	May 9, 2012

In cooperation with:
Fraunhofer Research Institution for Applied and Integrated Security
Parkring 4, 85748 Garching, Germany

Abstract

Today's most disruptive cyber threats can be attributed to botnet-connected malware. Thus, timely intelligence on emerging trends in the malicious landscape is an essential prerequisite for successful malware defense. This is commonly gained by collection and examination of current real-world attack data and a preferably meticulous analysis of the most recent malware samples. However, the ongoing sophistication of malware led to intensive obfuscation and anti-debugging measures and also resulted in a complex and multi-staged malware execution life-cycle.

In this thesis we present a novel approach for integrated honeypot based malware collection and analysis, which extends the functionalities of existing approaches. Specifically our approach addresses the separation of collection and analysis, the limitations of service emulation and the operational risk of high-interaction honeypots. Our overall goal is to capture and analyze malware at a large-scale while covering the entire execution life-cycle of a given malware. This should happen in a preferably automated fashion within a controlled environment while being able to handle novel malware and the respective command-and-control (C&C) communication as well. Contrary to purely network-based approaches, we aim towards retrieving information about the malware's logics at runtime on a collection and analysis system. Thus we can provide the currently being analyzed malware with all requested resources in time, despite it is executed within an isolated environment. Our assumption is that being able to track the entire malware execution life-cycle enables a better understanding of current and emerging malware.

In addition we develop a concept for providing emulated services to a malware sample under analysis thereby fulfilling possible liability constraints. In particular we focus on the issue of handling unknown traffic patterns, such as C&C protocols, within our approach. To this end we present a proof of concept implementation leveraging finite state machines for generating service emulation scripts intended to spawn an emulated C&C service. We evaluate the feasibility of our proof of concept using C&C traffic from a self-made minimal botnet.

Zusammenfassung

Die verheerendsten Bedrohungen im Internet lassen sich heutzutage auf Schadsoftware (Malware), welche über Botnetze gesteuert wird, zurückführen. Zeitnahe Erkenntnisse über neue, durch Malware verursachte, Bedrohungen sind daher eine unerlässliche Voraussetzung für deren erfolgreiche Bekämpfung. Derartige Erkenntnisse werden üblicherweise durch die Sammlung und Auswertung aktueller Angriffsdaten sowie einer möglichst sorgfältigen Analyse der neuesten Malware Samples gewonnen. Die zunehmende Professionalisierung im Bereich der Internetkriminalität, und damit einhergehend die Weiterentwicklung der Malware, hat jedoch zum Einsatz umfassender Abwehrmechanismen geführt, welche die Analyse der Malware verhindern sollen. Aktuelle Malware durchläuft demnach während ihrer Ausführungszeit einen mehrstufigen und komplexen Zyklus. In dieser Arbeit wird ein neuartiger Ansatz für eine integrierte, Honeypot basierte, Malware-Sammlung und -Analyse vorgestellt, welcher einen Mehrwert gegenüber vorhandenen Ansätzen darstellt. Insbesondere werden die gängige Trennung von Sammlung und Analyse, Einschränkungen hinsichtlich Service Emulation sowie Risiken, die aus dem Betrieb von High-Interaction Honeypots resultieren, adressiert. Die übergeordnete Zielsetzung besteht in der großflächigen Sammlung und Analyse von Malware, wobei deren gesamter Lebenszyklus zur Ausführungszeit berücksichtigt werden soll. Die Sammlung und Analyse soll dabei möglichst automatisiert innerhalb einer kontrollierten Umgebung erfolgen. Dennoch soll es möglich sein, bis dato unbekannte Malware und deren entsprechende C&C Kommunikation zu analysieren. Im Gegensatz zu rein Netzwerk basierten Ansätzen setzt der vorgestellte Ansatz daher zusätzlich auf der Systemebene an und ermöglicht dadurch die der Malware zugrunde liegende Logik bereits zur Laufzeit zu erfassen. Damit kann ein Malware Sample während der Analyse mit allen aus dem Internet angeforderten Ressourcen versorgt werden, obwohl es sich tatsächlich in einer isolierten Umgebung befindet. Die zugrunde liegende Annahme ist dabei, dass die Analyse des gesamten Lebenszyklus der Malware ein besseres Verständnis hinsichtlich der Funktionsweise, und somit auch hinsichtlich der resultierenden Bedrohung, ermöglicht. Des Weiteren wird ein Konzept zur Bereitstellung von emulierten Services im Rahmen des vorgestellten Ansatzes erarbeitet. Indem nahezu jegliche Kommunikation der Malware mit der Außenwelt innerhalb der kontrollierten Umgebung nachgebildet werden kann, wird die Notwendigkeit, dass Malware während der Analyse mit Dritt-Systemen im Internet kommunizieren muss, erheblich gesenkt und so ein mögliches Haftungsrisiko minimiert. Im Besonderen wird dabei die Emulation von Services für unbekannten Netzverkehr (wie z.B. C&C Protokolle) betrachtet. Anhand einer prototypischen Implementierung wird evaluiert, ob basierend auf endlichen Zustandsautomaten entsprechende Skripte zur Emulation eines C&C Service erzeugt werden können. Dazu wird der Netzverkehr aus einem eigens zu diesem Zweck erstellten, minimalen Botnetz heran gezogen.

Ich versichere, dass ich diese Master's Thesis selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 9. Mai 2012 Martin Brunner

Acknowledgments

I am very grateful to my supervisor Prof. Herwig Unger for the opportunity to work on this thesis. I would also like to thank my advisors Daniel Berg and Sascha Todt for the guidance throughout the creation of this thesis. In addition I want to thank my colleagues from Fraunhofer AISEC in Munich, especially Sascha Todt, Christian Fuchs and Hans Hofinger, for their valuable feedback and the fruitful discussions.

Furthermore this thesis would not have been possible without my parents Gerlinde and Rudolf and their constant support and encouragement throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber crime has become one of the most disruptive threats today's Internet community is facing. The major volume of these contemporary Internet-based attacks is thereby attributed to botnet[1]-connected *malware* (short for malicious software) infecting hosts and instrumenting them for various malicious activities. Most prominent examples are Distributed Denial of Service (DDoS) attacks, identity theft, espionage and Spam delivery. Botnet-connected malware can therefore still be considered the major threat on today's Internet. Even worse, due to the ongoing spread of IP-enabled networks to other areas (such as the automotive domain, SCADA[2] or mobile devices) it can be expected that the threat posed by botnet-connected malware will

1. intensify (assuming attractive business models for the underground economy) and
2. reach further domains in public and private life, even taking "inconsiderable devices" (i.e., non-standard systems like workstations or servers) of the daily routine into account due to the rise of pervasive computing. This assumption is substantiated by the fact, that these inconsiderable devices (e.g., game-consoles, home-routers, network-printers) have already been reported to be easily abused by adversaries although the underground economy seems to lack a viable business model yet.

The threat gets furthermore aggravated due to recent technologies and trends (continuously cheaper processing power, cloud computing, social networks, etc.) as adversaries have already proven to adapt their attack vectors and business models in a very flexible way in the past. Thus there is a fundamental need to track the rapid evolution of these pervasive malware based threats. Especially timely intelligence on emerging, novel threats is essential for successful malware defense since attackers typically have a lead over defenders. This requires both, collection and examination, of current real-world attack data, such as malware samples and shellcodes[3]. Acquiring this data in sufficient quantity and variety is a major challenge and has been addressed by various research work in the past years.

[1] Botnets will be covered in more detail in 2.4

[2] SCADA is the abbreviation for Supervisory Control And Data Acquisition and refers to computer systems intended to monitor and control the infrastructure and processes in industrial environments, e.g., facilities

[3] Shellcode is a compact sequence of instructions (typically intended to spawn a shell) which is used within the exploitation process

1.1 Initial Situation

Honeypots (see 3.2) have been proven to be a fundamental part for malware collection as they are designed to lure attackers and malware thus providing valuable information on current attacks that would have been difficult to acquire otherwise. This enables further analysis and examination of the collected malware samples finally resulting in the ability to anticipate recent trends in the malicious landscape. Conventional types of honeypots (e.g., low-interaction) are mostly instrumented for the detection of undirected, widely spread attacks and are thus limited in the scope of attacks they can cover. The fact that an ongoing paradigm shift to client-side and targeted attacks (in terms of e.g., topological malware) has been witnessed in the past years led to the development of new types of honeypots, such as client- and high-interaction honeypots. Especially the latter can provide more valuable information than low-interaction honeypots while posing much more risk for uninvolved third party systems getting compromised at the same time which is often insufficiently addressed. Beside the evolution of the attack vectors also the actual malware itself evolved, amongst others regarding used obfuscation techniques: Current malware checks for several conditions before executing its malicious tasks, such as hardware resources of the victim host, Internet connectivity or whether it is executed within a virtualized environment. If these pre-conditions fail, it may behave different or refuse to execute at all. This implies, that the malware needs to get all requested resources during runtime in order to follow the whole execution life-cycle of the malware. While this could be achieved by allowing full interaction between the malware and Internet-connected third party systems, it is not applicable to most setups for several reasons. These issues in turn led to several enhancements in malware analysis techniques (e.g., instrumenting virtual machine introspection or sinkholing approaches) addressing the evolving sophistication of malware. However, existing publicly known approaches for malware collection and analysis suffer from several shortcomings.
First, collection and analysis is separated, loosing the context of the exploited system while analyzing the malware. Thus reactions to malware initiated attempts remain static during runtime (i.e., a specific sinkholing service can only be spawned once it is known that the malware will attempt to connect to it).
Second, the exposed risk to third parties, which results from the operation of high-interaction honeypots, is often inadequately addressed thus making their application hard to non-academic environments due to possible legal constraints and liability issues.
Third, if not being able to provide the malware with all requested resources during runtime, it will stop executing thus making it impossible to follow the whole life-cycle of the malware.
The research community lacks a viable solution for integrated malware collection and analysis which is able to handle these challenges.

1.2 Task and Contribution of this Thesis

In this thesis we will explore methods and techniques of existing measures for honeypot based malware collection and analysis and identify their shortcomings. The aim of the the-

sis is to develop a conceptional approach for advanced, integrated malware collection and analysis. For the service emulation component of the approach a proof of concept implementation has to be developed in order to evaluate the feasibility of leveraging finite state machines for generating service emulation scripts. To accomplish this task the procedure is divided into several steps:
First of all, the state of the art in honeypot based malware collection and analysis will be outlined in brief and necessary enhancements will be identified.
Based on these findings a holistic approach for integrated malware collection and analysis overcoming the identified shortcomings will be designed and described in a second step.
Next, a proof of concept implementation of the service emulation part as one component of the approach will be done in a further step. The focus is on handling unknown traffic patterns, such as C&C protocols. Therefore finite state machines will be used to generate service emulation scripts intended to spawn an emulated C&C service.
Finally, the feasibility of the proof of concept will be validated using using C&C traffic from a self-made minimal botnet.
The assumption is, that being able to follow the whole malware life-cycle enables a better understanding of currently ongoing or emerging attacks by botnet-connected malware. The findings are expected to make a significant contribution to enhanced analysis of current malware in particular and malware research in general.

1.3 Outline

The remainder of the thesis is organized as follows:

Relevant background information on cyber crime are provided within chapter 2 in brief. Therefore we define the assumed attacker model and provide some basics about malware and botnets. As a result we specify the execution life-cycle of modern malware, which has to be considered for a comprehensive analysis.

Chapters 3 and 4 present the state of the art in malware research and thus cover research work that has been done so far to the extend it is relevant for the thesis. This includes research on honeypots and the creation of a taxonomy of existing honeypot approaches and honeypot software as the primary means for malware collection. Also shortcomings of the existing honeypots are identified and based on these findings a honeypot for use in the holistic approach is proposed. Furthermore existing approaches in the field of malware analysis are briefly outlined as a decision basis for the presented approach.

Chapter 5 introduces the proposed holistic approach for integrated malware collection and analysis. It also refers to related work in this area and outlines the expected contributions and enhancements of the approach.

The implementation of the service emulation part for unknown traffic patterns as one component of the approach is covered in chapter 6. It presents also the findings and achieved results.

Chapter 7 summarizes the thesis, provides an outlook and proposes future work.

Chapter 2

Background

2.1 Introduction

This chapter is intended to serve as an introduction to cyber crime basics. First of all it presents the specified attacker model. It will therefore briefly describe common malware based attacks to the extend they are relevant for this thesis. After that the different attacker types including their specifics are outlined and finally some malware-specific aspects are described. The second part of this chapter summarizes basic information on malware in general, the various types as well as the evolution of malware and its life-cycle. It concludes with an introduction into botnets, their characteristics and mitigation.

2.2 Attacker Model

Attacker models are a well-established means in the area of computer security in order to classify attackers according to the (costly) impact they can cause. While there are some origins in the area of fault tolerant systems (e.g., the worst case within the *Byzantine failure model* [LSP82] is comparable to an adversary who has gained full control over a system) attacker models for computer security were first introduced by Dolev and Yao [DY83]. Generally malware aims at violating one or more protection goals of a system such as integrity, confidentiality or authenticity in order to perform unwanted actions. Specifically the aim of a given malware depends on its use case and thus of the intention of the attacker instrumenting the malware. Since a given malware acts in place of the attacker the term refers to both, a human attacker and malware, in the following sections. According to [Eck09] an attacker model describes attacker types taken into account as well as their resources, motives, aims and skills. In order to specify an attacker model for use within this thesis we will therefore briefly define the types of expected attacks as well as assumptions regarding the adversary in the following sections.

2.2.1 Attack Classes

According to [ISO] an attack is defined as
"any attempt to destroy, expose, alter, disable, steal or gain unauthorized access to or make unau-

thorized use of an asset.“

and as

”an assault on system security that derives from an intelligent threat, i.e., an intelligent act that is a deliberate attempt (especially in the sense of a method or technique) to evade security services and violate the security policy of a system.”

respectively according to RFC 2828 [Shi00]. This implies that there exists a variety of basic attacks against a broad range of diverse targets. While various of these attacks - such as denial of service (DoS), brute force, injection, attacks exploiting buffer overflow and format string vulnerabilities or network exploitation (sniffing, scanning, spoofing, flooding, etc.) - are also instrumented by malware we will focus on the following attack classes (i.e., malicious activities) which have been witnessed to be most commonly used by nowadays malware and thus are considered as relevant in this context.

Distributed Denial of Service (DDoS)

A denial of service (DoS) attack aims at preventing legitimate users from accessing a resource and can occur either locally or remotely (as well as accidentally or intentionally) depending on the targeted resource. A distributed denial of service (DDoS) attack is performed remotely and can be characterized as a large-scale, coordinated attempt to attack a network resource (i.e., a server or service) by exhausting its bandwidth thus harming its availability. A DDoS attack is thereby launched indirectly, since it instruments a large number of compromised hosts each of them launching a DoS attack against the actual target making the attack much more disruptive than a conventional DoS. This enables the attacker to remain hidden (i.e., much harder to track him down) and hinders countermeasures from a victims perspective. A remote (D)DoS attack can use several techniques, such as SYN-flooding[1]. Further information as well as a taxonomy and countermeasures are described in [SL04] and [KL01] in more detail.

Spam Delivery

Spam is the (ab)use of electronic messaging systems in order to distribute unsolicited messages. While it also occurs in various other media (e.g., instant messaging) we refer hereby to Spam delivered via Email, mostly known as unsolicited bulk Email (UBE) or unsolicited commercial email (UCE). Although Spam itself merely represents an illegitimate usage rather than an actual attack it is relevant in this context for two reasons: (i) Malware distribution is (still) done via Spam messages on several ways (e.g., as an attachment, via malicious content or malicious links pointing to a manipulated website which infects the victim host with malware) and (ii) nowadays the actual delivery of UBE is mainly performed using malware on compromised hosts. Furthermore the large amount of Spam Emails seriously affects the operation of both service providers and their users.

[1] http://en.wikipedia.org/wiki/SYN_flood

Identity Theft and Espionage

Identity theft refers to illicitly obtaining confidential information through a computer network for malicious purposes and has become one major application of todays malware. Transmission channels for sending back the collected information are widely used protocols such as HTTP, FTP, Email or IRC. Additionally the use of public key cryptography has been witnessed and can be considered state of the art. But also further stealth measures such as steganography techniques are possible. There are many kinds of confidential information, for a private person an (incomplete) list would include: Credit card numbers, bank account numbers, social security numbers and user credentials for Email, social networks or online-shops. Likewise there is a number of possible applications: Once installed malware is often used to impersonate the victim for fraudulent purposes (e.g., phishing), data theft (trade secrets, intellectual property, etc.), installation of ransomware[2] or resale on another market. Once an identity has been completely taken over the adversary may even commit crimes in the name of the victim resulting in more than financial issues. Furthermore the gathered personal information can be used for sophistication of additional attacks. A comprehensive insight, including several examples for identity theft using malware can be found in [Gut07].

Sabotage

Sabotaging a system may aim at secondary objectives, i.e., obfuscation mechanisms in order to ensure the malware remains undiscovered. But it can also be the primary objective, i.e., manipulating the compromised system itself or a system that is controlled by the compromised host (e.g., a SCADA system). Especially the latter can result in devastating attacks when using highly sophisticated malware, as recently demonstrated by Stuxnet [FMC11, BHK+10].

Social Engineering

Contrary to instrumenting technical means to break into a computer system social engineering is about manipulating people into revealing confidential information or performing actions using psychological manipulation, therefore also referred to as *"art of deception"*. Thus these class of attacks targets the human as the weakest link in chain. The attacker instruments social aspects such as establishing of trust, which can be facilitated by knowledge of internal information (e.g., contacts). While there are various social engineering techniques, such as *phreaking*[3] or *dumpster diving*[4], social engineering is nowadays widely used for distributing malware, e.g., by tricking users into clicking on a link in order to install the malware. Most prominent example for a social engineering attack is phishing, although there is a number of other attack vectors, such as exploiting popular events or social networks. A comprehensive outline of social engineering techniques including a variety of examples can be found in [Mit02]. Furthermore recent studies, such as [BSSW11],

[2] Ransomware encrypts data on the compromised system and extorts money from the victim to restore it
[3] http://en.wikipedia.org/wiki/Phreaking
[4] http://en.wikipedia.org/wiki/Dumpster_diving

expect malware combining social engineering with large-scale automated identity theft to remain a major issue in future.

2.2.2 Attacker Types

Basically we divide attackers into two types:

- *External attackers (Outsiders)* do not have access to internal information or resources of the target, such as network access or user credentials. Thus they operate from outside the envisioned target using technical and/or social engineering means.
- *Internal attackers (Insiders)* do have access to internal information and resources (e.g., an employee would have access to the corporate network). An insider may have access to or obtain internal information which is difficult, if not impossible, to acquire for an external attacker.

In addition, according to [Eck09], the attacker model is influenced by further aspects which represent the potential impact an attacker can cause. We will consider them as follows.

Motives

The attacker motives refer to the question why an adversary attacks a given target. While there are various possible motivations for an attack (due to human nature) we group them into three basic classes:

- *Specific victim*: The attack is directed to a specific target (i.e., an individual system or person) for reasons of sleuthing, revenge or sabotage.
- *Victim group*: A given target is chosen due to its image, e.g., a successful compromise reflects political reasons, represents a challenge or promises glory.
- *Random victim*: The adversary does not have any specific relation to the target and thus performs an opportunistic attack (e.g., gaining of financial profit or vandalism).

Goals

The goals describe the intention, i.e., what does the attacker ultimately want to achieve. Exemplary goals cover spying, disruption, finding vulnerabilities, deception or preparation of further attacks but also non-malicious intentions like demonstrating vulnerabilities (e.g., within a penetration test).

Skills and Resources

Depending on the sophistication of the attacker (i.e., what is he able to achieve using his technical skills) and the available resources (e.g., time, monetary, processing power) we make the following classification.

- *End-users* do not have any expertise on attacking computer systems, but may have access to critical resources and knowledge about trade secrets enabling them to cause serious damage, either intentionally (e.g., a frustrated employee,) or unintentionally (e.g., an end-user tricked into installing malware). They will typically appear as insiders and mostly have little resources.
- *Script kiddies* have little knowledge about IT security and utilize existing scripts or easy-to-use exploitation frameworks such as Metasploit[5] without understanding the underlying concepts and techniques. Since such tools are publicly accessible for free this type of attacker is quite common. Their typical motives are curiosity, vandalism or glory rather than financial profit. Both, their motives and their skills, suggest that script kiddies have typically little resources.
- A *hacker* is a highly-skilled attacker intending to discover weaknesses and vulnerabilities in computer systems in order to develop exploits demonstrating the feasibility of his findings. The motivation of a hacker is mostly not personal enrichment, but to enhance the overall security of computer systems. He does so by disclosing his exploits to the vendor, a community or the public in order to point to a discovered security issue often enforcing the release of a patch. A hacker is up to invest a lot of time but does not necessarily have much financial resources.
- A *cracker* is another type of highly-skilled attackers whereas his goal is, contrary to a hacker, to systematically instrument his skills for personal enrichment or causing directed harm to others. Furthermore this type of attacker has the most resources regarding both, monetary and time.

2.2.3 Malware-specific Aspects

First we differentiate between two aspects:

1. how malware infects a system and
2. how malware affects a system.

The first aspect deals with the attack vectors, i.e., how malicious code gets into a system, and will be covered by 2.3.4. The second aspect refers to what the malware is used for, i.e., its purpose. Hereby we distinguish between two basic types of malicious activities depending whether the victim host is

- passively abused: this refers to issues related to information disclosure (such as data theft and espionage) and identity theft (e.g., phishing and credit card fraud) or
- actively abused: this includes Spam delivery, illegal content hosting, sabotage and attacking further systems (e.g., DDoS).

Furthermore, on a general layer (i.e., independent of the classification above), we differentiate between undirected and targeted attacks. depending on the motives of the attacker.

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

Undirected attacks are not dedicated to a specific target but aim at compromising random nodes. They mainly depend on the attack method rather than on the victim host and its specific properties. This includes the compromise of preferably many systems for establishment of a botnet (refer to 2.4), vandalism (e.g., website defacement) or directly instrumenting the takeover for monetary, criminal reasons such as (large-scale) identity- or data-theft. Therefore undirected attacks are represented by autonomous spreading malware or activities driven by script kiddies utilizing attack tools.

Targeted attacks on the other hand primarily take one or more specific properties of the intended target(s) into account in order to perform nefarious purposes and cover (again, depending on the motives):

- hijacking of "profitable" targets, such as government systems for political reasons, well-known large enterprises for image- or monetary-reasons (also representing a challenge) as well as
- technically specific (groups of) targets, such as DNS servers (enabling further attacks), hosts with specific (versions of) operating system, third party applications and also the general device type (network device, printer, smart-phone, ECU[6], smart meter, PLC[7]).
- Another aspect relates to targeting a specific person, organization or nation for several reasons as recently seen in the media[8] [9] [10] whereas there is some obvious overlap with the points described above.

These type of directed, often sophisticated attacks thereby utilize specialized means (e.g., topological malware) and are issued by crackers for professional reasons, such as cyber crime or intelligence.

2.3 Malware

2.3.1 Definition

Generally malware is an umbrella term that refers to

"any software designed to cause damage to a single computer, server, or computer network"[11].

As there are various ways of causing damage this implies, that there exists a broad range of different types of malware. This is indicated by another, more specific, definition by [Nas05] specifying malware as

[6] http://en.wikipedia.org/wiki/Electronic_control_unit
[7] http://en.wikipedia.org/wiki/Programmable_logic_controller
[8] http://www.wired.com/gamelife/2011/04/psn-down/
[9] http://pandalabs.pandasecurity.com/operationpayback-broadens-to-operation-avenge-assange/
[10] http://en.wikipedia.org/wiki/2007_cyberattacks_on_Estonia
[11] http://technet.microsoft.com/en-us/library/dd632948.aspx Retrieved 2011-06-10

"Programming (code, scripts, active content, and other software) designed to disrupt or deny operation, gather information that leads to loss of privacy or exploitation, gain unauthorized access to system resources, and other abusive behaviour. Examples include various forms of adware, dialers, hijackware, slag code (logic bombs), spyware, Trojan horses, viruses, web bugs, and worms."

2.3.2 Types and Terminology

A variety of different types (and in turn subtypes) of malware including overlap and hybrid forms have shown up in the past years. Thus there are various approaches to classify malware according to given characteristics, e.g., the way it interacts with the operating system (OS) as proposed by Rutkowska [Rut06]. Furthermore there is no common unique naming scheme (yet), although there has been made some effort in the past by the Anti-Virus (AV) industry such as the CARO malware naming scheme[12] and the Common Malware Enumeration (CME)[13]. In this section we will therefore briefly describe the main basic types as frequently seen in the Internet.

Viruses

Cohen gives in [Coh87] the following informal definition of a computer virus:
"A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself."
A virus is therefore a non-autonomous sequence of instructions (thus relying on a host program to execute) able to reproduce itself during execution. It does so by either writing a copy or a modified version of the virus into a memory area that does not contain the instruction sequence so far (infection). This means that a virus becomes active by the execution of a certain program. Additionally to reproducing capabilities viruses typically carry a damage routine that may be triggered due to certain conditions [Eck09] p.51ff. This malicious routine often provides characteristics of other malware such as placing a backdoor or destructive tasks. Depending on the method used to infect a system viruses are divided into file viruses, boot sector viruses, macro viruses and script viruses[14].

Worms

A worm is a piece of malicious software that runs autonomously and has the ability to to reproduce itself via computer networks. In contrast to viruses, worms are typically standalone applications without the need of a host program thereby not spreading locally but via services provided by computer networks. Each subsequent copy of a worm is in turn able to self-replicate. [Szo05]

Worms use various techniques to remotely penetrate machines in order to launch copies of themselves[15], including

- social engineering (e.g., tricking the user into opening a file attached to an Email),

[12] http://www.caro.org/naming/scheme.html Retrieved 2011-06-11
[13] http://cme.mitre.org/ Retrieved 2011-06-11
[14] http://www.securelist.com/en/threats/detect/viruses-and-worms Retrieved 2011-06-11
[15] refer to footnote 14

- utilizing configuration errors (copying to a fully accessible network-share) and
- exploiting vulnerabilities in the operating systems network services or network applications.

Thereby they often instrument more than one propagation vector to spread copies via computer networks such as Email attachments, Peer-to-Peer (P2P) file sharing networks or malicious links pointing to a web or FTP resource (which in turn may be included in websites, Emails or within instant messaging services). Some worms propagate directly as network packets targeting the computer memory of the vulnerable machine, where the code is activated once placed. Since in this case the propagation itself is done without any user interaction and thus typically without the user being aware worms may spread with very high speed. While not necessarily containing functions to harm the infected machine worms often carry a malicious payload (e.g., a trojan) in addition to the propagation mechanisms.

Trojans

A Trojan (short for Trojan horse) is a program whose expected behaviour does not correspond to its actual behaviour. That is, while implementing the expected features it also implements additional, hidden (and malicious) functionality. Its main purpose is to perform harmful actions on a compromised machine which are not authorized by the user who has thereby no influence on this hidden functionality. Unlike viruses and worms, Trojans do not provide self-replication mechanisms and thus must have a carrier. They can either be installed manually by the attacker, by the unsuspecting user tricked into installing it or via the malicious payload of a worm. In order to attract the user a Trojan disguises as a benign software offering something desirable to the user. Common functionality includes unauthorized deletion, copy or modification of data for fraudulent purposes such as stealing credit card numbers and passwords or obtaining sensitive information (i.e., espionage). To accomplish these task Trojans instrument various malicious techniques such as
Keyloggers: A keylogger records all keystrokes pressed on the compromised machine and may be implemented in software and hardware. Its typical purpose is to monitor all activities issued by the user on the victim host in order to gather sensitive information such as passwords. The recording process is usually performed as a covert operation leaving the user unaware that his actions are being monitored.
Backdoors: A backdoor is a functionality offering unauthorized remote control over an infected machine to the attacker by offering hidden access bypassing the security measures of the system. This enables the attacker to access the compromised host anytime and unnoticed in order to perform any action (e.g., displaying the screen, deleting data, transfer files, reboot, etc.) keeping full control over it. Backdoors are often used to merge a group of compromised hosts to a botnet and instrument them to perform various malicious activities such as Spam delivery or DDoS.

Rootkits

Once a system has been successfully compromised an attacker likely wants to cover all tracks that might result in detecting the intrusion such as suspicious log-files. Furthermore

he may want to install a backdoor for keeping unnoticed access to the compromised host enabled and install additional malicious tools in order to to utilize the host for malicious purposes. Therefore several modifications to the system have to be made for the malicious processes to remain unnoticed (e.g., hide their presence to the process tree). These activities require a number of non-trivial, manual tasks in order to hide the abusive activities from the user, the system and possibly installed Anti-Virus (AV) software. Therefore they are bundled in a software integrating and automating these tasks, a so-called rootkit. The term derives from the superuser in UNIX-based systems which is called root. The level of sophistication of rootkits emerged continuously over the past decade resulting in a variety of rootkits for all common operating systems. Depending on the targeted privilege ring of the corresponding hardware there exist rootkits operating in user-mode, in kernel-mode, on the hypervisor- or even the firmware-level. An overview of UNIX rootkits can be found in [Chu03].

Bots

The term bot is derived from (software) robot and refers to an autonomously (i.e., without human interaction) acting computer program. That is, it executes the same tasks continuously, e.g., by autonomously reacting to external events. While bots are not necessarily malicious programs (bots were originally designed for benign purposes such as web-crawler for search engines or chat-bots) we refer hereby to the term bot in its destructive characteristic. Within the IT security domain a compromised host is also referred to as bot, *zombie* or *drone*. A malicious bot is intended to nest in a system after successful compromise and to sustain a remote control mechanism for the attacker in order to command the compromised system. A group of bots (i.e., infected machines) can be united to a *botnet* (refer to 2.4) by the attacker in order to further enhance the effectiveness of his attacks. This connecting back to a server or another infected machine is the main characteristic of a bot. Moreover, depending on the sophistication, bots contain various routines to perform malicious actions. Thus a modern bot represents a concept of advanced malware which incorporates several techniques that were introduced by the aforementioned malware types and enables an attacker to overcome their disadvantages, e.g., when utilizing a worm an attacker must define all actions in advance. Bots add another layer of flexibility compared to conventional malware types since they enable the attacker to react more flexible to specific situations (e.g., the attacker can send Spam today and launch a DDoS attack tomorrow). Therefore bots have emerged to the major malware-class in the past years showing up in various variants of the four major bot codebases: Agobot/Phatbot/-Forbot/XtremBot, SDBot/RBot/UrBot/UrXBot, Spybot and mIRC-based Bots (GT-Bots) [BY07]. As a result of the continuous evolution of bots they have widely gained attention in the research community and led to numerous research in this area (further details on bots are provided by e.g., [Hol05] and [BHKW05]).

Other types: Crimeware

Apart from the basic types described above - who do not necessarily have a commercial background (e.g., destructive viruses) - a whole subclass of malware has emerged primarily targeting at gaining (financial) profit for the attacker. These type of malware, whose us-

age is dedicated to (automated) fraudulent purposes, is also called *crimeware*[16] and often instruments Trojan capabilities. One of todays most prominent examples for crimeware is the ZeuS Trojan[17] [18] which specifically targets users' banking credentials. In addition there is a class of malicious applications behaving in an annoying or undesirable way but posing less serious risk to a system than malware. This so-called *grayware* is obviously a subclass of crimeware and refers to programs performing various unwanted actions such as tracking user-behavior or raising pop-up windows thus negatively affecting the system. An exemplary listing of crimeware would encompass:

- *Spyware*[19] collects information about the computer system and its user (e.g., Internet activity) without an adequate consent of the user and forwards this information to third parties. While the impact caused by spyware is lower than the possible impact of a Trojan, spyware can still be considered to pose risk to the system, since the forwarding of personal information may expose the system to illicit modifications.

- An application which is designed to display advertisements, to redirect search requests to advertising websites or to collect marketing-type data about the user (e.g. websites visits) in order to display customized advertising on the computer is referred to as *adware*[20]. Contrary to a Trojan it collects data with the user's consent.

- *Ransomware* refers to crimeware that renders a compromised computer's data unusable and offers to restore the data for a fee. Typically ransomware achieves this by encrypting data on a hard drive, and decrypting it after payment.

- *Scareware* covers applications with typically limited or no useful functionality rather than malicious content. In order to be sold scareware instruments social engineering techniques to trick users into believing that they have a demand for the application. One frequently used method is to make the user believe that he is infected with malware suggesting to buy the advertised software which pretends to be the only solution to get rid of the malware. This software is thereby useless and the supposed malware is totally fictional[21].

2.3.3 Evolution

While the idea of self-replicating code dates back to John von Neumanns theory of Self-Reproducing Automata [Neu66] virus-like programs showed up in the 1980s [Szo05], where also Fred Cohen presented the concept of a virus for the first time [Coh87]. However, these first publicly known types of malware have little in common with todays variants, since they mainly dealt with the phenomena of self-replicating code itself and were intended merely as a proof of concept (e.g., by presenting a poem on the desktop after infection) rather than actually performing malicious tasks [22]. Although designed to perform useful

[16] http://usa.kaspersky.com/resources/crimeware Retrieved 2011-07-05
[17] http://www.secureworks.com/research/threats/zeus/?threat=zeus Retrieved 2011-07-05
[18] , https://zeustracker.abuse.ch/statistic.php Retrieved 2011-07-05
[19] http://www.microsoft.com/security/portal/Threat/Encyclopedia/Glossary.aspx#s Retrieved 2011-07-05
[20] http://www.securelist.com/en/threats/detect/adware Retrieved 2011-07-05
[21] http://news.bbc.co.uk/2/hi/technology/7955358.stm Retrieved 2011-07-05
[22] http://vx.netlux.org/timeline.php Retrieved 2011-08-05

tasks in a distributed environment also harmful effects showed up, e.g., within the launch of the Morris Worm [Spa88] which was originally designed to estimate the size of the Internet. Accordingly the first virus outbreaks affected those systems which were widely used at that time (e.g., Apple II, MS-DOS, VAX/VMS). With the rise of the Internet in the 1990s also the early days of widespread malware begun. While that era was dominated by Trojan-like remote control programs such as SubSeven[23], NetBus[24] and Back Orifice[25] and the primary motivation for malicious activities seems to have been vandalism and hacktivism (i.e., hacking for political reasons) performed by script kiddies the motivation changed from beginning of 2000 when the professionals moved in. Within the resulting paradigm shift towards an economic motivation for malicious activities[26] backed by organized (cyber-)crime also the sophistication of malware continuously evolved hence increasingly impeding the defense against the malware threat. Thereby the main goal of the cyber criminals is to inject and run their malicious code on the victim host in order to instrument it for malicious purposes. To achieve this there are basically two methods

1. attacking the system by exploiting a (remote) vulnerability in the software in order to inject shellcode which then downloads the actual malware binary and runs it.

2. tricking the user into downloading, installing and running the malware, e.g., instrumenting social engineering means.

After all, for a professional attacker type (refer to 2.2) the invested effort and the achieved result must be in an economic relation. Hence we experience the phenomena of a moving target. That is, cyber criminals will chose the targets according to the best economic relation. In the first years after 2000 the first method was predominant. The thereby emerging professional underground economy focused on mass attacks, aiming at the infection of preferably many hosts utilizing remote vulnerabilities in the operating system. The main target of these attacks was Microsoft's Windows XP (before the release of Service Pack 2) due to its widely deployment at that time. Also various flaws in the RPC services existed, there was no firewall installed by default and most users connected to the Internet via a dial-up link rather than via a router with NAT or even firewalling capabilities. Thus most end-user systems were exposed totally unprotected to autonomous spreading malware easing large-scale infections. Due to the common usage of routers for broadband up-links and the increasing resilience of nowadays operating systems against remote flaws a paradigm shift has been witnessed in recent years to the second method, which is manifested e.g., in [BSI11]. As a result nowadays primary attacks vectors for malware distribution are

1. attacking (third-party) client-software that interacts with the Internet such as Browsers, Email clients and corresponding extensions like Adobe's Flash and PDF viewers and

2. attacking the human as weakest link in the chain using social engineering techniques.

A common scenario includes both, i.e., a user receives a link to a malicious website (e.g., via Email or social networks) and gets infected via a so-called *drive-by download*. That is,

[23] http://www.tcp-ip-info.de/trojaner_und_viren/subseven.htm Retrieved 2011-08-18
[24] http://www.tcp-ip-info.de/trojaner_und_viren/netbus_pro_eng.htm Retrieved 2011-08-18
[25] http://www.cultdeadcow.com/tools/bo.html Retrieved 2011-08-18
[26] http://www.securelist.com/en/threats/detect?chapter=72 Retrieved 2011-07-05

once visiting the website, vulnerabilities in the browser, its plugins or the OS itself are exploited in an automated fashion without any user interaction resulting in a malware infection and thus shifting the attack to the application level. Thereby the user remains unable to recognize whether a given website has been manipulated. As a result drive-by-downloads can currently be considered as one of the most serious attack vectors. Tricking the user into clicking on a link or visiting a certain website is hereby done using various social engineering techniques which can also be automated. Even human conversations an be automatically misused for social engineering as recently shown by Kirda et al. [KPL10]. Furthermore several work indicates an ongoing specialization of the various groups in the underground economy offering "Malware as a Service" and "pay per install" schemes including elaborated models for pricing, licensing, hosting and rental [Gut07, PW10, FPPS07, HEF08, CGKP11]. This includes specifically professional maintenance, full support and service level agreements (i.e., guaranteeing to be undetectable) of the purchasable malware itself as well as many innovations in the maintenance of malware-infected victim hosts (i.e., installation, updating and controlling). Therefore there is (i) one group specializing on the development of the actual crimeware, (ii) a second group deals with the operation platform and the distribution of malware (for establishing botnets) and (iii) a third group focuses on suitable business models. In addition several recently witnessed trends include

- the usage of scareware instead of viruses (since viruses are not worth money),[27]
- the drastic decrease period of using a given malware (even to several hours)[28] and
- the ongoing sophistication of targeted attacks against companies, governmental institutions and even small political organizations. Large-scale infections as known from Sasser[29] are therefore not expected anymore, since malware-based attacks are continuously individualized, i.e., they target a very limited number of victims, are often sophisticated and the utilized malware is replaced frequently [BSI11].

This evolution led to a sophisticated, increasingly complex and multi-staged life-cycle of todays malware which has to be considered and is described in the following section.

2.3.4 Life-Cycle

With respect to the facts outlined in the previous section (2.3.3) and findings of related work [CGKP11, Oll11] we model the execution life-cycle of today's (autonomous spreading) malware as depicted in figure 2.1. Particular stages within this life-cycle may differ depending on the malware type (refer to 2.3) and are accordingly addressed in separate work (e.g., the life-cycle of Web-based malware has been analyzed in [PMP08]). In any way widely spread malware is most effectively managed within a botnet, therefore a newly compromised host is likely to become a botnet-member. A common setting consists of three phases:

[27] http://h-online.com/-1318714 Retrieved 2011-08-05
[28] http://heise.de/-206078 Retrieved: 2011-07-02
[29] http://www.symantec.com/security_response/writeup.jsp?docid=2004-050116-1831-99 Retrieved 2011-08-18

1. **Propagation and exploitation**
   Within this initial phase a worm spreads carrying a malicious payload that exploits one or multiple vulnerabilities. In this context a vulnerability encompasses also the human using social engineering techniques (refer to 2.3.3) thus possibly requiring user interaction. Furthermore a vulnerability may be OS based (e.g., a flaw in a network service) or - more commonly - application based. The latter includes specifically vulnerabilities in browsers, their extensions (such as Adobe's flash), Email clients (e.g., attachment, malicious links, Emails containing malicious script code, etc.) and other online applications such as instant messaging clients. Thereby the malicious payload of the worm may instrument a variety of attack vectors reaching from (classical) buffer and heap overflows [One96, Eri06] to recent return oriented programming techniques [Sha07] while evading appropriate countermeasures such as address space layout randomization (ASLR), data execution prevention (DEP) and sandboxing [Kle04]. After successfully exploiting a vulnerability a shellcode is placed on the victim host which gets then extracted and executed, including decryption and deobfuscation routines when necessary.

2. **Infection and installation**
   As a result of executing the injected shellcode a binary is downloaded and placed on the victim host. This binary is typically a so-called *dropper*, which contains multiple malware components and is intended to disable the security measures on the victim host, to hide the malware components and to obfuscate its activities before launching the actual malware. It is synonymously referred to as *downloader*, which has the same features except that it does not contain the actual malware but downloads it from a remote repository resulting in a smaller size [Oll11]. As there is an emerging trend that multiple cyber criminals instrument a single victim host for their malicious purposes also several droppers may be installed (in parallel) within this step. Once the dropper is executed it extracts and installs further components responsible for hardening and updating tasks. That is, they prepare the system for the actual malware using embedded instructions. These tasks include e.g., disabling security measures, modifying configurations and contacting a remote site for updates ensuring that the actual malware is executed after every reboot and impeding its detection and removal. After the update site has verified the victim host as "real" and probably worth getting compromised it provides the dropper components with information on how to retrieve the actual malware (e.g., via an URL) and updated configurations, when necessary. Again, this may include multiple binaries each representing a different botnet. Once downloaded, the malware is executed by the dropper component installing its core components. Finally these core components remove all other (non-vital) components resulting from previous stages and the malware is operational. This installation life-cycle is described more detailed in [Oll11].

3. **Operation and maintenance**
   First of all the malware launches several actions which are intended to directly gain profit from the victim in case the attacker loses control over the compromised host later on. Therefore the malware harvests valuable information such as credit card numbers and all kinds of authentication credentials and sends it as an encrypted

file to a remote server under the control of the attacker. Next, the the malware attempts to establish a communication channel to the attackers command and control infrastructure (C&C, refer to 2.4) awaiting further instructions. These may include commands to launch different malicious actions but also maintenance operations such as retrieving updates, further propagation or even to terminate and remove the malware.

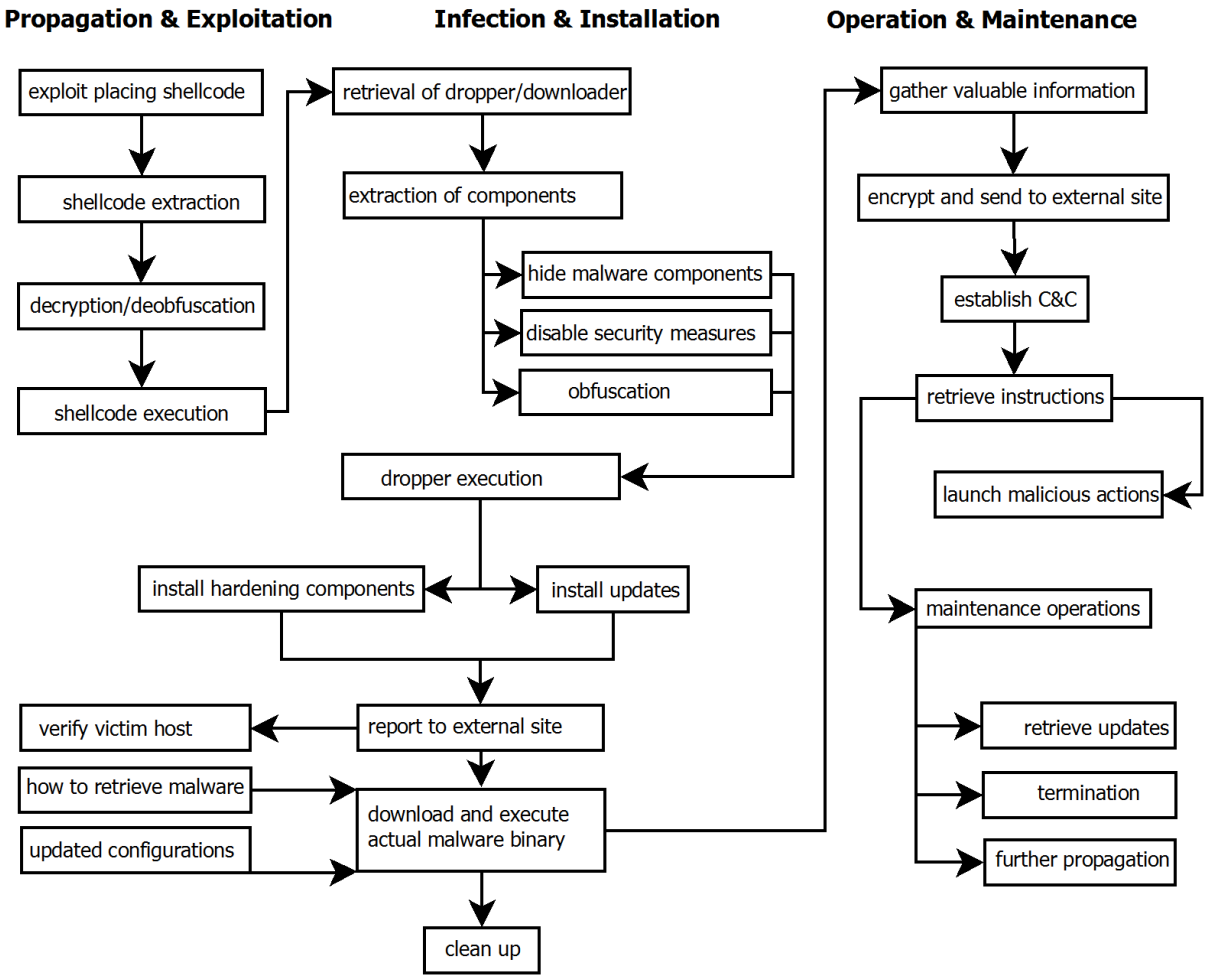

Figure 2.1: Today's Multi-Staged, Complex Malware Life-Cycle

The various steps of the outlined, complex malware life-cycle include many checks and measures to increase the resilience of the various features each intended to maximize the success of the malware installation, ensuring a reliable operation and to protect the cyber criminals from being tracked down. Thus the reasons for this complex life-cycle, especially the use of droppers in an intermediate step, are apparent: First the dropper components evade discovery of system compromise. Also the adversary has no need to distribute the core malware components in the first phase thereby impeding the successful collection and thus detection and mitigation of the malware. In addition he can distribute the malware

more selective and targeted. Finally he can ensure, that the victim host is real (i.e., not a honeypot or virtualized analysis system) and that it is worth being compromised (e.g., by checking available resources). Amongst others, this implicates, that for a comprehensive analysis of a given malware it must receive all resources that it requests during its life-cycle, since it may behave different or refuse to execute at all otherwise.

2.4 Botnets

As outlined in section 2.3 bots represent a powerful and efficient means in remotely controlling compromised hosts (zombies) for illicit purposes. For an attacker bots become even more powerful once he manages to control a large number of these zombies in parallel. Therefore a malicious remote control network, commonly referred to as *Command-and-Control (C&C)* network, is established uniting a (possibly very large) number of zombies to a so-called *botnet* [BHKW05, McC03] which is then under unified control of a single attacker, the so-called *botmaster* (also referred to as botherder). A single botnet, such as Torpig [SGCC+09], can thereby consist of hundreds of thousands of grouped bots. The botmaster uses the established C&C network to push out commands and updates. Compared to other propagation means, such as worms, botnets allow a greater control and management over compromised hosts, are less emergent and provide optimal flexibility. A detailed insight into botnets is provided by Barford and Yegneswaran in [BY07]. In order to optimize operation various means have been instrumented within botnets providing encrypted and stealthy communication, maximum resilience and robustness [Pol10], amongst others resulting in different, continuously enhanced architectures:

2.4.1 Centralized C&C Server

Conventional botnets are centralized i.e., the bots connect to a central instance, which is typically an IRC server. The botmaster pushes instructions via this IRC server using a special IRC channel, that is also known (i.e., hardcoded) to the bots [CJM05]. While this architecture provides low latency there are two main disadvantages from an attacker's perspective: (i) As many clients connect to this central server the botnet is easily detectable and (ii) once the central instance has been compromised the whole botnet can be taken down. Therefore it may be sufficient to observe the communication of only one participating client.

2.4.2 P2P based C&C Server

Contrary to centralized C&C architectures, within a P2P based botnet each bot represents a node in a decentralized network. Thereby a bot serves as both, client and server, that is it consumes network-offered services from other nodes while offering services on its own. Therefore this architecture provides much more resilience and robustness and is as a result much harder to take down. Rather than IRC P2P based botnets utilize (modified) HTTP or proprietary protocols for C&C communication. A comprehensive case study on P2P based botnets has been presented by [GSN+07].

2.4.3 Fast-Flux Service Networks

Another sophisticated technique instrumented within botnets are so-called Fast-Flux Service Networks (FFSN). Thereby an intermediate layer is inserted between the zombie hosts and the C&C servers using public, rapidly changing (within minutes) DNS entries. Therefore a combination of the round robin scheme and a preferably low TTL per DNS record (RR) is used. The goal is to assign a preferably high number of IP addresses (up to several thousand) to a given domain. If a zombie attempts to establish a connection to a C&C server within a FFSN it contacts a given domain. As a result the zombie receives several IP addresses, whereas it choses one using round robin. The IP addresses in turn are not those of the real C&C servers but belong again to compromised hosts acting as proxies forwarding the communication to the C&C server(s). In addition load balancing schemes are utilized in order to ensure maximum availability and bandwidth over all layers of compromised hosts.

A taxonomy of botnet structures has been proposed by Dagon et al. [DGLL07]. In recent years botnets have gained widespread attention to the research community and thus emerged to a major research topic resulting in numerous work addressing the detection, tracking and mitigation of botnets, such as [Hol09, FHW05, CLLK07, GZL08, HSD+08, DZL06, RSL+10, ZDS+08, PGPL11].

2.5 Summary

In this chapter we introduced the relevant background on cyber crime and malware basics as a common basis for the following chapters. Therefore we first specified an attacker model for use within this thesis consisting of the attack classes (as relevant for malware based attacks), the corresponding attacker types (including their characteristics) and outlined malware specific aspects. We differentiate between DDoS, Spam, identity theft, sabotage and social engineering and describe each attack class. We classify the various attacker types also taking their motives, goals, skills and resources into account. In our resulting classification we differentiate between end-users, script kiddies, hackers and crackers. Malware specific aspects consider how malware infects a system and how malware affects a system, differentiating between passive and active abuse and, with respect to attacker motives, undirected and targeted attacks. Second we summarized some basics about malware. After generally defining the term malware we specified the used terminology and described the main basic malware types. We separately deal with crimeware, a whole subclass of malware that has emerged primarily targeting at gaining (financial) profit. After that follows a brief history of malware outlining the evolution from the first idea of self-replicating code to nowadays sophisticated malware types, including paradigm shifts in attack vectors, propagation and usage due to an ongoing specialization of the cyber crime scene. Based on that we described the sophisticated, increasingly complex and multi-staged life-cycle of todays malware as a result of this evolution. This life-cycle considers the propagation and exploitation, the infection and installation as well as operational and maintenance aspects, since it is important for the approach proposed later on in this thesis. We concluded with a brief section introducing botnets and their characteristics including the main architectures in order to outline the threat posed by current botnets.

Chapter 3

Malware Collection using Honeypots

3.1 Introduction

This chapter presents honeypots as a valuable means for efficient malware collection. Therefore it outlines the principle of honeypots which differs from conventional approaches for attack detection. Next the various techniques used by honeypot implementations to realize this principle are described. A classification of existing approaches is then made, briefly describing the known honeypot implementations and possible shortcomings to the extend they are relevant within the thesis. Based on the findings follows the creation of a taxonomy of existing honeypot approaches and honeypot software. This is done as a decision basis for selection of one implementation that will be used for the proposed approach. The chapter concludes with a brief section on honeynets which are networks of honeypots bundling their resources.

3.2 Honeypot Concept

While conventional, defensive means for attack detection, such as intrusion detection systems (IDS), are based on knowledge about specific attack patterns and thus are designed to detect *known* attacks (e.g., by detecting signatures or anomalies), honeypots represent a different, complementary approach. Bill Cheswick was one of the first presenting the honeypot approach in the paper "An Evening with Berferd"[Che92]. Generally spoken honeypots are electronic bait, i.e., strictly monitored resources intended to get compromised in order to gather information about the methods and tools that have been used to exploit known and especially unknown vulnerabilities. Specifically honeypots are utilized to research and explore strategies and practices of malware authors and attackers. Lance Spitzner gives one of the few precise definitions by describing a honeypot as

> *"a security resource whose value lies in being probed, attacked, or compromised."* [Spi02] p.40.

It has similarly been used by [PH08] p.7:

"A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource."[1]

In general there are two types of information that can be gathered using honeypots and which are difficult to acquire otherwise:

1. Novel attack vectors used within attacks, including the actual exploit corresponding to a given attack and
2. the actions performed on a compromised host, which can be monitored and saved for further investigation. In particular honeypots have been proven to be well suited for measuring attacks (and thus propagation) of autonomously spreading malware.

This said, honeypots therefore offer a broad spectrum for using the findings of the observed attacks in order to take appropriate countermeasures. They reach from long-term gathering of attack data, capture of zero-day exploits (i.e., attacks using undisclosed attack vectors and vulnerabilities) and the creation of signatures for the detected novel attacks (improving future detection) up to observing botnets (enabling the identification of infected systems) and automated collection of malware binaries. In addition, honeypots produce much less information than conventional means for attack detection, such as IDS, while the gathered information has much higher value at the same time. Therefore honeypots are increasingly accepted as a legitimate and valuable sensor for attack detection complementing traditional security measures, also in productive environments. However, complexity still remains a big problem, since it seems to hinder extensive deployment outside academia so far (and is the enemy of security beside this). More information on the honeypot concept can be found in [Spi02].

3.3 Honeypot Technology

As the previous section indicated honeypots are not a standardized solution dedicated to solve a specific problem but rather a highly flexible and effective means to detect and track abusive and malicious activities. Depending on the scope and the use case they can be deployed in many ways acting in various roles. Therefore there is a couple of technologies implemented (and often combined) by honeypots. The most common ones are:

- **Darknets**
  A darknet (commonly also referred to as *network telescope*) is a portion of allocated and routed address space while no hosts or services reside in it[2] [3]. Instead all traffic hitting the addresses within the darknet is observed and thus any packet can be threated as suspicious by its presence. Therefore the traffic may be forwarded to one (or more) hosts (e.g., honeypots) which handle this traffic. It offers an insight into unwanted network activities which may result from misconfiguration or malicious actions such as DDoS backscatter (which occurs as a side effect when spoofed IP addresses are used), port scans or malware propagation.

[1] originally posted on the honeypot mailing list
[2] http://www.team-cymru.org/Services/darknets.html Retrieved 2011-09-28
[3] http://www.caida.org/research/security/telescope/ Retrieved 2011-09-28

- **Vulnerability emulation**
  Instead of just passively monitoring the address space (e.g., of a darknet) a honeypot may pretend having a specific vulnerability. It does so by behaving like a given OS when probed remotely using TCP/IP fingerprinting techniques announcing a given software and version. Therefore it is not necessary to emulate a full service or platform but sufficient to emulate the parts of it indicating the vulnerability which is necessary to trick the attacker into believing that he interacts with such a system. This approach goes one step further by provoking an interaction with an attacker or malware thus gathering more information than from just passively monitoring the network traffic.

- **Rootkit techniques**
  The central collection, capture and control of the data gathered by a honeypot is fundamental in order to ensure the integrity of this data. Possible modifications to the honeypot system made by an attacker or a malware may include changes at low system levels. Thus it is necessary that components for data collection and analysis utilized by a honeypot operate at an at least equally low level in order to remain undetected and unmodified. Therefore several means are used, such as a transparent (i.e., IP-less) bridge or - similar to rootkits (refer to 2.3) - a kernel module logging all actions and covertly sending the data to another host.

- **Tarpits**
  A tarpit is a service intended to delay network connections as long as possible. Within the context of honeypots this technique may be utilized to slow down outgoing connections initiated by malware in order to mitigate further propagation or other malicious activities. Another usage is to slow down outgoing SMTP connections of a honeypot acting as open relay (so-called SMTP-tarpit). This makes Spam delivery unattractive while still being able to monitor abusive activities performed via the open relay.

- **Deception**
  In order to distract an attacker from the fact, that he is on a honeypot various deception techniques may be applied tricking the attacker into believing that he is on a real system. The intention is to hold up this illusion as long as possible and thus to track preferably much of his activities. Therefore authentic looking data, such as fake accounts, files, database entries or log files, is placed on the honeypot. As there is no legitimate usage of this data this approach equals the honeypot concept (refer to 3.2). Accordingly this type of data is commonly referred to as *honeytokens*.

- **Auxiliary software**
  In addition to the generic techniques described above several tools have been deployed adding novel and useful capabilities to honeypots, here called auxiliary software. This refers to software that is not a dedicated honeypot but either a commodity software or a a software specifically designed to support honeypot operation. Examples include QEMU[4] [Bel05] (a processor emulator commonly used for virtual ma-

[4] www.QEMU.org Retrieved 2011-10-01

chines) and libemu[5] (providing x86 emulation and shellcode detection dedicated for use within honeypots).

3.4 Classification of Existing Approaches

Within the last years honeypots have received a lot of attention in the research community. Therefore various approaches have been introduced which are classified in the following sections. On a general layer honeypots can be classified by several aspects, such as

- whether they are *client*- or *server*-based,
- their level of *interaction* with an attacker,
- if they are deployed within a *virtual* or on a *physical* machine and
- whether they are running on a productive system or not (*shadow honeypots*) respectively *research* and *production* honeypots (i.e., intended to protect a production network)

Basically there are two types of honeypots with fundamental different approaches:

1. *Server honeypots* are the traditional honeypot variant. They provide a vulnerable machine or parts of it (such as a given service or OS) and wait *passively* for incoming attack attempts whereas
2. *Client honeypots* (synonymously also referred to as *honeyclients*) act *actively*, i.e., they search malicious systems e.g., by visiting websites acting as a vulnerable system in order to find out whether or not they experience attack attempts.

As outlined in section 2.3.3 todays predominant attack vector is client based and therefore sensors that are capable of detecting this type of attack are required. Server honeypots are not able to detect client-side attacks, since they aim at luring attackers to their exposed vulnerabilities (i.e., services). Thus client honeypots play an increasingly important role in detecting state of the art attacks as their approach is at some points fundamentally different compared to server honeypots. Client honeypots simulate or run client-side applications and therefore they need to actively probe (possibly) malicious servers and interact with them. This implies two more major differences: First it is necessary to determine the servers which are suspected to be malicious in an efficient way since probing all servers on the Internet is not feasible. Second, a client honeypot has to determine on its own whether an examined server is malicious or not compared to a server honeypot where every connection attempt can be threatened as malicious (or at least suspicious). More information about client-side attacks, including the differences between these two basic honeypot types, can be found in [SSH+07].

Regarding the level of interaction Spitzner proposed in [Spi02] p.73ff the separation into low-, medium- and high-interaction honeypots. Low-interaction (LI) and high-interaction (HI) honeypots thereby represent the two ends of a broad spectrum that includes various trade-offs regarding complexity, the cost for deployment and maintenance, the type and

[5] http://libemu.carnivore.it/ Retrieved 2011-10-01

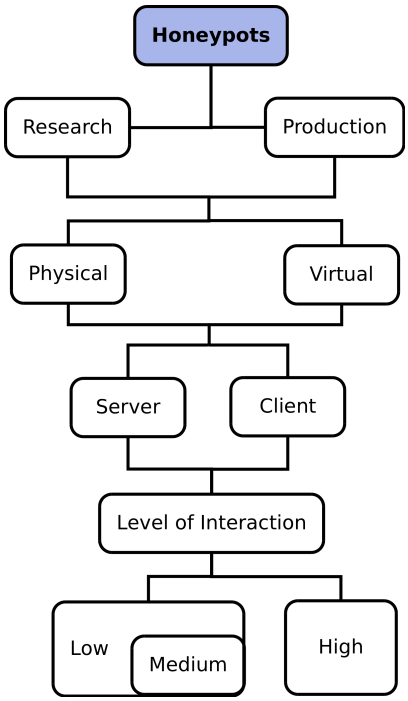

Figure 3.1: General Classification of Honeypots

amount of the gathered information as well as the risk posed by the operation of the corresponding honeypot type. Thus the level of interaction offers one metric for comparing and measuring different honeypots. A LI honeypot has thereby the lowest complexity as it is easiest to deploy and maintain due to its simple design. LI honeypots typically emulate services and vulnerabilities, so that the attacker's interaction is restricted to this emulated functionality. Thus their scope is also limited to simple malicious connection attempts (such as port scans and brute force of logins) and autonomous spreading malware. The operation of LI honeypots poses the lowest risk, since there is little functionality offered which is furthermore just emulated. Therefore it is highly unlikely that such a system gets successfully compromised and abused e.g., for attacks against other systems. In contrast HI honeypots require the highest effort to build and maintain and pose also the highest risk, since they are usually real systems. That is, a HI honeypot exposes an entire (i.e., real) vulnerable system to an attacker to interact with rather than only emulating a functionality, system component, service or vulnerability. Therefore an attacker may be able to fully compromise the system and gain full control over it including the execution of arbitrary commands and usage of his own tools. Thus they offer the highest level of interaction and need therefore to be closely monitored in order to prevent harm to other systems in case the honeypot got compromised. Furthermore they produce a vast amount of information which is two-edged: On the one hand they allow deep insights into attacker's activities and on the other hand the gathered information has to be handled and analyzed. Medium-interaction (MI) honeypots aim at combining the advantages of LI and HI honeypots but are merely an emulated resource rather than a real system. A common usage of MI honeypots is malware collection since therefore it is necessary to provide slightly more

interaction (i.e., allow the connect back to grab the malware sample) than offered by a LI honeypot. More information on medium-interaction honeypots can be found in [Wic06]. However, the boundaries between LI and MI honeypots are often fluid and the terms are not consistently used. For the sake of easiness LI and MI honeypots are therefore summarized in this classification ending in a differentiation between LI and HI honeypots, whose known implementations are briefly introduced in the following sections. Our classification of honeypots is depicted in figure 3.1.

3.4.1 Low Interaction Server Honeypots

Early LI honeypot implementations were mainly intended to act as deception systems and to provide protocol emulation.

The Deception Toolkit (DTK)[6] has been introduced by Fred Cohen in 1998 and is one of the earliest publicly known honeypots. It uses emulation techniques to provide deceptive services on unused ports to attackers probing these ports. It comes with a couple of scripts (written in C and Perl) for usage of a TCP wrapper or specific services such as Telnet or SMTP forging appropriate responses. A notable property is, that it announces itself as the deception toolkit, someone connecting to it, it will be informed that he is connecting to the DTK. It is mentioned for historic reasons here.

BackOfficer Friendly (BOF) was developed by Marcus Ranum in 1998 and released by NFR Security [Spi02]. It is a Windows- and UNIX based honeypot. BOF was originally created to detect Back Orifice scans and has evolved to detect attempted connections to other services, such as Telnet, FTP, SMTP, POP3 and IMAP for which it is able to fake appropriate replies. An outstanding property is its extremely easy usage which enables everyone to instantly run a honeypot by simply downloading and installing the tool.

HoneyPerl by Brazilian Honeypot Project [7], which is based on Perl, is another, similar honeypot implementation emulating common services such as HTTP, SMTP and telnet.

Bait N Switch Honeypot by Team Violating[8] is a similar approach, but intended to be placed within a production network leveraging honeypot technology as an active measure in system defense. Therefore the system reacts to hostile intrusion attempts by redirecting all hostile traffic to a honeypot which mirrors the production system. Thus the intruder is effectively attacking the honeypot instead of the reals system.

LaBrea Tarpit[9] has been written by Tom Liston, originally as a response to the Code Red worm. It is a program that creates a tarpit (also called a "sticky honeypot"). It has become famous for introducing this concept of a tarpit which is an approach to slow down attackers, spammers and worms. LaBrea utilizes unused IP addresses on a network and creates "virtual machines" that answer to connection attempts. It does so by intercepting ARP requests, i.e., watching the ARP request and ARP replies: Once it observes continuous ARP requests without appropriate responses it assumes that the corresponding IP is unused and forges an ARP reply with a bogus MAC address for response to the requester. The pretended virtual machines are then "created" using IP aliasing. The tarpit functionality is

[6] http://www.all.net/dtk/index.html Retrieved 2011-10-18
[7] http://sourceforge.net/projects/honeyperl/ Retrieved 2011-12-09
[8] http://baitnswitch.sourceforge.net/ Retrieved 2011-12-09
[9] http://labrea.sourceforge.net/labrea-info.html Retrieved 2011-10-18

realized through answering to connection attempts in a way that causes the machine at the other end to get ”stuck”. Therefore for an incoming TCP segment with SYN flag set a response is crafted with SYN/ACK set and a small MSS (to enforce sending small chunks). While the connecting instance (i.e., the worm) confirms the three way handshake with ACK (believing that the connection has been established) LaBrea Tarpit only responds to SYN segments and ignores all other arriving segments. This results in a time-out of the TCP connection and due to the small MSS a minimum of generated traffic.

SPECTER [10] is a commercial honeypot implementation announced as smart honeypot-based intrusion detection system. It simulates a variety of OS and (currently 14) common Internet services such as SMTP, FTP, POP3, HTTP and Telnet. The services appear normal to an attacker but are in fact traps intended to log everything and notify the appropriate persons. For the simulated host there are five characteristics available which represent different system behaviors. For example the simulated machine may appear to be badly configured or to be suffering from various vulnerabilities depending on the selected operating system and services. Furthermore, SPECTER automatically investigates the attacker by collecting information and evidence while trying to break in. Therefore it provides so-called “intelligence modules” using e.g., banners, traceroute, HTTP server header and whois. It can also automatically generate custom decoy content (such as documents and password files) and generate decoy programs leaving hidden marks on the attacker’s computer. SPECTER claims to detect and analyze all ICMP, TCP and UDP connections on every port. As the focus of SPECTER is clearly to be a production honeypot automated online updates for content and vulnerability databases are provided allowing the honeypot to change constantly without user interaction.

Further commercial implementations of this honeypot type, intended for use within production networks (often complementing existing security measures such as IDS), are for example **HoneyPoint**[11] by MicroSolved, Inc. and **KFSensor**[12] by Keyfocus.

Symantec Decoy Server[13] (formerly called ManTrap) provides an attacker with a controlled deception environment, called “cage”, which is based on a Solaris system. It therefore creates a copy of the entire Solaris system (as it is installed on the host computer) and places it into a new directory. Only the resulting “virtual OS” is then visible to an attacker and can be stuffed with any application supported by Solaris. It was clearly designed to a act as a production honeypot supplementing existing security solutions such as firewalls. However, as the latest version (3.1) is from 2003 Symantec seems to have abandoned its further development.

Tiny Honeypot (THP)[14] provides simple protocol emulation by utilizing xinetd. It listens on all ports which are not in legitimate use and generates according fake responses e.g., by presenting a service banner or a root shell. It is a single daemon to which all connection attempts are redirected using netfilter rules.

Honeyd, introduced by Niels Provos in 2004 [Pro04], is a small but very powerful daemon that enables the creation of virtual hosts on a network. These hosts can be configured

[10] http://www.specter.com/ Retrieved 2011-10-18
[11] http://microsolved.com/?page_id=9 Retrieved 2011-12-09
[12] http://www.keyfocus.net/kfsensor/ 2011-12-09
[13] http://www.symantec.com/business/support/index?page=content&id=TECH111948 Retrieved 2011-10-18
[14] http://freshmeat.net/projects/thp Retrieved 2011-10-28

to run arbitrary services and pretend to run on arbitrary OS. Service emulation can be achieved either via service scripts (such as Perl or Bash) simulating the behavior of the corresponding service, via Python services or by running the actual UNIX application as a subsystem. When using the script based method the quality of the emulated service heavily depends on the quality of this underlying script. OS emulation is done on the TCP/IP stack level (i.e., the corresponding TCP/IP stack fingerprint from the nmap fingerprint file is used as personality) in conjunction with **arpd** [15]. Arpd is a daemon which listens to ARP requests and generates replies for the unallocated IP addresses. Thus Honeyd enables a single host to claim multiple (i.e., thousands of) addresses for network simulation stuffing them with virtual honeypots and corresponding services. Therefore is usually a portion of unallocated IP address space used. For every single IP address (but also for address ranges) Honeyd can be advised how the simulated, virtual host should behave (i.e., what OS and services should be announced). Honeyd has a number of powerful features, e.g.,

- it can easily spawn a diverse set of numbers of virtual hosts,
- it provides arbitrary services via one configuration file,
- support of subsystems, i.e., real UNIX applications,
- simulation of OS on the TCP/IP stack level,
- support of arbitrary routing topologies.

The design and usage of Honeyd is extensively described in [PH08]. Its architecture is depicted in figure 3.2. An outstanding point is also the excellent performance of Honeyd. Provos tested in [Pro04] that Honeyd can handle about 2000 transactions per second for 65536 instances (i.e., virtual honeypots) on a 1Ghz Pentium II processor. In general arbitrary large networks can be simulated, however the performance depends on the complexity of the emulated services. Because of its easiness, flexibility and performance Honeyd is one of the most powerful, commonly known and widely deployed honeypots actually forming a framework for honeypot creation. Therefore various add-ons for Honeyd exist such as **Honeyd for Windows**[16] (a Windows port providing all capabilities of the UNIX version except subsystems), **HOACD**[17] (a Honeyd based LI honeypot running directly from a CD) or **SCADA-honeynet**[18] (a framework for simulating industrial networks such as SCADA and PLC architectures using Honeyd). Another notable approach is **ScriptGen** [LMD05] which is intended to automatically generate emulation scripts for Honeyd. It will be discussed in more detail within chapter 6.

Malware collection honeypots emulate (vulnerable) services, too but - contrary to other LI honeypots such as Honeyd - they do not implement the TCP/IP stack and the protocols. Instead they focus on the emulation at an application level and just deal with the handling of network sockets. The management of the actual connection is done by the OS. Regarding the level of interaction they might be classified as MI honeypots.

[15] http://www.citi.umich.edu/u/provos/honeyd/ Retrieved 2011-10-18
[16] http://www2.netvigilance.com/winhoneyd Retrieved 2011-10-28
[17] http://www.honeynet.org.br/tools/ Retrieved 2011-10-28
[18] http://scadahoneynet.sourceforge.net/ Retrieved 2011-10-28

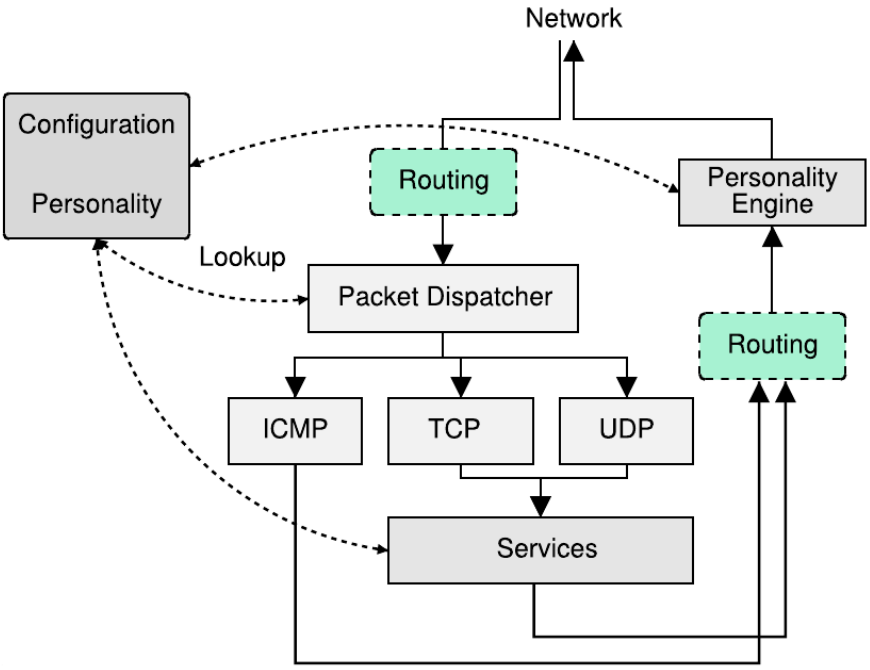

Figure 3.2: A Sketch of Honeyd's Architecture (from [Pro04])

The **Nepenthes** platform [BKH+06] is designed to automatically collect (autonomous spreading) malware by utilizing vulnerability emulation techniques and probably the best known malware collection honeypot. The concept of the Nepenthes approach implements therefore five types of modules with the following functionality:

- The *vulnerability modules* provide the emulation of known vulnerabilities of specific services such as a IIS, WINS or DCOM service. Thereby they emulate only the necessary parts of the corresponding service. The Nepenthes host listens on the appropriate ports for a vulnerable service on one or multiple IP addresses. By utilizing virtual interfaces one host may handle multiple IP addresses on a single interface. Once a connection has been established to a given port the exchanged payload is handled via the corresponding vulnerability module. Thereby each port is restricted to one module impeding the emulation of more than one service per port (e.g., Apache and IIS) at the same time.
- The *shellcode parsing modules* analyze the payload retrieved by the vulnerability modules and extract information about the malware. They do so by XOR decoding of the shellcode and applying pattern matching, such as looking for the URL of the dropper or the actual binary.
- These information is then handed over to the *fetching modules* which try to download the corresponding binary, e.g., in case an URL has been extracted. Currently the fetching modules implement HTTP, (T)FTP and IRC.
- The *logging modules* are responsible for logging the whole work-flow.

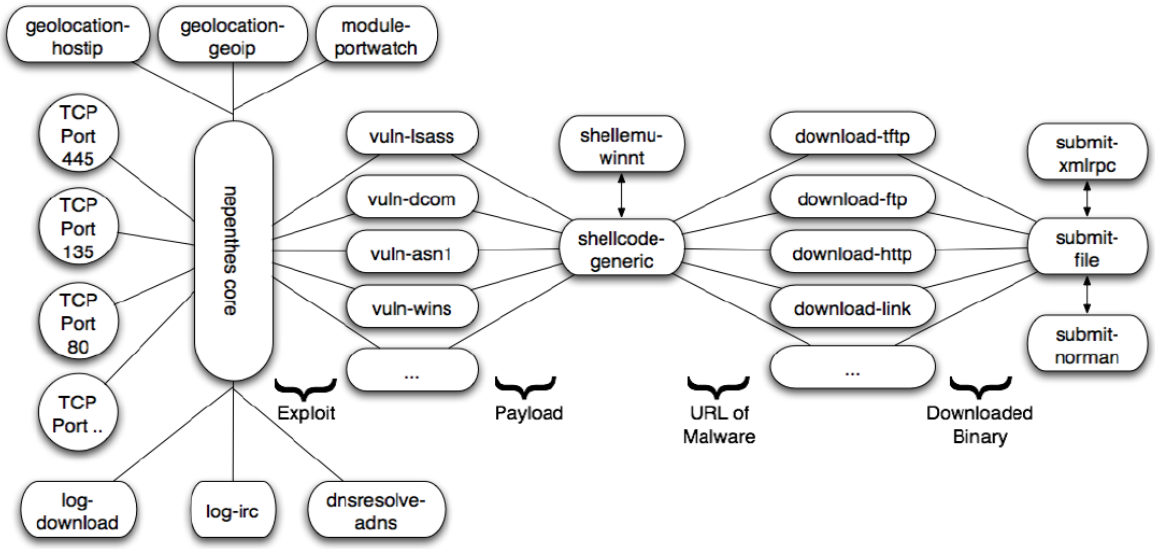

Figure 3.3: The Architecture of the Nepenthes Platform (from [BKH+06])

- If a download was successful the *submission modules* submit the collected malware to a given instance (e.g., the local disk, a database or a third-party service).

The overall architecture of Nepenthes including the work-flow between the various modules is shown in figure 3.3.

However, the approach suffers also from several shortcomings: As the approach is based on the knowledge about existing vulnerabilities it remains unable to handle previously unknown attacks and also heavily depends on the availability of vulnerability module implementations. Currently there are only a few (approximately 15) implementations of vulnerability handling modules available. Especially compared to high-interaction honeypots there is a significant difference in the amount of malware that can be collected. In addition the Nepenthes approach allows only the collection of autonomous spreading malware, i.e., which is self-propagating by scanning for vulnerabilities. Thus it excludes the collection of other malware types such as Trojans or rootkits. Also web-based exploits can not be covered due to the passive nature of the approach. Furthermore Nepenthes is not very stealthy, since it is quite easy to detect by examining the offered open ports and services which may not correspond to a real system and thus are suspicious. Apart from this limitations the Nepenthes approach has been proven to be very useful for collection of autonomous spreading malware and has been implemented within several other tools. **Amun**, which has been introduced by Jan Goebel [Goe08], has basically the same scope and design as Nepenthes. Written in Python it has enhancements in service emulation and supports more vulnerabilities. With **Omnivora**[19] [Tri07] there is also a Nepenthes-like port for Windows available. Furthermore Nepenthes sensors can be united to a distributed network allowing the submission of the collected malware binaries to a central instance accessible to all members of the federation, such as the mwcollect alliance[20].

[19] http://sourceforge.net/projects/omnivora/ Retrieved 2011-10-28
[20] https://alliance.mwcollect.org/ Retrieved 2011-10-28

Honeytrap, written by Tillman Werner, follows a similar approach to Nepenthes while overcoming some of its limitations. It aims at collecting the initial exploit which is then analyzed. The difference to Nepenthes is that it provides a dynamic concept, i.e., it monitors the network stream for incoming sessions and starts appropriate listeners just in time rather than emulating known vulnerabilities. The investigation of the network traffic is done by utilizing a PCAP sniffer, the ip_queue-API or the netfilter_queue. As it does not rely on specific vulnerability emulation modules and instead dynamically handles the incoming data it is capable of handling most (even unknown) network-based attacks. Once Honeytrap detects an inbound connection it dynamically opens an appropriate port to handle the data. Therefore it is not necessary to keep a lot of ports open thus improving stealthiness. Honeytrap offers four different response types which can be configured on a per port basis: (i) Service emulation, (ii) mirror mode, (iii) proxy mode and (iv) ignore mode. Specifically interesting is the mirror mode: Thereby all retrieved data is sent back to the attacker so that Honeytrap acts like a mirror and in the end the attacker is effectively attacking himself. In ignore mode all incoming packets are simply ignored thus being suitable to block certain ports. In proxy mode all traffic is forwarded to another service or host, which may be real or a honeypot. The generic approach and the different modi allow Honeytrap to be deployed as a *meta honeypot*. For further processing of attacks (i.e., saving and analyzing the captured data) several plugins such as parsers for downloading are available. The deployment and usage of Honeytrap is described in [PH08] and on the website[21].

Mwcollectd[22] (current version 4) aims to be a successor of Nepenthes uniting the features of Nepenthes and Honeytrap. Its entire functionality is based on several modules which cover several download modules, a mirror module (similar to the mirror mode of Honeytrap), Python modules for emulating services and command shells, file store, logging and submission modules. The shellcode-libemu module is the key module of mwcollectd and takes advantage of **libemu**[23], a x86 emulation and shellcode detection library. Libemu emulates a x86 CPU and parts of the Win32 API and the Linux process environment. Thus system calls can be simulated in order to gain knowledge about the malicious actions launched on the exploited system. Libemu uses a so-called *GetPC heuristics* (short for 'Get Program Counter'). GetPC represents a sequence of ASM instructions within malicious code, which is necessary to determine the value of the program counter and thus the position in memory. This is required since the shellcode may placed on an arbitrary position within the memory and therefore needs to determine his position in order to get active. By doing so a sequence of e.g., *CALL* and *POP* instructions is executed. Libemu looks for corresponding (suspicious) combinations of such ASM instructions in order to detect shellcode.

Dionaea[24]is another, very similar, Nepenthes successor also using Libemu for shellcode detection and supporting IPv6. It supports common network protocols, whereas the excellent SMB implementation (which is also used by mwcollectd) represents the main protocol. Retrieved shellcodes are executed within a Libemu virtual machine (VM) and several ways to deal with the given payload are provided, including command shells, download-

[21]http://honeytrap.carnivore.it/ Retrieved 2011-10-30
[22]http://code.mwcollect.org/projects/mwcollectd Retrieved 2011-10-30
[23]http://libemu.carnivore.it/ Retrieved 2011-10-30
[24]http://dionaea.carnivore.it/ Retrieved 2011-12-04

ing a file from an URL and executing a single command. Further modules include logging, download and submission.

Multipot[25], authored by David Zimmer, is an emulation-based honeypot which is designed to run on Windows. It aims at capturing malicious code spreading by exploits. The basic approach is the same as introduced by nepenthes. In addition Multipot emulates also vulnerable parts of network services as well as popular backdoors.

HoneyBOT[26] by Atomic Software Solutions is a Windows based honeypot. It opens more than 1000 UDP and TCP listening sockets on the host which are designed to mimic vulnerable services. Uploaded files are stored on the honeypot as well as exploits launched against it. In addition all communication with the attacking host is captured and logged for further analysis.

Billy Goat [PH08] is an approach intended for worm detection by automatically detecting infected machines within a given network. It does so by instrumenting the propagation strategy of a worm (i.e., the scanning for other vulnerable hosts). The assumption is, that within this propagation phase a worm will hit unused IP addresses which a re then handled by Billy Goat. Thereby it utilized vulnerability emulation techniques in order to gain information about the type of exploit that has been faced.

The majority of todays advanced attacks occur at the web application layer. Thus a couple of honeypots were introduced operating at this level intended to capture the corresponding attacks.

GHH - The Google Hack Honeypot[27], introduced by Ryan McGeehan et al., addresses web based attacks driven by attackers utilizing search engines (commonly also referred to as "Google hacking"). That is, adversaries seek for vulnerable web applications which are available on the Internet, such as message boards and administrative web interfaces, using the Google search engine. Due to vast rise of such web applications also the number of vulnerable or misconfigured web applications increased. The reconnaissance of such vulnerable systems, that can be compromised, has been eased by utilizing the power of the Google search engine. In order to detect such attacks GHH emulates a vulnerable web application that is indexed by the search engine. While it remains hidden to regular viewers it is found by searches performed via a crawler. It does so by using a transparent link, i.e., a link which is not detected by normal browsing but found when a given site is indexed (e.g., via a search engine or crawler). As a result, every attempt the GHH experiences can be assumed to be malicious. More details on GHH can be found in [RMEM08] and [PH08].

Another approach, which is similar to GHH, is **PHP.HoP** [PH08]. It is a web based deception framework aiming at capturing web based attacks and web based worms.

Glastopf[28], presented by Lukas Rist et al., emulates a minimalistic web server. Glastopf collects information about web based attacks, such as SQL injection and remote file inclusion. Therefore it investigates incoming URIL strings (e.g., "http://") and attempts to download an examine the retrieved data. After that it generates an appropriate response, which aims to be preferably close to the attacker's expectation. In the end Glastopf receives a malicious binary. An in-depth description on its function and usage can be found

[25] http://labs.idefense.com/ Retrieved 2011-12-04
[26] http://www.atomicsoftwaresolutions.com/honeybot.php Retrieved 2011-12-04
[27] http://ghh.sourceforge.net/ Retrieved 2011-12-04
[28] http://glastopf.org/ Retrieved 2011-12-04

in [RVKM10].

In addition various honeypot implementations exist, which are dedicated to a specific service, device or attack scenario.

For studying attacks against the SSH service there are dedicated honeypots, such as the **kippo SSH honeypot**[29] and **Kojoney**[30] which emulate a SSH server and are intended to (i) observe brute-force attacks against the SSH service and (ii) even more interesting the full shell interaction of the attacker. This includes also a fake filesystem which allows adding and removing files as well adding fake file contents (e.g., /etc/passwd). The session log is stored in an UML compatible format for easy replay with original timings and downloaded files are kept for later inspection. Since the service is emulated some properties of a real SSH service do not work e.g., it only pretends to connect to somewhere and the shell cannot be left (i.e., the exit command does in fact not really exit).

SMTP honeypots are mainly services pretending to be an open relay and used as *spam-traps*[31]. The ultimate goal is to trick the spammer into believing that his Spam has been sent while in fact it was silently dropped rather than delivering the messages. Implementations for this honeypot type include **SMTPot**[32], **Spamhole**[33], **Spampot** [34] and **SWiSH**[35]. **OpenBSD's spamd**[36] attempts in addition to waste the time and resources of the Spam sender. The **Jackpot Mailswerver**[37] is furthermore capable of submitting accurately-aimed complaints which include detailed documentation about the Spam on a built-in web-server.

Furthermore several honeypot implementations exist beside the commodity PC world:

- **Sandtrap** by Sandstorm Enterprises, Inc. [38] is a multi-modem wardialer detector. It logs incoming calls on several lines or emulates open modems by responding with a user-defined banner and login prompt.
- **FakeAP** by Black Alchemy Enterprises [39] imitates lots of 802.11b access points in order to confuse wardrivers.
- Mobile honeypots such as **SmartPot** [FW09] and the **iphone honeynet project**[40] aim at porting honeypot technology to smartphones.

[29] http://code.google.com/p/kippo/ Retrieved 2011-12-04
[30] http://kojoney.sourceforge.net/ Retrieved 2011-12-04
[31] Email addresses which are not intended for communication but rather to lure spam
[32] http://tools.l0t3k.net/Honeypot/smtpot.py Retrieved 2011-12-04
[33] http://sourceforge.net/projects/spamhole/ Retrieved 2011-12-04
[34] http://woozle.org/ neale/src/python/spampot.py Retrieved 2011-12-04
[35] shat.net/swish/swish.c Retrieved 2011-12-04
[36] http://www.openbsd.org/spamd/ Retrieved 2011-12-04
[37] http://parsetext.com/0/files.nsf/0/298c073d26a5a14f882577f5002009ff Retrieved 2011-12-04
[38] http://www.sandstorm.net/ Retrieved 2011-12-09
[39] http://www.blackalchemy.to/project/fakeap/ Retrieved 2011-12-09
[40] http://iphonehoneypot.wordpress.com/ Retrieved 2011-12-09

3.4.2 High Interaction Server Honeypots

HI honeypots are generally real machines and thus any production system, e.g, a workstation, could be considered as HI honeypot as well. Since this has several drawbacks, such as disruption of personal data, a dedicated machine for this purpose will certainly deployed. A common method to so is to setup a *virtual* machine (VM) dedicated to this task using commodity virtualization solutions such as VMware[41], Xen[42] or KVM[43]. Nevertheless, several solutions have been implemented in order to ease the deployment of HI honeypots and adding extra functionality, specifically regarding data capture and monitoring.

User Mode Linux(UML)[44] by Jeff Dike is a port of the Linux kernel to its own system call interface. Thus the kernel itself can be run as user process. That is the Linux kernel spawns a new instance of the Linux kernel as a process which then becomes the guest system. In the end each spawned process is a complete virtual machine which can (almost) not be distinguished from a real system. However, the main drawback is, that due to its nature it can not provide other systems than Linux. Its usage is describe in [PH08].

HoneyBow presented by Zhuge et al. [ZHH+07] is a toolkit intended for malware collection utilizing the HI honeypot principle. Therefore it uses a virtual machine which is running an unpatched Windows and acts as the HI honeypot. The key assumption is, that a successful compromise is detected by observing its effects. That is, once attacked, new files will be created (e.g., self-propagating malware will place a copy of itself on the system.) Thus this approach detects new attacks as well without any need for signatures or (in contrast to LI honeypots) emulation of vulnerable services. Therefore HoneyBow integrates several components:

1. MwWatcher continuously monitors the VM and detects changes e.g., in the file system.
2. MwFetcher seeks for changes (i.e., new files) after a detected compromise, extracts them and resets the VM to a safe state in order to wait for the next attack.
3. MwHunter, based on PE Hunter, is a plugin for Snort (i.e., a dynamic preprocessor) intended to extract Windows executables (PE files) from the network stream.
4. MwSubmitter unites the data (e.g., captured binaries) from a distributed set of MwFetcher instances and submits newly collected binaries to
5. MwCollector, which is a daemon on a central instance. Its purpose is then to store all gathered information and malware binaries in a central database.

By using true vulnerable services rather than emulating them HoneyBow is even capable of collecting malware utilizing zero-days (i.e., the vulnerability exploited for propagation has been unknown before). In addition it is not necessary to study the details of the vulnerabilities in order to implement an emulated version which would be required for the

[41] http://www.vmware.com/
[42] http://xen.org/
[43] http://www.linux-kvm.org
[44] http://user-mode-linux.sourceforge.net/ Retrieved 2011-12-09

LI honeypot approach. However, its main shortcoming (as of all HI honeypots) is the poor scalability compared to LI honeypots. To overcome this limitation the authors integrate HoneyBow in [ZHH+07] with nepenthes.

HIHAT[45] (short for: High Interaction Honeypot Analysis Toolkit) was presented by Mueter et. al. [MFHM08] and is a toolkit that (i) automatically transforms an arbitrary PHP application into a HI honeypot and (ii) semi-automatically analyses the collected data. Hence HIHAT provides the flexibility of HI honeypots (i.e., the full behaviour of the original application) while using only resources from the application level. HIHAT thereby automatically scans for known vulnerabilities and is capable of detecting a wide range of web based attacks such as SQL injections, XSS, file inclusions, command injections or attempts to download malicious files. For further analysis of the gathered data HIHAT offers various features like saving malicious files, data examination (e.g., scanning for new incidents, information on HTTP requests and Cookie data) and generation of statistics (such as traffic statistics and geographical IP based mapping of attack origins). Furthermore they also studied the necessary procedures for advertising (i.e., indexing) the web honeypots in search engines via transparent linking. That is, a link on a regular website pointing to the honeypot which remains invisible to a benign Internet user on the one hand but is still recognized by web spiders crawling websites for search engines on the other hand. In the end the goal is that the announced honeypot becomes part of the hitlist of a given search engine. Since attackers targeting web applications commonly instrument search engines in order to find victims it is thus likely that the honeypot will be probed as well.

ARGOS[46] by Portokalidis et al. [PSB06] is a system emulator intended for use in honeypots. Therefore ARGOS extends QEMU enabling it to detect remote attempts to compromise the emulated guest OS (i.e., the actual honeypot). Due to the use of QEMU ARGOS runs on multiple and supports multiple guest OS, whereas the emulated guest system serves as a virtual honeypot. The basic idea behind ARGOS is to detect the effect of a successful system compromise rather than the malicious code itself which is executed after injection. Therefore ARGOS is deployed in a network and analyzes all incoming network traffic. Once it detects, that a given network input is misused, the assumption is that the system is being exploited and more information is collected about the currently observed attack. A misuse of network input thereby refers to actions intended to redirect the control flow, such as using the network data as jump target or instruction. To detect such attacks influencing the control flow of the target system ARGOS utilizes a technique called *dynamic taint analysis*, which is its key feature:
Roughly spoken that is, any data coming from an untrusted source is tracked throughout execution and any attempt to use this data in an unsolicited way is detected. The technique is based on the fact that any attacker who wants to change the execution flow of a program must substitute a value, which normally comes from a trusted source, with a value derived from his own input. This is the case e.g., when exploiting a buffer overflow by crafting a packet such that it overwrites critical data structures on the stack thus gaining control over the target system. Hence the technique of dynamic taint analysis marks any input data from untrusted sources, such as the network, as tainted since it is considered to

[45] http://hihat.sourceforge.net/index.html Retrieved 2011-12-12
[46] http://www.few.vu.nl/argos/ Retrieved 2011-12-29

be unsafe by default. Specifically, the tainted region is the area in memory where this data is stored. The further program execution is then closely monitored in order to track how the data gets used and thus what other data gets tainted as well. This monitoring is possible due to virtualization, i.e., before data reaches ARGOS it is recorded in a network trace and as ARGOS controls all operations running within the VM it can detect all changes to the tainted data. If this tainted data is copied or used in operations the destination (memory area or register) is tainted as well. For example an add operation between a tainted and an untainted register ends up with a tainted result. Within that monitoring process ARGOS checks, whether tainted data is used in a way that changes the control flow (e.g., used as a jump target). Once such a change (which then represents an attack) is detected, ARGOS logs the memory footprint of this attack. The overall architecture of ARGOS is shown in figure 3.4.

This so created log contains information about the attack including the corresponding physical memory blocks, registers and network trace. A very interesting feature is, that this information can then be used to generate an appropriate signature. Therefore information between the network trace and the memory dump are correlated using two approaches:

1. The longest sequence, which is equal within both the memory footprint and the network trace, is located.

2. The memory location, which was used by the attacker to break into the system, is determined using the physical memory origin and the value of the instruction pointer register (EIP). This EIP value is then located within the network trace and the trace is extended until different bytes are found.

In the end the signature consist of the resulting byte sequence, the encountered protocol and the used port number.

While able to generate signatures out of the observed attacks ARGOS can automatically detect both known and unknown (zero-day) attacks without any need for signatures. The basic advantage of ARGOS is its ability to detect attacks without false positives and independent of the attacked application or the sophistication of the attack. Hence an ARGOS honeypot does not need to utilize deception techniques but can even be advertised. Thanks to the dynamic taint analysis it is also not necessary anymore to distinguish between malicious and benign traffic. The major drawback of the approach is the performance overhead, which is mainly caused by the memory tainting and the tracking of the tainted data. Another limitation is, that the approach only works for unencrypted communication.

3.4.3 Low Interaction Client Honeypots

Contrary to server honeypots, which passively wait for an incoming attack, the basic idea behind client honeypots is to actively look for malicious content. Therefore client honeypots simulate the behavior of a user in order to determine if it is exploited. Generally client honeypots aim at detecting attacks against client-side applications such as web browsers. Thus also passive client honeypots would be thinkable, e.g., based on applications for Email, instant-messaging and P2P-networks. However most currently known implementations are active client honeypots and focus on web based attacks (such as drive-by-downloads) since these attacks are currently considered the most popular ones (see

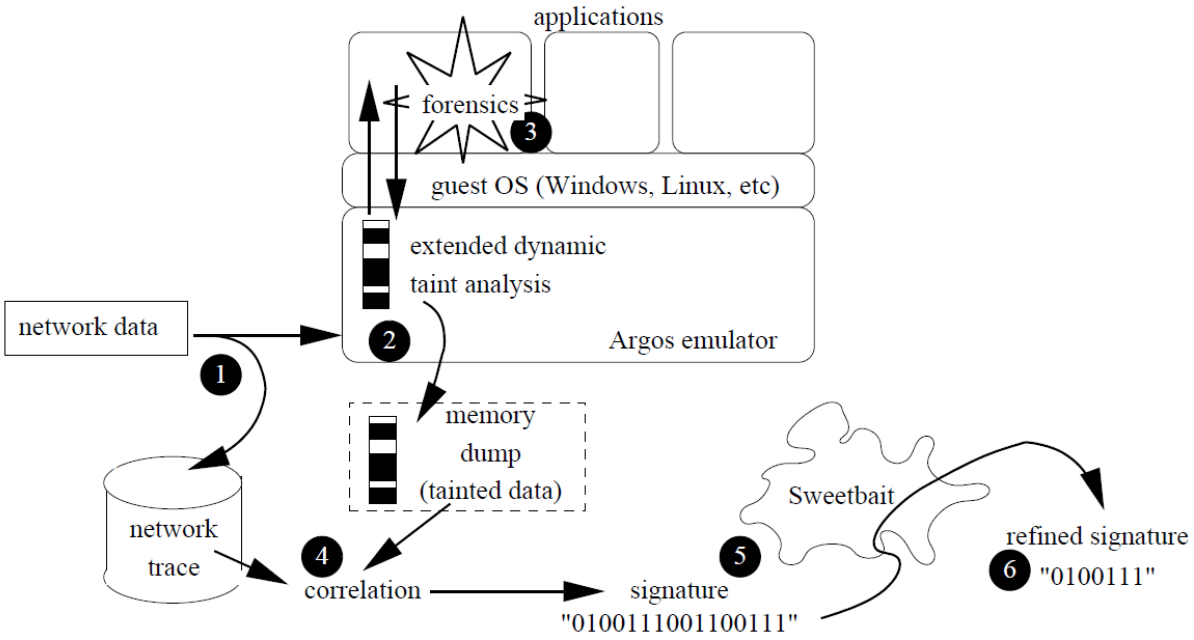

Figure 3.4: General Architecture of ARGOS (from [PSB06])

2.3.3). Thereby the concept of LI honeypots can be applied to client honeypots as well. That is, rather than operating the corresponding real application, a LI client honeypot just emulates the application (e.g., a web browser) or necessary parts of it.

HoneyC [SWK06a] implements this concept by simulating clients which are able to query a response from a server to the extend it is necessary in order to analyze the malicious content. It has a modular design enabling it to simulate different clients, to search for malicious websites in various ways and to analyze the so collected data. The focus is thereby on searching for malicious web servers by utilizing Snort signatures. HoneyC consists of three different components:

1. The queuer creates a queue of suspicious servers which should be analyzed (e.g., via a static list or querying a search engine API).

2. The visitor receives the collected data from the queuer and manages the actual interaction with the corresponding suspicious server. Therefore HoneyC launches a request to the server while simulating a common web browser.

3. All information captured by the visitor is then forwarded to the analysis engine, which checks if violations occurred during the interaction with the server. This is achieved by processing the response from the server with a set of Snort signatures.

Due to its nature HoneyC is not able to analyze sophisticated malicious attempts from websites such as using obfuscated dynamic HTML. Since it relies on Snort signatures fro detection it may also easily being evaded.

PhoneyC [Naz09] is another emulated client which overcomes the limitations of HoneyC by using dynamic analysis which enables it to remove the obfuscation from many malicious websites. Hence it is able to interpret HTML tags for remote links such as iframes as well as scripting languages such as javascript. In addition Similar to Nepenthes PhoneyC implements the concept of vulnerability modules, which rely on ActiveX and are used for exploit detection. Through libemu PhoneyC is also capable of shellcode detection and analysis.

MonkeySpider [IHF08] follows a similar approach and is a framework utilizing many existing freely available software systems. Its basic idea is to first crawl the content of a website and then analyze the crawled content. First, starting URLs for the crawler are generated using various sources such as web search engines, Spam Emails or previously detected malicious websites. Second, the websites are crawled using the Heritrix web crawler[47]. Finally the downloaded content is analyzed using various AV software and malware analysis tools. The main drawback of this approach is the slow content analysis compared to a signature based approach. Thus both tasks (crawling a website and analyzing the content) are thereby separated for performance reasons.

3.4.4 High Interaction Client Honeypots

HI client honeypots are commonly used to crawl the content of a website while using a real resource (e.g., a web browser or OS) rather than an emulated one. Since the operation of HI honeypots poses more risk compared to LI honeypots the resource is typically virtualized (i.e., running within a VM).

HoneyClient[48], presented in 2004 by Kathy Wang, is intended to detect client-side attacks against web browsers, specifically the Internet Explorer. Therefore it utilizes two scripts. The first acts as a proxy whereas the browser is set to have it as a proxy. After crawling a website the second script performs integrity checks of files and registry entries. Once a change has been detected all modifications (additions or deletions) are logged.

HoneyMonkey [WBJ+06] was presented by Wang et al. and is a not publicly available honeypot intended for automated web crawling to discover malicious websites. HoneyMonkey utilizes programs which mimic human web browsing and are executed within virtual machines. The honeypot has also an exploit detection system which does not rely on signatures and is capable of detecting zero-days. The underlying software (i.e., the web browser and the OS) have thereby various patch levels. The detection system has three stages. First, each HoneyMonkey instance visits a given number of websites simultaneously within an unpatched VM. Once an exploit is detected the system goes into the "one-URL-per-VM-mode". That is, each suspect URL runs in a separate VM in order to determine the exploit URLs. Second, HoneyMonkey scans the detected exploit URLs and performs a recursive redirection analysis in order to identify all websites including the corresponding relationships which are involved in the malicious activity. In the third stage HoneyMonkey scans all exploit URLs (which have been detected within the previous stage) using patched VMs in order to detect the attacks which exploit current vulnerabilities. Every URL is thereby visited using a separate browser instance. HoneyMonkey does

[47] https://webarchive.jira.com/wiki/display/Heritrix/Heritrix Retrieved 2012-01-02
[48] http://www.honeyclient.org Retrieved 2012-01-02

not perform any action suited for software installation, such as clicking on a dialog. Thus every entry, which has been created outside the browser (e.g., an executable file or registry entry) can be considered to be the result of an exploit.

SpyBye [PH08] is intended to determine whether a given website has been compromised in order to install malware. Therefore it acts as a proxy and analyzes all fetches that the local web browser makes as a result of visiting a given website. It then classifies every checked URL into one the categories harmless, unknown or dangerous. By showing up unexpected URLs SpyBye provides an indicator for a compromise.

Spycrawler [MBGL06] is a proprietary (i.e., not publicly available) crawler based HI client honeypot, which is dedicated to detection of spyware on the Internet. Utilizing Google search as a starting point Spycrawler follows a three step approach. First it determines whether a given object contains an executable component. This is either achieved via the content-type information within the HTTP header (e.g., application/octet-stream) or based on the URL when it contains known extensions of executables such as exe. Second, the corresponding executable is downloaded and executed within a VM. Finally Spycrawler analyzes whether the previous step led to an infection by using a spyware scanner. The approach has also some disadvantages. First, it relies on only one spyware scanner and only detects known spyware. And second the system will probably miss some spyware, since the first step is not sufficient to detect all spyware.

Beside this other proprietary implementations such as **SiteAdvisor**[49] from McAfee exist.

Capture-HPC [SS06] is another approach to detect attacks against HTTP applications (such as web browsers) by automatically letting them interact with potentially malicious web servers. It consists of one server and multiple clients. The server communicates locally or remotely with the clients, while each client resides in a monitored VM. These VMs are utilized as the actual honeypots and closely monitor unauthorized changes on the system (such as in registry, file and processes) and network activity through modified system calls at the kernel level. By following this approach Capture-HPC is also capable of detecting zero-day attacks. Furthermore the architecture is designed to be fast and scalable allowing to control the Capture-HPC server a number of clients across the network. Beside the detection of a system compromise Capture-HPC is able to extract the malware that has been installed as well making it a valuable tool for the collection of malware which propagates via web browser attacks.

Contrary to most other client honeypots, which focus on web browser based attacks, **Shelia**[50] is designed to detect Email based threats (i.e., Spam Emails). It does so by automatically checking all links and attachments received by Email. The basic idea behind it is to emulate a naive user (i.e., someone who follows every link and opens every attachment that he received from Spam Emails). For each received link Shelia performs a behavioral analysis in order to detect whether a resource is malicious. The outstanding point is, that it - unlike most other comparable honeypots - does not look for changes to the system after a given website has been visited. Instead Shelia tracks the origin of the corresponding operations. That is, once an attempt is made to change given resource (such as in the file system or the registry) Shelia determines whether the call is derived from an area that is not expected to contain code. If this is the case, Shelia considers this as a malicious attempt

[49] http://www.siteadvisor.com Retrieved 2012-01-02
[50] http://www.cs.vu.nl/ herbertb/misc/shelia/ Retrieved 2012-01-02

and raises an alert.

3.4.5 Honeypot Taxonomy

With respect to the classification scheme we defined in 3.4 we apply the following metrics to the listing of honeypot implementations for use within our taxonomy of honeypots:

- The role played in a multi-tier architecture i.e., whether a given implementation is *server* or *client* based.
- The provided level of interaction (*low* or *high*).
- The type of deployment i.e., whether a given honeypot operates on a *physical* machine (indicated with 'P') or runs within (requires) a *virtual* environment (indicated with 'V'). For the sake of easiness we subsume hereby every implementation which does not explicitly require a virtual environment under 'P' although honeypots marked as 'P' might also be operated within a VM for several reasons such as security.
- The targeted attack vector, whereas we differentiate between system (marked as '*SYS*'), service ('marked as '*SRV*') and web ('marked as '*Web*'). SYS refers thereby to implementations which physically offer or emulate a specific hardware device, platform, OS or system specific services. SRV indicates that a honeypot is dedicated to a (given set of) specific service(s) which are used as attack vector, whereas the corresponding service(s) are not necessarily related to a given platform (e.g., HTTP, FTP, IRC). Web summarizes all implementations focusing on any kind of web based attack.
- The source code availability is an important factor as well in this context since the resulting taxonomy is intended to server as a decision basis for the approach presented later on.

While there is some related work proposing a honeypot taxonomy such as Seifert et al. [SWK06b] and Zhang et al. [ZZQL03] we consider our metrics as relevant for the approach presented within this thesis later on. We argue that some of the metrics presented in the mentioned related work (e.g., the used communication interface or the distribution appearance) do not influence the goals of our approach and thus do not need to be considered here. On the other hand they miss information such as availability of source code which is relevant within this context. The following table presents our resulting taxonomy of honeypots. Since many of the mentioned honeypot implementations are evolving research projects and also honeypot research in general is constantly evolving we point out that this taxonomy represents only a current snapshot of the state of the art.

Name	Role in Architecture	Level of Interaction	Deployment	Attack Vector	Source Code available	Reference
Amun	Server	Low	P	SYS/SRV	yes	[Goe08]
ARGOS	Server	High	V	SYS	yes	[PSB06]
BackOfficerFriendly (BOF)	Server	Low	P	SRV	yes	[Spi02]
Bait N Switch Honeypot	Server	Low	P	SRV	yes	fn. 8
Billy Goat	Server	Low	P	SYS/SRV	no	fn. 26,[PH08]
Capture-HPC	Client	High	V	Web	yes	[SS06]
Dionaea	Server	Low	P	SYS/SRV	yes	fn. 24
FakeAP	Server	Low	P	SYS	yes	fn. 39
Glastopf	Server	Low	P	Web	yes	[RVKM10]
Google Hack Honeypot	Server	Low	P	Web	yes	[RMEM08],[PH08]
HIHAT	Server	High	P	Web	yes	[MFHM08]
HOACD	Server	Low	P	SRV	yes	fn. 17
HoneyBOT	Server	Low	P	SYS/SRV	no	fn. 26
HoneyBow	Server	High	V	SYS	yes	[ZHH+07]
HoneyC	Client	Low	P	Web	yes	[SWK06a]
HoneyClient	Client	High	P	Web	yes	fn. 48
HoneyMonkey	Client	High	V	Web	no	[WBJ+06]
HoneyPerl	Server	Low	P	SRV	yes	fn. 7
HoneyPoint	Server	Low	P	SRV	no	fn. 11
Honeyd	Server	Low	P	SRV	yes	[Pro04],[PH08]
Honeyd for Windows	Server	Low	P	SRV	yes	fn. 16
Honeytrap	Server	Low	P	SYS/SRV	yes	fn. 21,[PH08]
Iphone honeynet project	Server	Low	P	SYS	yes	fn. 40
Jackpot Mailswerver	Server	Low	P	SRV	yes	fn. 37
KFSensor	Server	Low	P	SRV	no	fn. 12
Kippo SSH honeypot	Server	Low	P	SRV	yes	fn. 29
Kojoney	Server	Low	P	SRV	yes	fn. 30
LaBrea Tarpit	Server	Low	P	SYS	yes	fn. 9
MonkeySpider	Client	Low	P	Web	yes	[IHF08]
Multipot	Server	Low	P	SYS/SRV	no	fn. 25
Mwcollectd	Server	Low	P	SYS/SRV	yes	fn. 22
Nepenthes	Server	Low	P	SYS/SRV	yes	[BKH+06]
Omnivora	Server	Low	P	SYS/SRV	yes	[Tri07]
OpenBSD spamd	Server	Low	P	SRV	yes	fn. 36
PHP.HoP	Server	Low	P	Web	yes	[PH08]
PhoneyC	Client	Low	P	Web	yes	[Naz09]
SCADA-honeynet	Server	Low	P	SRV	yes	fn. 18

SMTPot	Server	Low	P	SRV	yes	fn. 32
SPECTER	Server	Low	P	SRV	no	fn. 10
SWiSH	Server	Low	P	SRV	yes	fn. 35
Sandtrap	Server	Low	P	SYS	no	fn. 38
ScriptGen	Server	Low	P	SRV	yes	[LMD05]
Shelia	Client	High	P	SRV	yes	fn. 50
SiteAdvisor	Client	High	P	Web	no	fn. 49
SmartPot	Server	Low	P	SYS	yes	[FW09]
Spamhole	Server	Low	P	SRV	yes	fn. 33
Spampot	Server	Low	P	SRV	yes	fn. 34
SpyBye	Client	High	P	Web	yes	[PH08]
Spycrawler	Client	High	V	Web	no	[MBGL06]
Symantec Decoy Server	Server	Low	V	SYS	no	fn. 13
The Deception Toolkit (DTK)	Server	Low	P	SRV	yes	fn. 6
Tiny Honeypot (THP)	Server	Low	P	SRV	yes	fn. 14
User Mode Linux (UML)	Server	High	P	SYS	yes	fn.44

Table 3.1: Honeypot Taxonomy

3.4.6 Honeynets

Several honeypots can be united to a so-called *honeynet* [Pro06]. Honeynets are a network of (diverse) honeypots and are deployed to enlarge the potential attack surface thus increasing the probability of getting hit by an attacker or malware. By doing so they can be used to collect attack statistics and to help quantifying risk. Specifically honeynets substantially increase the effectiveness of malware collection compared to individual honeypots. The specialized network architecture of a honeynet is intended to fulfill the following requirements of a honeynet in a central way:

- *Data control*, which deals with the containment of (malicious) activities within the honeynet. Specifically important is to block any malicious attempt from a honeypot to a third party system.
- *Data collection*, that is the secure forwarding of all gathered data to a central collection point (i.e., a repository or database).
- *Data capture*, which covers the logging and monitoring of all activities within the honeynet such as any action resulting from an attacker or malware initiated outbound attempt.
- *Data analysis* deals with the analysis of all captured information while meeting the appropriate requirements of the corresponding organization.

To achieve this the architecture of honeynets has evolved according to the sophistication of attacks. The first generation (*Gen I Honeynet*) consists of a firewall, complemented by an IDS, acting as gateway and the honeypots reside behind it. Thereby the gateway has two network interfaces (inside and outside). This architecture can thus easily be compromised

since the gateway can be detected (and thus attacked) on the network level. In addition this architecture allows the analysis of unencrypted traffic only.

The second generation (*Gen II Honeynet* [Pro05a]) changed this architecture by introducing the concept of a *honeywall* [Pro05b] which replaces the gateway. The honeywall has three network interfaces (inside, outside and management). It operates as a transparent bridge and only the management interface is accessible on the network layer from a dedicated management network. Thus the honeywall remains hidden to an attacker on the network level and also handles data capture and data control mechanisms. In addition the honeywall concept introduced several other enhancements such as intrusion prevention technology (Snort inline).

The third generation (*Gen III Honeynet*) has the same architecture as Gen II but adds several enhancements regarding the deployment and management of a honeynet and the corresponding data analysis. Specifically it introduces the tool Sebek[51] which is implemented as a kernel module and therefore enables the analysis of encrypted communication by capturing data prior to encryption.

Similarly to the classification of honeypots, which distinguishes between research and production honeypots, we can extend this classification to honeynets. A production honeynet may thereby reside within a production network (either by consisting of honeypots which are distributed within the address space of the production systems or by as an exact copy of the entire production network). Thereby a production honeynet can reveal weaknesses in the production network. Contrary a research honeynet unites different (virtual) machines running various OS and services and is a dedicated network, which is separated from the production network. It servers as an environment to study attacks and weaknesses of the various systems and enables the collection of statistics as well as the detection of unknown attacks.

As a honeynet can contain a diverse set of honeypots several challenges in the deployment emerge, especially in large-scale honeynets. Therefore **Hybrid Honeypot Systems** were introduced, which combine both LI and HI honeypots while incorporating the advantages of both for large-scale honeypot operation. For example, when LI honeypots do not satisfy the needs and the deployment of HI honeypots is too expensive hybrid honeypot systems can be used to combine both principles. That is, the LI honeypots are utilized to act as gateways for the HI honeypots (i.e., to filter out noise). Only interesting traffic is then forwarded to a set of HI honeypots. Examples of such hybrid systems include Collapsar, Potemkin and RolePlayer and are described in [PH08].

3.5 Summary

In this chapter we introduced the concept of honeypots and pointed out their value for intelligence on novel attacks and efficient malware collection. We also summarized the most common technologies implemented by honeypots. Next we classified the existing approaches in general by taking several aspects into account, such as the level of interaction (low, medium, high), the role within a multi-tier architecture (client, server), the deployment (physical or virtual) and the operating environment (production or research). Within

[51] http://www.honeynet.org/project/sebek/ Retrieved 2012-01-03

that we outlined the differences between the basic honeypot types and the corresponding trade-offs. Our classification distinguishes between LI server honeypots, HI server honeypots, LI client honeypots and HI client honeypots. We briefly described the known implementations of each of this four categories with an emphasis on the best known implementations, specifically those dedicated to malware collection. To the extent it is relevant for the thesis we also outlined possible shortcomings of a given implementation. Based on these findings we created a taxonomy of all this honeypots which considers the metrics we have defined (role in architecture, level of interaction, deployment, targeted attack vector and source code availability). We concluded with a brief introduction into honeynets which substantially increase the effectiveness of malware collection compared to individual honeypots.

Chapter 4

Malware Analysis

4.1 Introduction

This chapter deals with the analysis of previously collected malware samples. Thereby two general approaches for malware analysis can be distinguished: Static and dynamic analysis. Both will be introduced within this chapter outlining the corresponding advantages and shortcomings as well as the used techniques. In addition we present the most common tools for dynamic malware analysis focusing on the respective applied techniques. We conclude this chapter with a brief description of the inherent implications and limitations of dynamic malware analysis approaches.

4.2 The Malware Analysis Problem

Malware is the root of many security problems in today's Internet: While in the 1990s an explosive spread of identical malware has been observed (as outlined in chapter 2.3.3), nowadays malware is mostly created utilizing creation kits. As a result we experience a diffusion of malware variants, which are designed to be difficult to identify and analyze. Accordingly a vast amount (i.e., hundreds of thousands) of new malware shows up *per day*. This is illustrated in figure 4.1, which outlines the number of malware samples submitted to Virustotal[1] based on a randomly chosen recent period. Virustotal is a service, which analyzes suspicious files and URLs using multiple AV engines and is thus widely used by malware researchers. Thereby the correlation between the total number of samples and the unique ones is notable, since they correspond to each other. This is related to the fact, that the majority of new malware can be attributed to be just new variants of already existing malware. That is, while most of the malware samples are "unique" (i.e., have a different file hash) they are in fact not. But since they are considered to be unique all of them need to be analyzed. Only once analyzed and the threat posed by a given malware sample is estimated a corresponding AV signature (i.e., a characteristic byte sequence) can be created. As a result malware analysts can not keep up with the vast amount of new malware and hence there is a need to automatically analyze malware and to automatically classify it (for

[1] www.virustotal.com Retrieved 2012-03-15

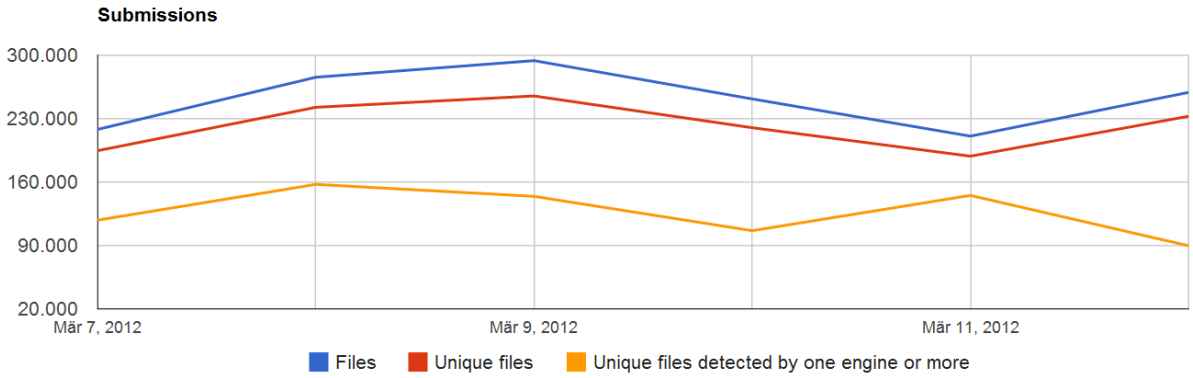

Figure 4.1: Number of Malware Samples Submitted to Virustotal Over a Period of Four Days

example in families). Another issue, that makes malware analysis a very challenging task, is the ongoing arms race between malware authors on the one hand and malware analysts on the other hand. That is, while analysts use various techniques to quickly understand the threat and intention of malware, malware authors invest considerable effort to camouflage their malicious activity and impede a successful analysis [ESKK08]. For analysis we distinguish between two basic approaches, which are introduced in the following sections.

4.3 Static Malware Analysis

Any analysis, that is performed on a given malware binary without executing it, is referred to as static malware analysis [Hol09, ESKK08]. This process of inspecting a given binary without executing it is mostly conducted manually and can be applied on different representations. For example, if the source code is available several interesting information, such as data structures, used functions and call graphs can be extracted. This information gets lost once the source code has been compiled into a binary executable and thus impedes further analysis. Within the malware domain typically the latter is the case, since the source code of a current malware binary is typically not available. Various techniques are used for static malware analysis, which involve, but are not limited to:

- **File fingerprinting:** Beside examining obvious external features of the binary this includes operations on the file level such as computation of a cryptographic hash of the binary in order to distinguish it from others and to verify that it has not been modified.
- **Extraction of hard coded strings:** Software typically prints output (e.g., status- or error-messages) which end up embedded in the compiled binary as readable text. Examining these embedded strings often allows conclusions to be drawn about internals of the inspected binary.
- **File format:** By leveraging metadata of a given file format additional, useful information can be gathered. This includes the magic number on UNIX systems to determine

the file type as well as dissecting information of the file format itself. For example from a Windows binary, which is typically in PE format (portable executable) a lot of information can be extracted, such as compilation time, imported and exported functions as well as strings, menus and icons.

- **AV scanning:** If the examined binary is well-known malware it is highly likely to be detected by one or more AV scanners.
- **Packer detection:** Nowadays malware is mostly distributed in an obfuscated form e.g., compressed or encrypted. This is achieved using a packer, whereas arbitrary algorithms can be used for modification. After packing the program looks much different from a static analysis perspective and its logic as well as other metadata is thus hard to recover. While there are certain unpackers, such as PEiD[2], there is accordingly no "generic" unpacker, making this a major challenge of static malware analysis.
- **Disassembly:** The major part of static analysis is typically the disassembly of a given binary. This is conducted utilizing tools, which are capable of reversing the machine code to assembly language, such as IDA Pro[3]. Based on the reconstructed assembly code an analyst can then inspect the program logic and thus examine its intention. Usually this process is supported by debugging tools such as OllyDbg[4].

More information about static malware analysis techniques, including practical insights, can be found in [Szo05], chapter 15.4, p.571ff and [Ken07].

The main advantage of static malware analysis is, that it allows a comprehensive analysis of a given binary. That is, it can cover all possible execution paths of a malware sample. Additionally, static analysis is generally safer than dynamic analysis (apart from accidentally executing the malware sample) as the source code is not actually executed. However, it can be extremely time-consuming, cumbersome and thus requires expertise. The main drawback of static malware analysis is, that it suffers from code obfuscation and packing techniques, which are commonly employed by malware authors to impede the extraction of semantics. The easiest way to evade the unpacking issue is to actually execute the binary, which leads to dynamic malware analysis.

4.4 Dynamic Malware Analysis

Executing a given malware sample within a controlled environment and monitoring its actions in order to analyze the behavior is called dynamic malware analysis [Hol09, ESKK08]. The thereby monitored behavior can then be matched with findings from a previous static analysis e.g., a disassembled object ([Szo05] p.585). Since it is performed during runtime (and malware unpacks itself), dynamic malware analysis evades the restrictions of static analysis (i.e., unpacking and obfuscation issues). Thereby it is easy to see the actual behavior of a program. Another major advantage is, that it can be automated thus enabling

[2] http://www.peid.has.it/ Retrieved 2012-03-17
[3] http://www.hex-rays.com/products/ida/index.shtml Retrieved 2012-03-17
[4] http://www.ollydbg.de/ Retrieved 2012-03-17

analysis at a large scale basis. However, the main drawback is so-called *dormant code*: That is, unlike static analysis, dynamic analysis usually monitors only one execution path and thus suffers from incomplete code coverage. In addition there is the danger of harming third party systems, if the analysis environment is not properly isolated or restricted respectively. Furthermore, malware samples may alter their behavior or stop executing at all once they detect to be executed within a controlled analysis environment.

On a general layer two basic approaches for dynamic malware analysis can be distinguished ([Hol09]):

1. **Analyzing the difference between defined points:** That is, a given malware sample is executed for a certain period of time and afterwards the modifications made to the system are analyzed by comparison to the initial system state.

2. **Observing runtime-behavior:** Thereby all activities launched by the malicious application are monitored during runtime using a specialized tool.

An example for the first approach is *Truman (The Reusable Unknown Malware Analysis Net)* [5]. Thereby malware is executed on a real Windows environment rather than within a VM. During runtime Truman provides a virtual Internet for the malware to interact with. After execution the host is restarted and boots a Linux image, which then mounts the previously used Windows image in order to extract the relevant data, such as the Windows registry and a complete file list. Finally the Windows environment is reset to its initial clean state. By using a native environment Truman is able to circumvent possible anti-debugging measures of malware. However, since the result is only a snapshot of the infected system, information related to dynamic activities such as spawned processes and temporarily created files are lost.

Hence observing the runtime-behavior of an application is currently the most promising approach. It is mostly conducted utilizing *sandboxing*. A sandbox hereby refers to a controlled runtime environment which is partitioned from the rest of the system in order to isolate the malicious process. This partitioning is typically achieved using virtualization mechanisms on a certain level. While in principle existing tools, such as chroot or BSD jails could be used to deploy such a controlled environment several sandbox environments dedicated to malware analysis exist implementing specialized techniques.

4.4.1 Techniques

Common basic techniques for analyzing an application during runtime include debugging, process monitoring, registry change tracking or sniffing network traffic and are described in [Szo05] chapter 15.4.4. In addition there are several approaches which are used specifically for dynamic malware analysis. The most common ones are introduced in the following sections. An in-depth survey of the different techniques as well as ways how to implement them can be found in [ESKK08].

[5] http://www.secureworks.com/research/tools/truman/ Retrieved 2012-03-26

Function call monitoring and hooking:
Functions are typically used to allow code re-usability for a given task. Thus it is worth to track the functions called by a given program in order to gain an overview about its semantics [Hol09, ESKK08]. Intercepting these calls is one possibility to track what functions are called. This process of intercepting the function calls is named *hooking*. Thereby the analyzed application is manipulated to that effect, that an additional function, the so-called *hook-function* is called, which implements the analysis routines e.g., recording the traced activity to a log file.

A common type of hooking for dynamic malware analysis is so-called *API-hooking*: An application programming interface (API) groups together a set of common functions. Several APIs on different layers are typically present in an OS in order to provide applications with a standardized interface for common operations.

Microsoft Windows for example offers the so-called *Windows-API*[6] which is a set of APIs each dedicated to a specific category (e.g., networking). The Windows-API can be used to access the Windows resources, such as processes or the registry, and consists of several DLL files (e.g., kernel32.dll and user32.dll). Applications utilize the Windows-API rather than making direct system calls. This applies to any malware executing in user space as well, since it needs to invoke the respective system calls for privileged operations. Thus the Windows-API can be instrumented for dynamic malware analysis by monitoring all relevant system calls and their parameters [WHF07, Hol09]. Thereby the Windows-API functions do not make direct system calls themselves but are in turn wrappers to the so-called *Native-API* which is implemented within the file ntdll.dll. The intention of the Native-API is to insert another layer of abstraction in order to increase the portability of applications. That is, while the functions of the Native-API change across different Windows versions and service packs, the functions of the Windows-API remain stable.

Thus the hook-function is particularly applied on the API level as follows [ESKK08]: In commodity OS every process, that wants to perform operations on system resources, must switch from user mode (ring 3) to kernel mode (ring 0). In Windows this is done by the Native-API. The actual switch to kernel mode can be done in several ways, such as via software interrupts (e.g., INT 0x2e in Windows) or via processor specific commands (e.g., SYSENTER for Intel processors). Afterwards the Windows kernel (i.e., ntoskrnl.exe) is in charge with taking control, executing the appropriate operation and returning to the calling address. If now a hooked function is going to be executed, control is redirected to the hook-function, which performs its custom operations. Afterwards it may redirect control to the API function again or even prevent its execution at all. In addition this technique allows the analysis or even modification of the parameters of a given call. Thereby it is hard for the calling application to recognize, that the API function has been hooked and hence the hook-function has been executed instead of the original one. While malware authors could bypass API hooking by calling the native functions or even the system calls directly this is not a feasible approach, since they need to know the target platform and thus must rely on the abstraction provided by the Windows-API to ensure portability of their code. Since there is no official documentation of the Native-API and the system call interface the design of malware that invokes system calls directly would require in-depth knowledge about Windows internals. However malware that gained kernel mode priv-

[6] http://msdn.microsoft.com/en-us/library/ff818516(v=vs.85).aspx Retrieved 2012-04-18

ileges may not need to invoke system calls and could thus bypass this analysis method. Beside the Windows-API there are several other points where a call between an application and the kernel can be intercepted. For example inside the Windows kernel this can be achieved using the Interrupt Descriptor Table (IDT) or the System Service Dispatch Table (SSDT) [Hol09].

Beside the actual function call itself also the parameters and their relations can be taken into account according to [ESKK08]. This so-called dynamic **function parameter analysis** tracks the values and return values that are passed. This values allow the correlation of different function calls, which are related to the same object (e.g., a file). Grouping these events provides an insight into the malware behavior from an object-related perspective. Another valuable information source is the **instruction trace** i.e., the sequence of machine level instructions. Since the therein included information is not present in higher levels, such as the function calls, an instruction trace may reveal additional information for examining the behavior of a given malware sample. **Autostart extensibility points** are used to automatically launch a given application after the system has booted. Since malware aims to persist after reboot analyzing these points can be valuable as well.

Information Flow Tracking
Independent of the function call monitoring approach is information flow tracking, which addresses the aspect of how a given malware actually handles the data. That is, data considered to be ”interesting” and all manipulations to it is tracked throughout the whole execution. Thereby the monitored data is labeled (*tainted*) with a corresponding label. Once the data is processed this taint label is transmitted as well. Thereby a so-called *taint source* introduces new labels. That is, it taints data which is considered to be interesting by the analysis component. A so-called *taint sink* on the other hand reacts to this tainted data. For example a reaction could consist in raising an alert once tainted data is processed in an abnormal way. In order to handle the correlation between tainted data and the resulting action appropriate policies have to be defined. Hence this approach is alike the dynamic taint analysis instrumented within the Argos HI honeypot (see chapter 3.4.2). Another notable approach that uses dynamic taint analysis for automated analysis of binaries is the BitBlaze project [SBY+08], which is introduced below.

Virtual Machine Introspection
Another approach which differs from the ones described above is virtual machine introspection (VMI). It is a method for monitoring and inspecting the inner state of a virtual machine (VM) from the hypervisor level, i.e., the virtual machine monitor (VMM). Garfinkel and Rosenblum [GR03] coined the term for usage in the intrusion detection domain. In the context of tracking malicious activity the VMI approach offers the following main advantage: While conventional, host based, intrusion detection measures provide good insights of what is happening on an infected system they can be detected and thus evaded by malware. In contrast a network based intrusion detection system is more resistant against attacks, but has a limited view on the events inside an infected system. VMI combines both advantages i.e., a good attack resistance while keeping a good insight of events that happen on the compromised host. A key problem in the area of VMI is what is called ”bridging the semantic gap”: Since the hypervisor has no knowledge about the structure

of the underlying OS, it needs to bridge the informational gap between the hypervisor's and the VM's perspective. That is, it needs to derive the corresponding information, such as current network connections and running processes, from the raw data in the memory of the VM. Commonly this is achieved accessing the (physical) memory of the VM thereby using debug information in order to identify the relevant structures. Therefore VMI instruments hardware based information extraction in order to extract the relevant information from the underlying OS (e.g., memory pages, states and registers). The knowledge of these events is then used to map the information to OS semantics. Hence VMI enables monitoring of the hardware and by this derives the software state.

VMI can be very well applied to security applications and in particular in the context of malware analysis it offers several advantages compared to other techniques:

(i) Similar to a whole system emulator, such as QEMU, VMI provides a very good isolation between the system which conducts the malware analysis and the system which got infected with the malware. Thus the victim machine is monitored "from outside" and the infected system does not pose risk to the analysis system. In addition security measures within the hypervisor remain isolated from attacks which occur inside the VM while the hypervisor maintains a complete view of the VM's system state.

(ii) Since VMI remains hidden to the guest OS, it offers an insight into a host, even if it has been completely compromised. This is not the case using measures which rely on trusted supportive components within the VM.

(iii) In addition the infected host can be suspended at any time to perform operations necessary for analysis, e.g., extract and modify data or reset it to a clean state.

(iv) Furthermore unwanted operations can be suppressed, such as malware setting the network interface into promiscuous mode.

More information about VMI techniques can be found in [NBH08]. However, malware authors may employ techniques to detect the VMI environment and thereby hinder analysis. Such environment sensitive malware could for example adapt its behavior or refuse to execute at all.

4.4.2 Tools

Various tools and frameworks for malware analysis have been presented over the past years. In this section we introduce those which are best known in the research community. For each tool we present a brief summary focusing on the applied techniques in order to outline the respective advantages and drawbacks of the different approaches.

Willems et al. [WHF07] presented **CWSandbox** (CWS, now called GFI Sandbox[7]), which executes a given binary in a simulated environment, monitors all system calls and generates a corresponding report. CWS is designed to capture the behavior of malware samples with respect to file system manipulation, registry manipulation, operating system interaction and network communication. Therefore the analyzed sample is executed in a controlled environment, either natively or in a virtual Windows environment. This virtual system is a full installation of a Windows OS which is used to execute the sample together with the analysis components. Finally a comprehensive XML report is generated which

[7] http://www.gfi.com/malware-analysis-tool Retrieved 2012-04-03

reflects the recorded behavior. The analysis functionality is implemented utilizing hook functions which perform monitoring on the API level and *inline code overwriting*:

For API hooking the analyzed sample is rewritten once it has been loaded into the memory. The thereby implemented hooking technique includes a monitoring function, which can perform analysis tasks before and after every API call. Hence CWS traverses an initialization phase before the actual analysis. That is, first the malware binary and all library dependencies are loaded so that CWS can examine all exported API functions of the libraries and insert the appropriate hooks. Afterwards the analysis is started and for every API call, which is invoked during execution, the control flow is redirected to the corresponding hook function, which analyzes the parameters. The hook then executes the backed up instruction and resumes execution of the original API function. Finally, after the API function has been executed, control is redirected again to the hook function which is then able to process the results of the API call.

Due to the use of inline code overwriting the code of the API functions, which resides within the loaded DLLs, is overwritten directly and all calls to these APIs can be redirected to the hook function. However it is necessary to patch the application before analysis, which is mostly conducted using the so-called *DLL injection* [Hol09].

Since the used techniques are applied within the execution context of the malware, CWS relies on supportive components within the VM. In particular the two main components of CWS, cwsandbox.exe and cwmonitor.dll, have to be integrated in the VM in order conduct malware analysis. Thus CWS applies rootkit techniques to camouflage its presence by hiding all system objects, which could reveal CWS to the analyzed sample. To this end the functions used to query the corresponding system details are hooked as well. In particular the controlling process and the injected DLLs are removed from the respective lists in order to hide their presence from the analyzed malware sample. In order to increase the invisibility, a vast amount of communication takes place between the two main components using IPC. More information about CWS including its design and internals can be found in [Hol09, PH08].

Several other tools using the technique of API hooking, such as the **Joe Sandbox**[8] and the **Cuckoo Sandbox**[9], exist but differ in several aspects, e.g., architecture or functionality. For example the **Norman Sandbox** [Nor09] emulates a whole computer and a network connected to it. Therefore the Sandbox emulates a complete environment, including hardware, software and all core components of the Windows OS, which have been rewritten for this purpose. The emulation has the advantage of increased transparency for the malware sample. However, compared to approaches, which rely on a real OS environment, the malware can not interact (e.g., modify) with other processes, since the environment is just emulated.

An approach, which utilizes the technique of information flow tracking is the **BitBlaze project** [SBY+08]. Moreover, it combines measures for dynamic malware analysis with static analysis techniques in order to broaden the gained insights on the behavior of a given malware binary. Therefore it is split into three components:

[8] http://www.joesecurity.org/ Retrieved 2012-04-03
[9] http://www.cuckoobox.org/ Retrieved 2012-04-03

(i) Vine, the static analysis component, consists of a platform-specific front-end and a platform-independent back-end. In addition it introduces a platform-independent intermediate language (IL), which represents the assembly languages. That is, ASM instructions of the underlying architecture are translated into the IL via the front-end. The actual analysis is conducted on the back-end utilizing the platform-independent IL. This enables the component to handle various architectures, such as x86 or ARM.

(ii) TEMU is the platform for dynamic analysis and is based on QEMU. It emulates a complete system, including OS and applications. A malware sample is executed within this environment and its behavior is analyzed using the so-called semantics-extractor (which derives OS level semantics from the emulated system) and the taint analysis engine (which performs dynamic taint analysis). A given binary can thereby be analyzed in both Windows and Linux systems. Additionally the state of the machine can be saved or restored at any point in time easing multi-path examination.

(iii) Rudder performs mixed concrete and symbolic execution at the binary level. That is, it can be fed with a given malware sample and symbolic inputs. During execution it explores multiple execution paths whenever the path depends on the symbolic input. Thus Rudder is capable of automatically revealing hidden execution paths, which are only entered, once a certain condition occurs.

Anubis[10] (Analyzing Unknown Binaries) comprises a couple of techniques for dynamic malware analysis. It is based on TTAnalyze [KKB06] and executes the analyzed malware sample in an emulated Windows XP which resides within a QEMU VM. As a result it generates a report summarizing the tracked behavior. Apart from analyzing samples Anubis is also capable of clustering malware samples into families according to their behavior [BCH+09].

The analysis functionality includes VMI and the invocation of Windows API calls as well as system calls to the Windows Native API. In addition it records the corresponding network traffic and tracks data flows. As Anubis executes the malware sample within a complete Windows environment it needs to focus on the operations launched by the sample while discarding all other operations, which are issued during OS operation (e.g., operations of other processes). To this end Anubis takes advantage of the fact, that Windows assigns a separate page directory to each running process. Thereby the physical address of this page directory (i.e., of the currently running process) is stored in the CR3 CPU register whose value is used by Anubis to perform the analysis for this process only. In addition Anubis monitors the functions, which are used for the creation of new processes, and is thus able to consider processes that have been spawned by the malware as well.

The technique of function call monitoring is realized by comparing the instruction pointer of the emulated CPU with the (known) entry points of the monitored functions. Therefore Anubis keeps a list of functions to be monitored and their corresponding entry points. Additionally the parameters of the tracked calls are examined.

Another technique used by Anubis is function call injection, which allows alteration of the execution during runtime or to inject own code into the running process.

The approach taken by Anubis allows also to address the issue of multiple path exploration in dynamic malware analysis. Therefore Moser et al. [MKK07] introduced a tool,

[10] http://anubis.iseclab.org Retrieved 2012-04-05

which implements dynamic taint tracking to analyze the data flow between function calls and derive data dependencies. However, the utilized techniques make Anubis prone to detection by the malware, e.g., due to timing issues or bugs in the emulation environment.

Overcoming such limitations was the motivation for Dinaburg et al. to deploy **Ether** [DRSL08]. In particular the authors state, that existing malware analysis tools suffer from detectability issues, i.e., that malware is able to recognize, that it is monitored. Ether is a framework for transparent malware analysis that leverages hardware virtualization. It is implemented within the Xen hypervisor, which operates at a higher privilege level than the monitored guest system in the VM. Thus Ether remains transparent to any malware which is executed within this guest system. To this end Ether is capable of monitoring executed instructions and memory operations. For Windows XP system calls can be traced and the analysis can be limited to a specific process.

In order to monitor the executed instructions Ether takes advantage of the CPU's trap flag. That is, once code within the VM is executed, this trap flag is set raising a debug exception after every executed machine instruction. In this way Ether is able to trace the instructions, which are executed by the monitored process within the VM. As environment-sensitive malware may check the trap flag as an anti-debugging technique in order to determine, whether it is analyzed Ether monitors all instructions accessing the CPU flags and modifies them accordingly so that the expected result is presented to the guest system in the VM. Therefore a shadow version of the trap flag is maintained.

In order to monitor write operations to the memory Ether sets all page table entries to read only. Once the guest system in the VM attempts to execute a write operation a page fault occurs, which enables Ether to check, whether the page fault resulted from a normal event (e.g., swapping). If this is the case, the fault is passed. Otherwise Ether assumes a memory write by the guest system and performs the according analysis steps. Thereby Ether modifies the settings of the shadow page tables. Thus this changes can not be detected by any software running within the guest system, since they are not visible to the guest OS.

The monitoring of system calls is accomplished by modifying the value of the SYSENTER_EIP_MSR register so that it points to an unmapped memory region. This results in a page fault once this address is accessed and Ether knows, that a system call was invoked. It can then restore the original value and execute the system call. Since Ether has full access to the memory of the guest system it can analyze the corresponding parameters of the system call. Afterwards the register is set to an invalid address again in order to capture the next system call. The SYSENTER_EIP_MSR register contains the memory address of the code, which is responsible for dispatching information on system calls. That is, in order to perform a system call its number as well as the corresponding parameters are loaded to the respective registers and afterwards the SYSENTER instruction is executed. As a result the execution continues at the memory address, which is stored in the SYSENTER_EIP_MSR register, whereas this code is executed in kernel mode. By instrumenting this register Ether is able to derive information about the actual system call number and its parameters. In addition Ether is capable of handling system calls, which are invoked via the deprecated INT2E instruction in a similar way.

The approach of Ether enables it to remain undetected by malware and thus circumventing any anti-debugging measures of modern malware, except those relying on an external

information source, such as timing issues. However, at the time of writing Ether's analysis capabilities are limited to Windows XP only. In addition Ether is based on the Xen hypervisor, whose support had been removed from the Linux kernel. While Xen re-entered the Linux kernel in recent versions, unfortunately Ether does not seem to be maintained anymore.

The limitations of Ether have been addressed by Pfoh et al. in [PSE11]. The authors introduced **Nitro**, a VMI based, evasion resistant framework for system call tracing and monitoring. The main difference between Ether and Nitro is the utilized hypervisor. Ether is based on the Xen hypervisor, while Nitro builds upon the Linux Kernel Virtual Machine (KVM). KVM consists of two components, (i) a user application, which is based on QEMU, and (ii) a set of Linux kernel modules.

Nitro takes a similar approach as Ether, i.e., causing an exception to realize the monitoring functionality. While the system call tracing mechanisms are similar as well, Ether's output differs to the extent, that it is specific for the guest OS (i.e., Windows XP). Since popular architectures, such as x86, do not support trapping to the hypervisor as a result of a system call, Nitro triggers this trap indirectly. In particular it takes advantage of system interrupts (e.g., general protection faults), because trapping for these events is supported by the Intel Virtualization Extensions (VT-x). As a result, system calls can be effectively trapped, although this is not natively supported by the hardware extensions. To this end Nitro implements trapping for all three system call mechanisms, namely system calls based on (i) interrupts, (ii) the SYSENTER instruction and (iii) the SYSCALL instruction.

For example, interrupt based system calls are trapped by leveraging the fact, that interrupts are handled via the IDT (in x86), whereas the IDTR is consulted to locate the appropriate handler. The Intel VT-x extensions provide mechanisms to trap system interrupts i.e., the interrupts from 0 to 31. However, they do not provide a mechanism to trap user interrupts (32 and above), which are used for system calls. Hence Nitro copies the IDT of the guest OS into the hypervisor and manipulates it to the extent, that system interrupts are left unaffected while user interrupts cause a general protection fault. Afterwards all faults are forwarded to the hypervisor, which is in turn natively supported by Intel's VT-x extensions. By inspecting the interrupt number (i.e., below or above 31) it is then distinguished, whether the fault was generated as a result of invoking a user interrupt. If that is the case, Nitro determines, whether a system call has been trapped and if so starts to collect data. The procedure for the other system call mechanisms work in a similar way.

Rather than providing a fixed set of data per system call, Nitro allows to specify rules for controlling the data collection during the system call tracing process. That is, Nitro can be instructed where a given guest OS stores specific information, such as the system call number. Moreover Nitro is able to identify a process with the help of the value stored in the CR3 register. This value points to the address of the top-level page directory and is unique for a single process.

Since Nitro is a purely VMI based system, is provides excellent attack resistance. That is, it remains isolated from attacks launched from the guest OS, since it is not visible from therein running applications and is furthermore resistant to evasion attempts due to the so-called hardware rooting. This means, that a given data structure, which is specified by the hardware architecture, can not be influenced by an attacker or malware, because

otherwise the hardware will not run correctly.

In addition Nitro is guest OS portable, since the used mechanisms utilize knowledge about the hardware. Hence the mechanisms work for any guest OS, which is compatible with the supported architectures (i.e., x86 or Intel 64) while any guest OS must use these hardware mechanisms according to the specifications.

After all Nitro is extremely flexible, since it supports all three system call mechanisms, is guest OS portable and evasion resistant. Furthermore it outperforms Ether in both, functionality and performance, as demonstrated in [PSE11].

4.4.3 Implications and Limitations

Generally, the design and operation of a dynamic malware analysis system poses several inherent implications, that have to be considered. They are listed in the following section for the sake of completeness.

Implementation level:
A system for dynamic malware analysis can be implemented on different levels (e.g., in user space or kernel space, as an emulator or within a VM). In general, the utilized analysis techniques have to apply at least on the same level as the malware operates in order to circumvent possible anti-debugging measures. Depending on the design decisions, the behavior of a malware sample can be recorded on several levels, such as system calls, library calls, API calls or on the machine instruction level, each providing a different detail of information. An overview on the various implementation strategies (i.e., how are the techniques actually realized and to what level do they apply) can be found in [ESKK08].

Duration of analysis:
Another implication is the duration of the analysis i.e., how long a given malware sample is actually executed in order to deduce its threat from the observed behavior. In general, this time period may reach from a few seconds (e.g., for examining the bootstrapping behavior) up to several weeks (e.g., for botnet measurement or to observe a Spam campaign) [KWK+11]. However, a given malware binary may expect to get captured and analyzed and thus suspend for a given period of time or wait for another external event before it gets active. Apart from conducting a comprehensive (static) analysis one can never exclude, that the currently analyzed malware sample incorporates such behavior. On the other hand, since dynamic malware analysis aims towards throughput, estimating the execution period is always a trade-off. In the context of dynamic malware analysis there is a broad consensus, that malware must be analyzed at least for a period of minutes. For example, the authors of [KWK+11] found, that a notable amount of captured malware requires more than three minutes on average for propagation. Thus they state, that a long-duration execution is important to observe the desired behavior of the malware sample.

Dormant code:
A general limitation that remains to dynamic malware analysis is, that in general only one execution path is covered. While work has already been done to overcome this this limitation [MKK07], it is not applicable to all approaches due to the used techniques.

Evasion techniques and the arms race:
Dynamic malware analysis performed using commodity virtualization solutions can be recognized and thus evaded by malware [Fer06]. Such environment sensitive malware is thereby not a theoretical issue as indicated by [CAM+08, Rut04]. Moreover anti-debugging measures[11] employed by malware authors and the resulting arms race affected the design of dynamic malware analysis tools as well in past years [LKMC11]. Hence keeping the analysis process preferably transparent to the malware remains an important issue.

Containment and Internet connectivity:
The common approach of dynamic malware analysis (i.e., to execute a malware sample within a controlled environment in order to study its behavior) exposes a dilemma. On the one hand malware should ideally have full Internet access in order to be able to operate as intended. On the other hand the analyzed malware sample must be isolated during execution such that it is hindered in performing malicious tasks and thus harming third party systems on the Internet. However, if executed in full isolation the malware will be unable to contact external hosts via its C&C channel to obtain further instructions or receive further data. Thus a fundamental requirement for dynamic malware analysis is Internet connectivity since otherwise malware may behave different or refuse to continue execution at all. As a result the real behavior of the malware can not be observed. The risk of damaging third party systems can be minimized by redirecting malicious traffic via sinkholing approaches. While this has been addressed in several work (e.g., [KWK+11]), an essential problem that remains is to examine outgoing requests and reliably determine whether they are malicious or benign.

4.5 Summary

In this chapter we introduced the two general approaches for malware analysis i.e., static analysis and dynamic analysis. While static analysis allows to track all execution paths of a given malware binary it requires a lot of effort and is time consuming. Dynamic malware analysis evades possible anti-debugging measures of the malware and allows to automate the analysis process. However, it may skip certain execution paths of the malware. While we are well-aware of the fact that dynamic analysis can not satisfy the needs in every case (e.g., a novel, sophisticated malware sample may still require manual analysis), we argue that for automated analysis of malware dynamic analysis provides most efficiency. In the end dynamic malware analysis is thus the only viable approach for automated malware analysis at a large scale.

In order to evade anti-debugging measures of malware we found, that VMI is currently the most promising approach.

A variety of tools for dynamic malware analysis have been presented over the past years. Depending on the respective design goals they differ in the applied techniques, implementations, analysis targets and features. Beside Ether, Nitro is the only publicly available tool for hardware-supported VMI at the time of writing.

[11] http://www.symantec.com/connect/es/articles/windows-anti-debug-reference Retrieved 2012-04-05

Chapter 5

A Holistic Approach for Integrated Malware Collection and Analysis

5.1 Introduction

As outlined in chapter 2 todays most disruptive cyber threats can be attributed to (botnet-connected) malware resulting in a fundamental need to track their rapid evolution. Especially timely intelligence on emerging, novel threats is essential for successful IT early warning and malware defense. This requires both, collection and examination, of current real-world attack data, comprising a meticulous analysis of the malware samples. To follow the whole execution life-cycle of the malware, ideally it needs unhindered access to all requested resources during its runtime. While this could be easily achieved by allowing full interaction between the malware and the Internet, this setting is not applicable within most setups and dedicated collection and analysis frameworks are needed. Although this is a mature research topic and has resulted in various approaches, the corresponding frameworks suffer from several shortcomings. In this chapter we present a novel approach for integrated honeypot based malware collection and analysis addressing the limitations of existing approaches. It is currently implemented within the infrastructure of the Fraunhofer AISEC Malware Analysis Laboratory [BEH+10, Fuc11b]. Parts of it have been designed and implemented in conjunction with the Department Secure Information Systems at the University of Applied Sciences Upper Austria in Hagenberg[1] [Fuc11a]. While still under development first preliminary results indicate the feasibility of our approach.

5.2 Problem Statement

Dedicated environments, intended to automate large-scale malware collection and analysis, can satisfy the requirements for tracking the rapid evolution of malware, which result from the vast amount of malware and its sophistication. As outlined in chapter 3 different

[1] www.fh-ooe.at/sec

honeypots and honeynet architectures offer a various degree of interaction with the corresponding network services. In chapter 4 we showed that dynamic malware analysis is currently the only viable approach to automatically analyze a large number of malware samples and that for dynamic analysis several techniques can be utilized. Hence diverse malware collection and analysis environments, such as the Internet-Malware-Analysis System (InMAS) [HEF+09] or AMSEL (Automatisch Malware Sammeln und Erkennen Lernen) [ABFM10], have been introduced differing in various aspects, depending on the particular design goals and application. However, one major issue that remains to all approaches is the need to provide malware with basic network services despite it is executed within an isolated environment. Otherwise the malware may behave different or refuse to continue execution at all thus impeding the achievement of high quality analysis results. This is commonly achieved by utilizing *sinkholing* techniques, that is malicious traffic is transparently redirected to a service within the administrative domain of the analysis environment. Sinkholing techniques are commonly used to study botnets, as shown e.g., in [SGCC+09].

However, as already indicated in the introduction, existing (publicly known) approaches suffer from several shortcomings:

1. **Separation of collection and analysis**
   Commonly malware is captured from the Internet with the use of (mostly low interaction) honeypots. Afterwards the collected malware samples are pushed to a dedicated analysis environment which is (at least partially) isolated from the Internet. Within this analysis environment a given malware sample is then executed and its behavior (i.e., activities such as file-system changes or outbound requests) is observed e.g., using system call tracing. This implies, that the context of the exploited system is lost before actually analyzing the malware. The context refers hereby to the actual process spawned or affected by the malware and thus includes specific information such as memory states and file handlers. Furthermore outbound connections do not persist during the entire collection and analysis process, due to separation of collection and analysis. While this separation is not necessarily a limitation (i.e., not mandatory to gain qualitative analysis results), we argue that its removal enables the acquisition of further information about the malware and its infection vectors that would have been lost otherwise.

2. **Limitations of service emulation**
   During the analysis phase malware will attempt to initiate outbound connections. If these connections can not be established, and thus the malware can not be provided with all requested resources during runtime, tracking further activities fails, i.e.,

   - the next stage within the malware life-cycle can not be reached (e.g., the actual malware binary is not downloaded).
   - a restriction of resource may lead to different malware behavior or even a refusal of execution thus impeding the tracking of the entire life-cycle.

   Within an isolated environment with limited or no Internet access such connection attempts fail, if they are not handled by an appropriate sinkholing approach which spawns an emulated service. However, a specific sinkholing service can only be

spawned once it is known (in advance) that (i) the currently analyzed malware sample will connect to it and (ii) what service needs to be emulated and thereby how the expected response looks like. In addition current approaches lack a possibility to reliably determine whether a given request (i.e., accessing a remote host) is malicious or benign and - based on that - needs to be redirected to an emulated service or not. Thus reactions to malware initiated attempts remain static during runtime. That is they present a set of pre-defined services to the malware, which provide commonly used protocols (such as HTTP, FTP, IRC and DNS) and hence are likely to be queried. Any other than the expected, and thus pre-defined, usage of these services (such as modified standard protocols, unknown C&C protocols and encrypted traffic can not be satisfactorily handled with this approach. Even slight deviations from the expected usage require additional actions (e.g., downloading a specific binary from a randomly generated domain) and will therefore likely result in a connection timeout.

3. **Operational risk of HI honeypots**
   As mentioned in chapter 3 HI honeypots allow the collection of highly valuable data but pose - beside complexity and maintenance issues - a high operational risk which is often inadequately addressed. That is, once compromised, a HI honeypot can be abused in order to launch attacks against other systems in the local network or the Internet since an attacker or malware may gain full control over it. While there are several ways to mitigate this risk, such as running the HI honeypot within a virtual environment, the remaining risk is therefore still higher than the risk posed by the operation of a LI honeypot. Beside ethical aspects this may raise legal and liability issues which are specifically important to organizations operating a HI honeypot. In addition, organizations and commercial enterprises may face reputation loss once third party systems got compromised by a machine which can be assigned to their networks. As a result the consideration of these issues leads to strict constraints that have to met for honeypot operation. Although organizations may gain valuable information about attacks and malware targeting their networks with the use of HI honeypots their application is often hindered within industrial environments due to the mentioned legal and liability issues and the resulting constraints that have to be met.

5.3 Overall Approach

5.3.1 Goals

Our overall goal is to capture and dynamically analyze malware at a large-scale basis. Thereby we want to be able to cover the entire execution life-cycle of a given malware sample in a preferably automated way within a controlled environment while being capable of handling novel malware as well. That is, we want to be able to cover also malware that uses a (yet) unknown or encrypted protocol (e.g., as C&C channel). In order to minimize harm to third parties the malware should have preferably no Internet access during the whole procedure but is tricked into believing to be on a real victim host with full Internet access. In the end we can then anticipate trends of current and emerging malware.

5.3.2 Basic Concept

Contrary to purely network-based approaches for service sinkholing our basic idea is to gain information about the malware logics directly during execution on the analysis system in order to identify the used services and protocols that need to be provided for the next step within the execution life-cycle of the malware. Therefore our presented approach operates at a binary level, directly interacting with the malware's host system. It is based on a HI honeypot and a virtual machine introspection (VMI) framework. Integrated in our hybrid honeyfarm this system enables us to carry out malware collection and analysis on one platform preserving the context (i.e., memory, register states, etc.) of the exploited system during analysis. We enrich this with a transparent pause/resume feature which we instrument to determine and - where appropriate - interrupt the next actions initiated by the malware. Furthermore this allows us to extract and manipulate instructions and data, such as protocol information, within the memory of the victim machine during runtime. This can be specifically valuable for extracting cryptographic material used by the malware in order to intercept or manipulate encrypted communication. Malware requests can be checked with regard to several conditions (e.g., download request or C&C traffic) and can be handled accordingly by a corresponding service-handler or a sinkholing service in order to keep full control of all interactions between the malware and the outside world. In order to handle unknown traffic as well we add self-learning capabilities to our system insofar as we automatically derive service emulation scripts from the observed traffic. Automating the whole process of integrated collection and analysis as much as possible aims at being able to handle a large amount of malware thus making our system scalable.

5.3.3 Added Value

The presented approach offers a couple of benefits.

First the context of the executed malware persists during malware collection and the subsequent analysis. Uniting collection and analysis on the other hand is alike the approach of a HI interaction honeypot and thus comes closer to real-world scenarios where a victim host is both, the system that is infected and affected by the malware (refer to chapter 2.2.3).

In addition we achieve increased transparency during malware analysis due to the use of VMI. We consider this as a benefit, since we argue that VMI based analysis is more likely to remain undetected by the malware compared to other techniques, which require trusted, supportive components within execution context of the malware. Hence we have a higher probability to follow the entire malware execution life-cycle (as described in 2.3.4). As the utilized VMI framework is based on KVM, we can furthermore use a variety of guest OS as honeypots.

Since our approach depends on no analysis components within the VM we believe it to be more secure than existing approaches while also expecting a better performance.

We are able to extract and inject data and instructions from or into the memory of the

VM during runtime. An envisioned use case for this functionality includes tapping and manipulation of encrypted C&C traffic (by acting as man-in-the-middle).

Furthermore the approach allows us to control any interaction between the malware and third party systems which is necessary to fulfill possible legal and liability constraints. Specifically we can analyze and filter outgoing requests that are unknown or respectively considered as possibly malicious. That is, we provide Internet access to benign (i.e., passive) requests such as downloading binaries on the one hand and redirect malicious communication patterns (e.g., C&C traffic) to a sinkholing service. Since the approach applied directly on the instruction level we are aware of the next actions initiated by the malware and can therefore provide the according services and even serve novel communication patterns. Hence the presented approach evades known limitations of existing approaches for service emulation that are purely network based.

After all, roughly speaking, the approach enables the operation of a HI honeypot at the security level of a LI honeypot.

5.3.4 Components

Our approach therefore utilizes the following components:

- For **Malware Collection** we use the ARGOS HI honeypot to which we made several modifications such that it enables us to carry out the actions describes above (such as pausing and resuming the VM).
- **Malware analysis** is performed using Nitro, a KVM-based framework for virtual machine introspection. We utilize it for transparent tracing of system calls and extend it with features for examining them. In particular we want to determine whether a given action (i.e., an outbound request) initiated by the analyzed malware requires Internet access.
- **Service provisioning** comes into play once the analyzed malware attempts to connect to any resource on the Internet. If this attempt is considered to be malicious we provide an appropriate service by taking advantage of honeyd which then spawns it, given that it is a known protocol. Otherwise our system observes the traffic while automatically learning the involved protocols. We achieve this by instrumenting ScriptGen technology which will be introduced in chapter 6.

We provide detailed information on each component and their interaction between each other along with an in-depth description of the malware collection and analysis process in the following sections.

5.4 General Design

The key component is the modified ARGOS honeypot in conjunction with the extended VMI framework. It integrates the honeypot based malware collection with dynamic malware analysis. The entire integrated malware collection and analysis process is structured as follows:

- In the first part malware is captured from the Internet using various honeypots and all received binaries are stored in a central repository. Thereby only unknown malware is forwarded and thus analyzed by the ARGOS honeypot. We consider this first phase to cover the initial infection and exploitation part with respect to our malware life-cycle we presented in 2.3.4. Hence we expect to receive a shellcode or a dropper during this phase rather than the actual malware binary.

- Within the second part the received binary is executed and analyzed. As stated before, when entering this part for the first time we expect this to be a shellcode or dropper performing the appropriate actions such as downloading further components. As a result of the syscall examination we check every action initiated by the malware to the extend it is interesting for the further analysis process. While what is 'interesting' is generally adjustable based on a set of rules, we primarily focus on the question whether any interaction is needed (i.e., does the malware try to initiate any outbound connection). If this is the case we pause the execution of the VM and check whether the connection attempt may be malicious. The fact that the VM has been paused is thereby transparent (i.e., not detectable) to the malware. The only viable approach for the malware to detect that is running within the analysis environment would be to check an external time source such as a time server on the Internet. However, this could be detected within the syscall examination and an appropriate response would be provided.

- In the third part we handle the corresponding request initiated by the malware according to its nature. That is, if we consider the request to be benign it is handed over to a dedicated service handler having Internet access which resides separated from the analysis environment. This service handler visits the corresponding URL, downloads the requested binary and inserts it into the memory of the VM. Afterwards the execution is transparently resumed. If the request is considered to be malicious it is forwarded to the internal service handler, which manages the connection to the corresponding service which in turn generates an appropriate response. This response is inserted into the VM and the execution is resumed again. We provide some more details on the decision whether a given request is considered to be malicious or benign below.

These steps are repeated thereby iterating throughout the entire execution life-cycle of the malware. The corresponding components are implemented as bots, i.e., acting autonomously in order to fulfill their tasks using the XMPP protocol. A high-level sketch of the general design is depicted in figure 5.1.

5.4.1 Setup

For the central component within our approach a honeypot for malware collection is required. We select a suitable honeypot based on our primary goal as a criteria, i.e., the collection of malware in a preferably large quantity and diversity. While well-known and popular LI honeypots like nepenthes have proven to be an efficient means for malware collection the knowledge-based approach of nepenthes has also some drawbacks regarding the diversity of malware it can collect as we outlined in chapter 3.4. Additionally these

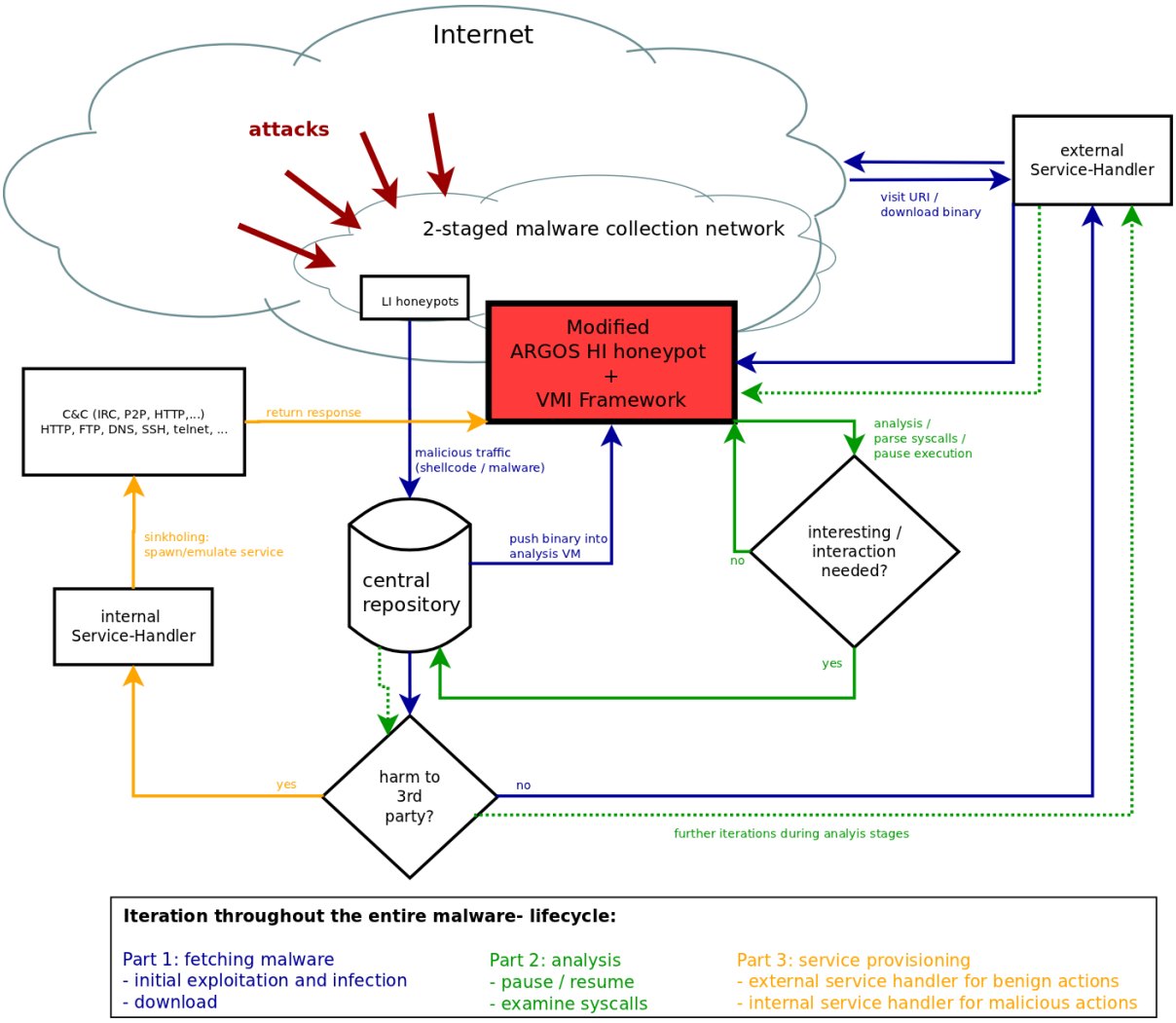

Figure 5.1: General Design of the Presented Approach

limitations do not only influence the diversity but also the quantity of the collected malware. This is substantiated in a paper by Zhuge et al. [ZHH+07]. The authors found by comparing the malware collection capabilities of nepenthes with those of their proposed HI honeypot a considerable difference: While the HI honeypot was facing a lower hit count (376,456 hits) compared to nepenthes (427,829 hits) the HI honeypot was able to collect considerably more binaries than nepenthes. In particular the HI honeypot collected 82,137 binaries out of 171 malware families with 1,011 variants. Nepenthes collected thereby 17,722 binaries out of 64 malware families with 467 variants. Hence we will use a HI honeypot for our setup. In addition the honeypot is required to run within a virtual environment (since we want to apply VMI) and as it will get served by the existing honeyfarm it should be server based. Based on our classification and the subsequent taxonomy (refer to 3.4.5) we chose ARGOS as the honeypot to use within our setup. It is capable of detecting both, known and unknown (0-day) attacks while it is independent of special

collection mechanisms. Compared to other suitable implementations such as HoneyBow ARGOS is based on QEMU, which perfectly matches with our KVM based VMI framework Nitro. The setup is depicted in figure 5.2.
The following modifications of these components have been implemented to meet the requirements of our setup[2]:

- A possibility to transparently pause and resume the QEMU VM of ARGOS:
  Since the ARGOS approach is more time-consuming than traditional approaches and thus detectable by the malware during analysis (e.g., due to anomalies such as connection timeouts or latency-issues) a pause and resume function has been implemented. While QEMU relies on virtual CPU ticks for time operations a freeze of these ticks (i.e., by repeatedly responding with the same tick counter) is not feasible, because it freezes the context switch and the scheduling as well. As a result the entire VM, including the QEMU backend, is stopped. Thus the pause and resume feature has been implemented as an offline approach i.e., the victim VM's RTC is detached off the host's clock.

- Trigger for starting the analysis:
  Once ARGOS detects an attack (i.e., the taint-map reports that tainted memory is being executed), it dumps the corresponding memory pages and we activate the analysis functionality, that is provided by the VMI framework. Simple interpretation and filtering of system calls and their parameters is conducted directly within hypervisor space, while more complex analysis is performed via the virtual machine monitor (VMM) in the host environment.

- Interface for extraction of protocol information:
  The purpose of this interface is to extract information on the protocol and the corresponding URI (pointing to the binary that should be downloaded) from the injected shellcode, that resides within the memory of the VM. Therefore a basic shellcode parser and decoder has been implemented, which detects an incident utilizing the ARGOS hook. More sophisticated approaches for shellcode detection end decoding, such as used by other honeypot implementations, could be integrated as well.

5.4.2 Part 1: Fetching malware

As stated earlier, we utilize the ARGOS honeypot for the central component of our approach. However, as a HI honeypot, ARGOS requires much effort in deployment and maintenance. Furthermore one of the main drawbacks of ARGOS is its poor performance which is amongst others related to the overhead caused by the taint mapping technique. To overcome this limitation we deploy a two stage malware collection network (i.e., a hybrid honeypot system, refer to 3.4.6) as outlined in figure 5.3. Therefore we take advantage of our existing honeyfarm infrastructure ([BEH+10, Fuc11b]). This malware collection network consists of various honeypots and honeypot-types. As outlined in section 3.4 especially client honeypots play an increasingly important role since the approach of this

[2] http://www.fh-ooe.at/campus-hagenberg/studiengaenge/bachelor-studien/sichere-informationssysteme/projekte-praktika/studienprojekte/projekt/21857/ Retrieved 2011-12-30

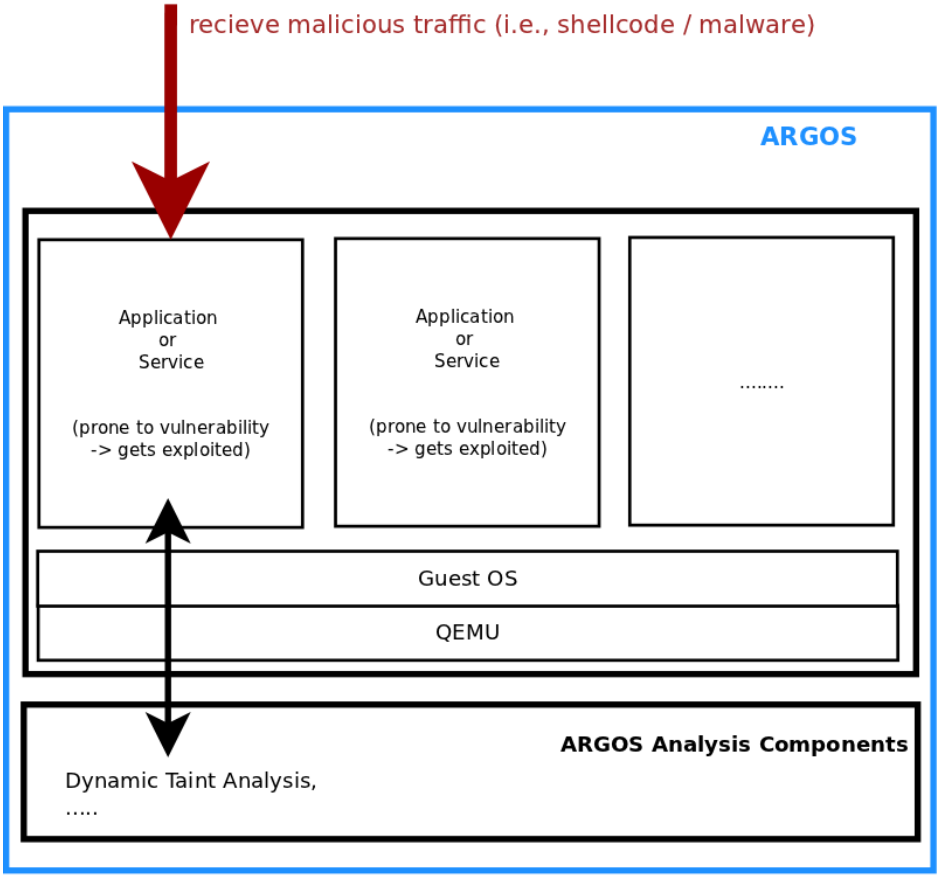

Figure 5.2: Setup of Modified ARGOS Honeypot

honeypot type covers the detection of state of the art attack vectors thus enabling client honeypots to capture current malware that may have not been collected using server honeypots. The honeyfarm utilizes a large-scale network telescope (in particular a /16 darknet) serving various different LI honeypots. We use this infrastructure in order to filter noise and known malware (in particular everything that can be handled by the LI honeypots or their vulnerability handling modules respectively). The so collected binaries (which we consider to be mostly shellcode containing URLs and droppers) are stored in the central repository. Based on the file-hash known files are distinguished from novel ones. Only novel attempts are forwarded to the ARGOS HI honeypot which then does the further processing. By doing so we minimize the load on ARGOS and thus justify its operation.

5.4.3 Part 2: Malware analysis

Dynamic malware analysis performed using commodity virtualization solutions, such as VMware, can be easily recognized and thus evaded by the malware as shown by e.g., [Fer06]. Hence VMI is currently the most promising approach to evade anti-debugging measures of malware, as stated e.g., by [ESKK08]. Thus we rely on the technique of vir-

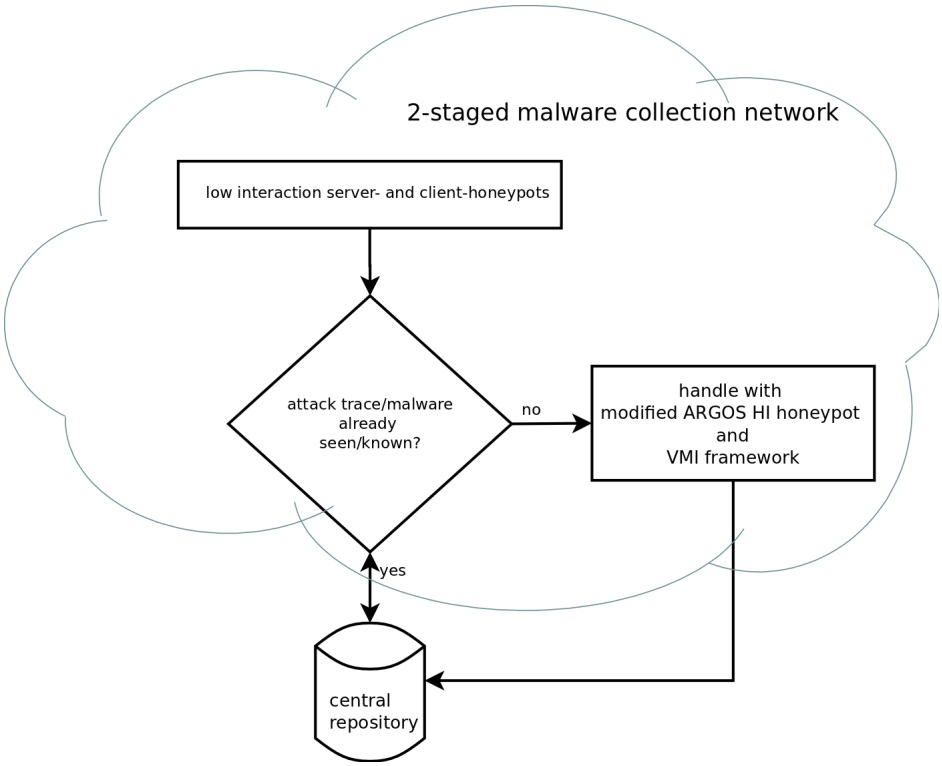

Figure 5.3: Scheme of the Two-Stage Malware Collection Network

tual machine introspection in order to achieve preferably transparent dynamic malware analysis via system call tracing. Therefore we chose Nitro as publicly available tool for VMI since it has several advantages regarding performance and functionality compared to Ether as stated by Pfoh et al. in [PSE11]. The biggest difference is that Ether builds upon the Xen hypervisor, while Nitro is based on KVM. Xen has been removed from the Linux default kernel in favor of KVM. In addition Nitro is guest OS portable. That is, it supports any guest OS that is compatible with the x86 or the IA64 architecture while Ether is mainly limited to Windows XP SP2. Furthermore the development on Ether seems to have stopped. By using Nitro we utilize KVM as interface to the Linux kernel and have full virtualization capability thanks to the hardware support provided by the Intel VT technology. Hence we expect a reasonable performance for the analysis process. We use QEMU as virtualization environment, which is in addition well combinable with KVM.

The analysis flow is depicted in figure 5.4. Thereby we receive in a first step the malicious binary. Again, in the first iteration we expect this to be a shellcode or a dropper within the initial infection path rather than the actual malware binary. The shellcode is then decoded and most likely contains a URL where to download the dropper or the actual binary. In the second iteration this binary is then executed after it has been downloaded and the VM has been resumed. The resulting system call traces produced by Nitro contain the actual system call name as well as its parameters. Thereby we look for interesting syscalls, i.e., those requiring interaction since they initiate an outbound connection (e.g.,

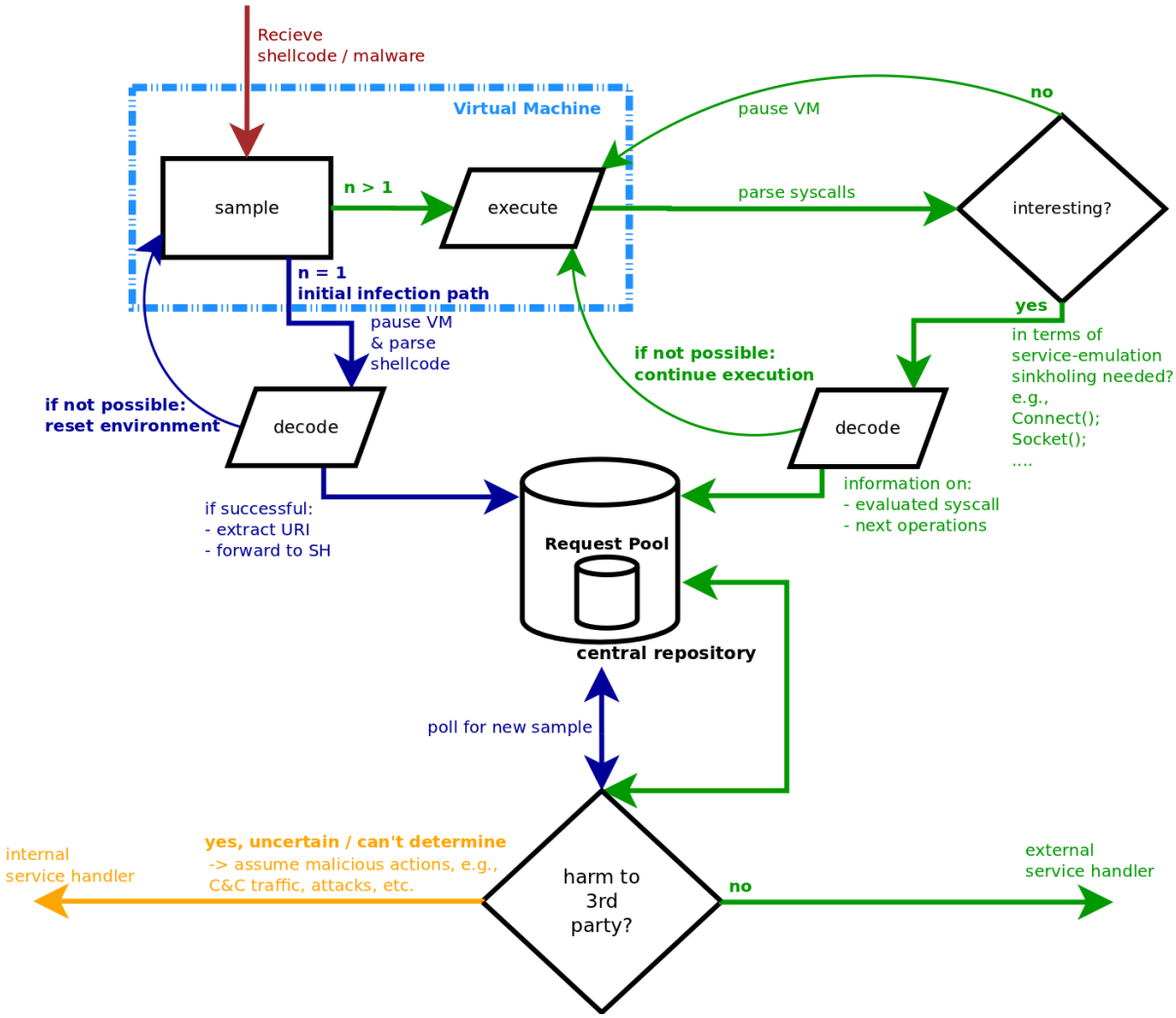

Figure 5.4: Malware Analysis Using Virtual Machine Introspection

'NtConnectPort'). There are basically two approaches for inspection of the recorded system calls. Either a function block analysis is performed (i.e., the system call itself and its associated parameters are evaluated) or the process is tracked covering the calling thread and the process. A limitation that remains, is the examination of swapped pages. Since they are not accessible via the VMM an examination would require interpretation of the internals of the given OS. An in-depth description on how protocol identification and information extraction at a binary level with the support of virtualization technology is achieved, can be found in [Fuc11a].

5.4.4 Part 3: Service Provisioning

The communication infrastructure for the service provisioning part is depicted in figure 5.5. It is based on the standardized XMPP protocol and is split into the following components.

New malware-driven outbound requests and connection attempts are polled from the central repository. Each request is then evaluated, whether it may cause any harm to third parties. The main goal is to prevent any affection of uninvolved systems, specifically to avoid any harm to them. This check is hereby fairly flexible. It may include simple black- or white-listing, the filtering of specific parameters that are suspicious (e.g., indicating XSS) or to pass the traffic through well-established IDS means such as Snort inline or a web application firewall.

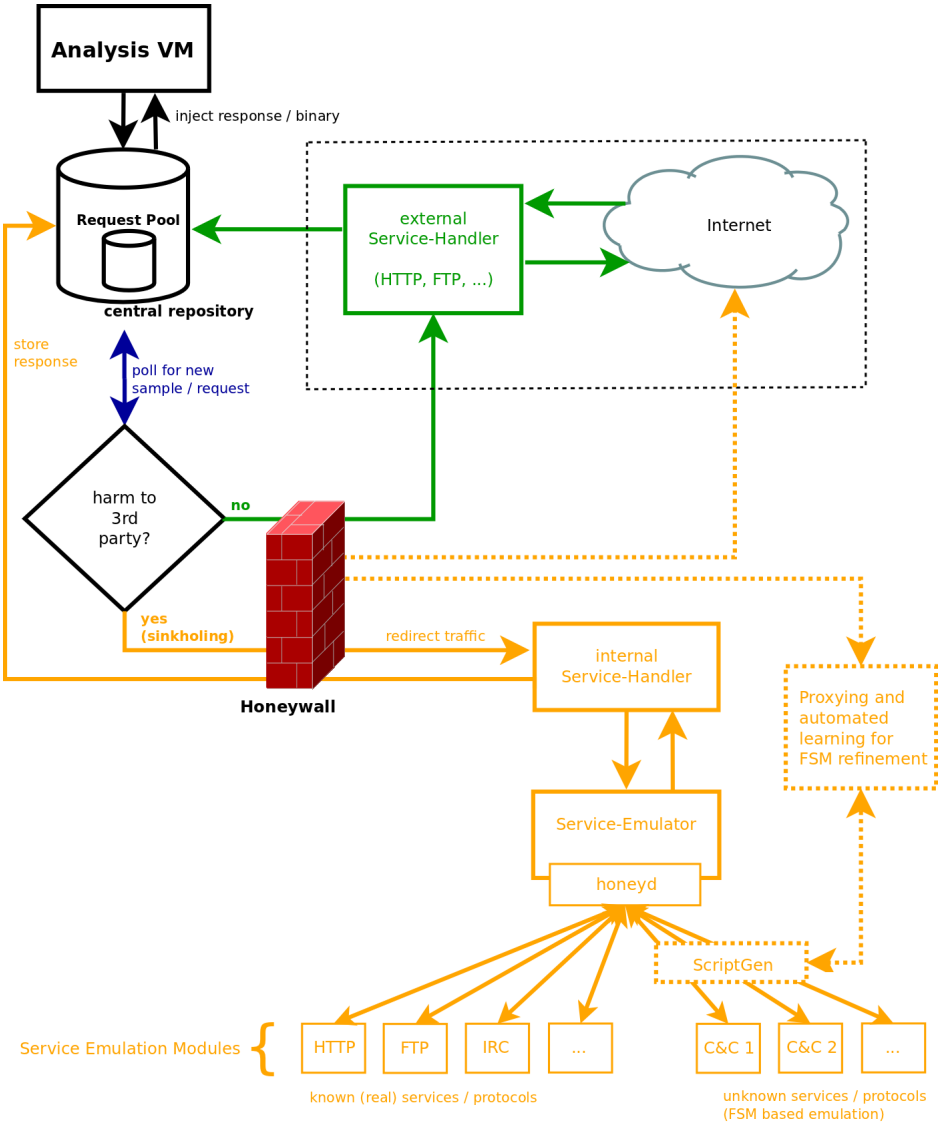

Figure 5.5: Service Provisioning Concept

If a given request turns out to be benign (i.e., it does not cause any harm), it is handed over to the *external service handler*. The assumption is, that any purely passive request, such

as a download, can be considered as harmless. This service handler is implemented as a simple bot, whose sole task consists in downloading the requested resource and storing the result in the central repository. Therefore it identifies the needed protocol based on the extracted URI (e.g., http://) and visits the corresponding resource on the web. The result of this request is in any case stored in the repository and presented to the malware in the next step (i.e., injected into the memory of the VM). Hence it may consist of a binary (in case of success) as well as a corresponding error (e.g., HTTP 404). Since the external service handler has full Internet access it resides in a dedicated network segment which is separated from the analysis environment.

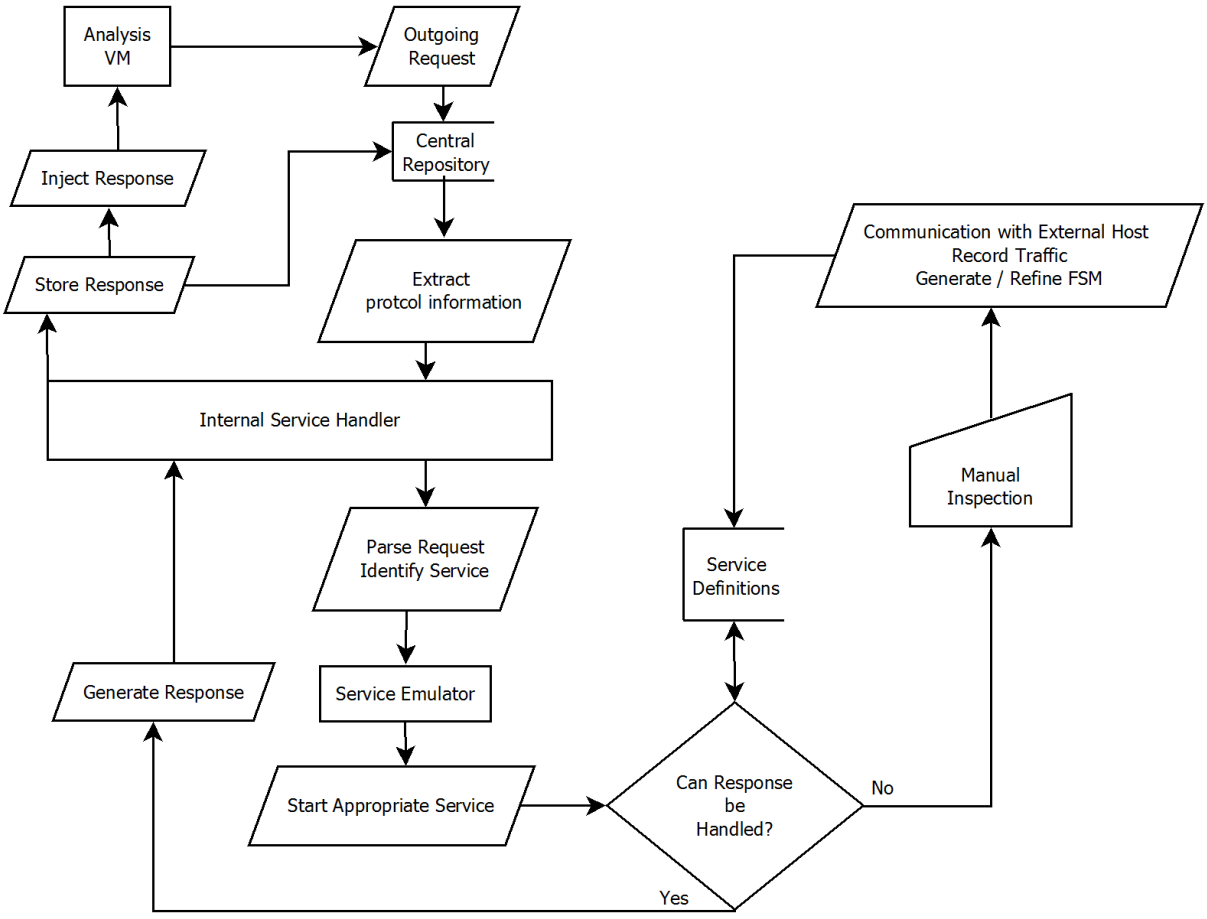

Figure 5.6: Data Flow of the Service Emulation Procedure

If a given request is likely to be malicious or unknown (that is, at least it can not be determined that it is harmless), it is sinkholed, i.e., redirected to the *internal service handler*. In particular we consider any traffic as malicious, which contains attack patterns, such as attempts to send Email or to launch a DoS attack. In addition any traffic, that likely belongs to C&C communication is treated as malicious as well. This covers both, known C&C protocols as well as unknown (i.e., encrypted) ones. This service handler is in turn implemented as bot detecting the used protocol based on the extracted URI, whereas it

manages and redirects the attempts to the service emulator.

The *service emulator* then identifies the requested service utilizing a TCP wrapper. Its main goal is to provide an appropriate payload for the request generated by the service handler. Since this can be extremely heterogeneous a slight and fast honeypot application is required which is able to emulate or spawn arbitrary services. Again, since it will be served with requests from the service handler, a server based approach is needed. In addition we aim to maximize the performance for scalability reasons. With respect to our presented honeypot taxonomy in chapter 3.4.5 we use honeyd for this task, because it is very flexible and extensible. Specifically it is also capable of running real services as subsystems rather than only emulating them. This is specifically important since in various cases a rudimentary emulation of the requested service may not be sufficient. For example, if the malware attempts to contact a given C&C server (e.g., IRC or HTTP) in order to retrieve instructions we can start the appropriate (real) service with a totally authentic environment (such as the corresponding channel in case of IRC). Obvious prerequisites are hereby that the C&C protocol is known and a DNS server pointing to our spawned service, which has to be launched as well. Furthermore honeyd has proven to be very fast, both by us (serving around 16,000 IP addresses without any difficulty) and others. For example the authors of NoaH reported in D3.2[3] to run honeyd on a /8 network which they flooded with connections from random hosts in order to measure stability and throughput. They found that honeyd did not have any malfunctions and established all connections while causing a low CPU and memory overhead. Handling unknown protocols is a much more challenging task, whereas we consider this to be mostly novel C&C protocols. Therefore we instrument *ScriptGen*, which was presented by Leita et al. [LMD05]. ScriptGen generates finite state machines (FSM) from the observed traffic. Thereby each generated FSM represents the behaviour of a given protocol on an abstract level. Hence it can be used to derive service emulation scripts for honeyd based the generated FSMs. These FSMs are thereby automatically refined while we point out that no prior knowledge about a given protocol or its semantics is needed by ScriptGen. By integrating ScriptGen into our approach we aim at adding 'self-learning capabilities' to the service provisioning part. Obviously this requires at least the one-time observation of a given communication (i.e., between the HI honeypot and the external system) in order to capture the required traffic dump. Hence we need a back-channel for learning about a novel protocol. That is, at this point a a manual intervention is necessary in order to inspect the current request and to examine, whether communication with the outside world is granted in this case. Once this has been established for one time and the appropriate FSM has been generated, we are able to handle the new protocol as well. While this is a clear limitation we consider it to be a reasonable trade-off. The data flow of this procedure is depicted in figure 5.6. More details on instrumenting the ScriptGen technology for use within our approach are provided within chapter 6.

[3] http://www.fp6-noah.org/publications/deliverables/D3.2.pdf Retrieved 2012-01-06

5.5 Summary

In this chapter we presented a holistic approach for integrated honeypot based malware collection and analysis which overcomes the shortcomings of existing approaches. Specifically it addresses the separation of collection and analysis, the limitations of service emulation and the operational risk of HI honeypots. The overall goal is to capture and dynamically analyze malware at a large-scale basis while covering the entire execution life-cycle of a given malware. This should happen in a preferably automated fashion within a controlled environment while being capable of handling novel malware as well. The basic idea is therefore to gain information about the malware logic directly on the analysis system in order to identify the used services and protocols which have to be provided for the next step within the execution life-cycle of the malware. The proposed approach offers several benefits, such as:

- The context of the executed malware persists during malware collection and the subsequent analysis.
- An almost transparent malware analysis due to the use of VMI.
- Ability to extract and inject data and instructions from or into the memory of the VM during runtime based on the transparent pause and resume feature.
- Control of any interaction between the malware and third party systems.
- Since the approach starts directly on the host level, we are aware of the next actions initiated by the malware and can therefore provide the according services and even serve novel communication patterns.
- The risk resulting from the operation of a HI honeypot is drastically reduced.

The approach consists of three parts and utilizes as its key components the ARGOS HI honeypot, Nitro as a framework for virtual machine introspection as well as honeyd and ScriptGen for service provisioning. Several modifications to these tools have been made and the first parts of the approach have already been implemented. While the work on this approach is an ongoing research activity (thus still under development) the first parts have already been successfully tested. We hereby referred to the corresponding related work. In the next chapter we present a proof of concept for the service emulator as a component for the service provisioning part.

Chapter 6

Proof of Concept Implementation

6.1 Introduction

In this chapter we present our proof of concept implementation which leverages ScriptGen output for service emulation. The aim of the conducted experiment is to evaluate, whether and to what extend the ScriptGen approach can be applied to a specific use case of service emulation, namely automated learning of C&C traffic. In section 5.4.4 we designed the service provisioning concept as one part of our approach. The therein described components raise several technical issues. However, since miscellaneous research has been conducted in this area, most of these issues can be addressed based on existing work.

For sinkholing several advanced approaches exist on which we can base on. For example *Truman*[1] *(The Reusable Unknown Malware Analysis Net)* and *INetSim (Internet Services Simulation Suite)* [2] simulate various services that malware is expected to frequently interact with. To this end common protocols, such as HTTP(s), SMTP(s), POP3(s), DNS, FTP(s), TFTP, IRC, NTP, Time and Echo are supported. In addition INetSim provides dummy TCP/UDP services, which handle connections at unknown or arbitrary ports. Hence these approaches can interact with a given malware sample to a certain level. *Trumanbox* [Gor07] enhances the state of affairs by transparently redirecting connection attempts to generic emulated services. To this end it uses different information gathering techniques and implements four different modes of operation, which allow the application of different policies for outgoing traffic. Thus, Trumanbox can provide different qualities of emulation and addresses issues in protocol identification, transparent redirection, payload modification and connection proxying.

In addition, several work has been conducted in the context of malware analysis to address the issues of detecting, observing and intercepting malicious (C&C-) traffic [GZL08, RSL+10, PGPL11, CG08] resulting in publicly available tools.

Hence we concentrate on the issue of handling unknown traffic patterns, such as C&C protocols within our service provisioning element. Therefore we instrument ScriptGen, which is introduced in the following section.

[1] http://www.secureworks.com/research/tools/truman/ Retrieved 2012-03-26
[2] http://www.inetsim.org Retrieved 2012-03-26

6.2 ScriptGen

6.2.1 Basic Idea

ScriptGen [LMD05] is a tool designed to learn protocol semantics from traffic, which has been observed during an interaction with a real server. The basic idea is to record several communication sequences and based on this generalize the monitored protocol information. As a result ScriptGen derives a finite state machine (FSM), which describes the observed protocol in an abstract way. The assumption is, that the so generated FSM is detailed enough to sufficiently emulate the protocol by responding to clients. ScriptGen is protocol agnostic, i.e., it does not make any assumptions about the currently observed protocol and does not require any a priori knowledge of the underlying protocol structure. Thus it can be applied to almost any protocol. The underlying assumption (as stated by the authors in [LMD05]) is, that the request, which needs to be answered, is generated by deterministic automata (i.e., exploits). Since exploits represent a limited subset of the total possible input range and commonly perform a limited number of execution paths, the task is drastically simplified. As depicted in figure 6.1 ScriptGen consists of the following four functional modules.

6.2.2 Modules

Message Sequence Factory
This module extracts messages, which have been exchanged between a client and a server, from a given tcpdump file. A notion of sequence can be given for different protocols (e.g. UDP, or IP-only based protocols) while this module reconstructs TCP streams as well, thereby correctly handling retransmissions and reordering. A *message* is defined as a piece of interaction between the client and the server (i.e., a sequenced set of bytes going in the same direction). A TCP session consists of a list of messages. In order to rebuild TCP sequences the tcpdump file is parsed, whereas ScriptGen takes into account, that a given client may not comply with the classical TCP state machine (e.g., to implement IDS evasion techniques). To this end ScriptGen makes the following assumptions for rebuilding the TCP flow:

- A packet is only considered to be relevant, when it carries a payload. That is, every pure ACK packet and retransmissions are ignored.
- A TCP session starts with the first SYN- and ends with the first FIN/RST-packet.
- The TCP sequence number is as index of an array, where the payload is stored. This allows ScriptGen to handle out of order packets and retransmissions.

State Machine Builder
The messages, which have been extracted from the previous module, are then used as building blocks to build a (complex) FSM. This FSM is composed of edges and *states*. For a given state, the outgoing *edges* are the possible transitions towards the next state. Each edge has an *edge label* i.e., a message representing the client request that triggers the given transition. In turn, each state has a *state label*, i.e., a message representing the answer of

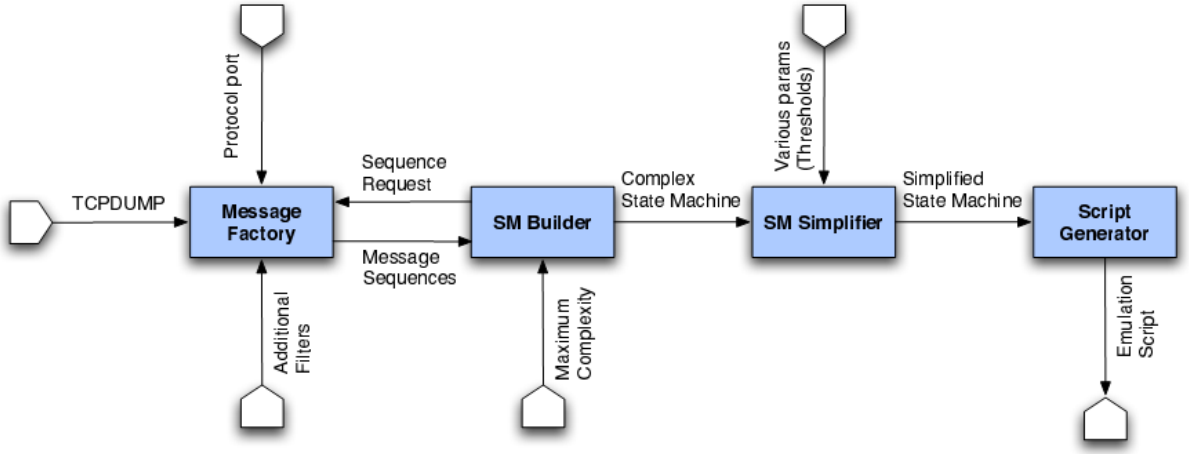

Figure 6.1: General Structure of ScriptGen (from [LMD05])

the server, which is sent back to the client, once entering the state. In addition every edge label has a weight, which represents the frequency with which the communication samples have traversed that specific transition. A given state can have more than one state label (i.e., the server can provide different answers) while the most frequent one is the default. A simple example of such a FSM is depicted in figure 6.2. It consists of one initial state, which is labeled with the server message S0 (e.g., a welcome banner). This initial state has three outgoing edges representing the different client messages (C1, C2, C3). Each of the edges leads to a future state, which in turn has one or more state labels.

At this point the FSM can be very large, redundant and highly inefficient. Hence it is necessary to define thresholds, which limit the number of outgoing edges per state in order to control the complexity growth. In particular, there are two thresholds: (i) The maximum allowed number of outgoing edges from one state and (ii) the maximum number of states.

State Machine Simplifier

This module is the core of ScriptGen. It is responsible for analyzing the previously generated raw FSM and introducing some sort of semantics. Since the raw FSM has been created without respect to the protocol semantics, it is specific to the corresponding tcpdump file, from which it has been generated. Thus it lacks generality i.e., it is not able to handle anything that has not been seen before. The task of this module is to simplify and generalize this raw FSM. This is achieved using two distinct algorithms:

- *Macroclustering*:
  This phase includes an inspection of the initially created raw FSM while collapsing together those states, which are considered to be semantically similar. Therefore ScriptGen takes advantage of the findings from the Protocol Informatics Project (PI), which provides a fast algorithm for performing multiple alignment on a set of protocol samples, as described in [LMD05]. If this algorithm is applied to the outgoing edges, it is capable of identifying the major message classes and aligning messages of each class using heuristics. ScriptGen utilizes the output of PI as a building block to align the sequences and create a first clustering proposal.

- *Microclustering*:
  The next phase consists of performing the newly introduced algorithm called *Region Analysis*. This algorithm uses the PI output to generate so called *microclusters*. Therefore the aligned sequences, which have been created by PI, are inspected on a per byte basis. For each aligned byte ScriptGen computes
  1. the most frequent data type (e.g., binary or ASCII),
  2. the most frequent value,
  3. the mutation rate (i.e., the variability of the values) and
  4. the presence of gaps in that byte.

  Based on this results ScriptGen defines a *region* as a sequence of bytes, which
  1. have the same type,
  2. have similar mutation rates,
  3. contain the same type of data and
  4. have gaps or not.

  Thus, a region is a piece of messages with homogeneous characteristics, therefore carrying the same kind of semantic information. The principle of microclustering is outline in figure 6.3. It shows, that macroclustering can not distinguish between the two HTTP GET requests, since the distance between the two sequences is not significant enough to group them into different clusters. However, by searching for frequent values in each region, new microclusters can be generated reflecting the semantically coherent parts. In order to identify slight, but important deviations, such as a bitmask, microclustering computes another distance, namely the variability of the value assumed by the region for each sequence. That is, microclustering assumes frequent values to contain probably some sort of semantic information.

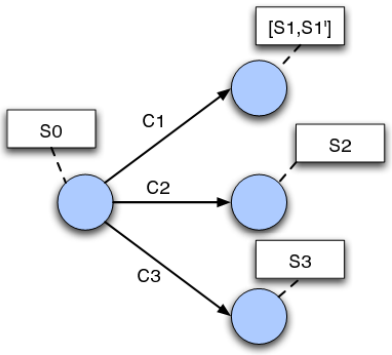

Figure 6.2: Simple FSM Example ([LMD05])

The described two clustering methods help to reduce the number of states and edges of a given FSM, whereas the regrouped states result in abstract states. In addition ScriptGen can handle dependencies for regions, which play a specific role in the communication. Such regions contain changing information, which are sent by a client and are expected to be included in the response of the server (e.g., a Session ID). This is taken into account using so-called *Random Regions*. That is, regions with a very high mutation rate (i.e., almost 100%). Therefore ScriptGen inspects the values of these regions and tries to find the according values in the server response.

Script Generator

After the simplified FSM has been created, it needs to be stored in a way that it can be emulated by a script usable by Honeyd. This is the task of this last module.

6.2.3 Discussion

While the approach of ScriptGen is sound, we also expect some limitations. Specifically, we assume, that the input (i.e., the tcpdump file) will have a very high impact on the resulting emulation quality. In particular, the recorded traffic is required to represent a preferably high diversity i.e., ideally the entire functionality provided by the corresponding protocol. As described in chapter 5 our approach utilizes the Argos HI honeypot for interaction with a malicious host. Thus we expect the thereby recorded traffic, which is used by ScriptGen to generate FSMs, to be representative, since the HI honeypot approach is closely aligned to real world scenarios. This assumption is substantiated by some measurement of ScriptGen within the SGNET infrastructure [LD08]. The authors found, that the system is capable of automatically learning how to respond to incoming attacks: After only 8 hours of operation, ScriptGen could collect enough network traffic to build a first path in the FSMs, which enabled it to handle 10 other attack activities of the same kind showing up on the same day. In addition they monitored the evolution of the FSM size and found, that the acquired knowledge does not explode over time: After a constant growth in the first 10 days of experimentation, the FSM knowledge stabilized to a number of 68 states requiring less than 1 MB. This shows that ScriptGen is able to generalize observed attacks and to automatically learn the emulation component based on the FSM thus demonstrating the viability of the approach. Moreover, this result suggests, that autonomous spreading malware utilizes a very limited number of exploits for propagation, as ScriptGen was able to handle all incoming attempts using the 68 states. The authors conclude that the generated FSM is satisfying to handle this exploitation phase, but it is insufficient to model complex interactions, such as code injection attacks.

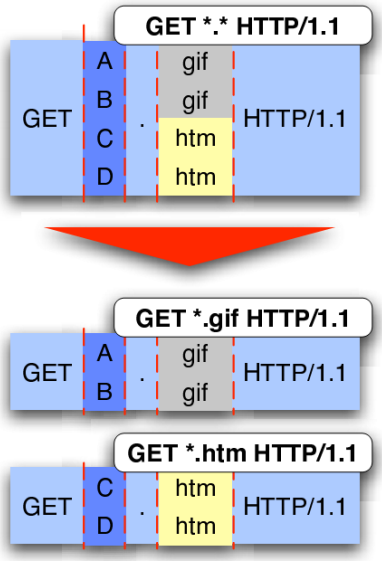

Figure 6.3: Example of Microclustering (from [LMD05])

Additionally ScriptGen has also some implicit limitations, as the authors state in [LMD05]. First, the emulation quality heavily depends on the value of the thresholds, since they influence the size and complexity of the FSM as well as the level of simplification due to the clustering algorithms. Second, ScriptGen is unable to handle causal dependencies between different communication, for example the control- and data-channel of the FTP protocol. This is also true for server responses, which depend on information that is not in scope of the TCP session, such as time of the day. Third, since ScriptGen can only replay payloads previously seen in a tcpdump file, it can not handle encrypted communication. However, due to the expected benefits of our presented approach, we believe, that such information could be incorporated as well, since it enables the extraction of cryptographic material.

Although a script generator is mentioned in the initial paper ([LMD05]), the authors abandoned this design later on in favor of integrating ScriptGen into the SGNET infrastructure ([LD07, LD08]). Hence, there is the requirement to adapt it to our needs in order to be applicable

within our approach. We describe the performed steps for implementation in the following section.

6.3 Implementation and Validation

6.3.1 Setup

In order to ensure defined test conditions we chose to build our own malware, since this provides full control over all test parameters thus assuring reproducibility. To this end we base upon the source code of the Agobot / Phatbot family and compile it using a custom configuration. We chose Agobot, since it is one of the best known bot families and widely used. In addition it provides a variety of functions. For the sake of easiness we use unencrypted IRC as the C&C protocol. We deploy a minimal botnet consisting of only one infected host and one C&C server. In addition we use a third host, which is responsible for the service emulation part. Our resulting test setup consists of three distinct machines:

1. The *victim host* resides on a machine running Microsoft Windows XP SP2. Traffic is captured on this host using WinDump[3] v3.9.5 (the Windows port of tcpdump) based on WinPcap[4] v4.1.2. This host is infected with our malware in order to capture the malware initiated network traffic as a reference for real C&C traffic.

2. The *C&C server* resides on a machine running Microsoft Windows XP SP3 and is updated to the latest patch level. This is necessary, since this host is also used to build our customized version of Agobot. For a successful build of Agobot a full featured Microsoft Visual C++ environment including the latest Visual Studio service pack and the latest platform SDK is required. The main purpose of this host is to act as the C&C server. Therefore we use UnrealIRCd, a well-known IRC daemon widely used by botmasters. There are several customized versions, which are optimized for botnet usage (e.g., designed to serve a vast number of bots). However, for our simple test case the latest standard version (UnrealIRCd 3.2.9[5]) is sufficient.

3. The *service emulator* runs a standard installation of Ubuntu Server 11.10 32bit. Its main task is to process the recorded traffic dump using ScriptGen and to generate a service emulation script out of the resulting FSM, which is then used by Honeyd. To this end this host is equipped with Honeyd and the dependencies of ScriptGen (i.e., python 2.7, python-dev, cython, python-numpy, python-pcapy, python-setuptools and the nwalign package[6]). Finally this machine replaces the original C&C server by running Honeyd with the previously created script.

The source code and all relevant configuration files of the experiment are included on the CD-ROM, which is enclosed to this thesis. The overall test procedure of our experiment consists of the following steps, which are described in the sections below:

[3] http://www.winpcap.org/windump/ Retrieved 2012-04-20
[4] http://www.winpcap.org Retrieved 2012-04-20
[5] http://www.unrealircd.com/ Retrieved 2012-04-20
[6] Obtainable via http://pypi.python.org/packages/source/n/nwalign/nwalign-0.3.1.tar.gz

First, we deploy our botnet using a real IRC daemon as C&C server. We launch several commands to generate and record distinct traffic patterns between the infected machine and the C&C server.

Second, we derive FSMs out of this recorded traffic, which are then used to generate a Honeyd service emulation script incorporating the abstract protocol behavior.

Finally, we replace the original C&C server with the service emulation host running Honeyd and the generated script. We launch different commands to evaluate, whether the created script can emulate sufficient responses to fool our bot.

6.3.2 Generation of C&C Traffic

First, on the C&C host, we build our custom version of Agobot, which is configured using the file *config.cpp*. Beside some other basic settings, we instruct the bot to connect to our C&C server (IP address 192.168.93.134 and port 31337) and join the channel *#MyMasterThesisBotnet#* using the respective login credentials and a randomly generated nick. Thereby the nick consists of a random combination of letters prefixed by the string "bot-" in order to generate data, which is variable on the one hand, but has a significant meaning on the other hand. In addition we build the bot in debug mode in order to be able to track its activities. The IRC daemon is setup accordingly.

Next, we execute our customized Agobot on the victim host, which connects back to the configured C&C server and attempts to enter the programmed channel (*#MyMasterThesisBotnet#*) awaiting further commands. This initialization can be observed in the debug window, which is depicted in figure 6.4. On the victim host we record the traffic using *WinDump.exe -s 0 -w e:\CundCtraffic.dump proto TCP and (src 192.168.93.134 or dst 192.168.93.134)* Thereby we set the option *-s 0* to avoid a limitation of the recorded packet size, since important information may be truncated otherwise. In order to command the bot we launch a conventional IRC Client (mIRC), connect as botmaster to our C&C Server and join the previously configured channel as well. We instruct the bot to execute a given command by performing a query as outlined in figure 6.5. Thereby we use a full stop as command prefix.

After login to the bot (*.login User password*) we instruct it to perform some harmless actions, such as displaying status information (*.bot.about*, *.bot.sysinfo*, *.bot.id*, *.bot.status*). Thereby we launch a diverse set of commands in order to obtain representative data. In particular we use specific commands, such as .bot.sysinfo, several times (see figure 6.6), since they also query random and regularly changing values (e.g., uptime).

Furthermore we also send some dummy commands and random strings intended to insert "noise". This is afterwards used to examine the result, i.e., none of these commands should appear in the resulting script. We perform a number of such sessions using randomly chosen commands in an arbitrary order. In doing so we simulate a number of distinct bots, since the bot generates a different nick for every session. Finally, the recorded traffic of these conversations is filtered to remove traffic produced by other applications running in background.

6.3.3 Traffic Dissection and FSM Generation

On the service emulation host we dissect the previously recorded traffic dump and extract the used ports within the communication. Since ScriptGen is port based, this analysis is necessary to determine for which ports a corresponding FSM needs to be generated. As a result we receive a list of identified ports and generate a FSM for each port. Therefore we apply the existing functions implemented by ScriptGen:
First, a simplified FSM is built by parsing the traffic dump file for the corresponding data-link type and reassembling the packets and the respective conversations (*sgload.py*). A conversation is composed of messages, whereas a message is the longest set of bytes going in the same direction. Thus it is the starting point to build the FSM. The rebuild of the resulting conversations is based on the unique tuple of source- and destination-address and the corresponding ports. Thus there is no check, whether a given port actually corresponds to the expected protocol but one FSM per port is build. An exemplary excerpt of such a conversation is listed below.

```
=== Item 201
{ <CONV TCP 80 src:('192.168.1.1', 64459) dst:('192.168.1.2', 80)>
        [MSG d:I l:127]
        [MSG d:O l:49]
        [MSG d:I l:178]
        [MSG d:O l:39]
        [MSG d:I l:262]
        [MSG d:O l:39 f:Cc]
}
```

It outlines the mentioned unique tuple (source- / destination-address, source-/destination port) along with the messages, where
d is the direction from the server perspective (I: incoming, 0: outgoing),
l is the length of the payload in bytes and
f describes the set flags (if any). In this example "Cc" refers to "client close".

Next, the functions implemented in *sgattach.py* are called in order to attach data contained in the traffic dump to an eventually existing FSM. In addition these functions could be used to infer content dependencies between known conversations and an existing FSM. The output is a serialized, updated FSM serving as input to *sgbuild.py*, which implements the Region Analysis and builds the actual FSM based on the chosen thresholds for macro-clustering and microclustering as described in 6.2. An excerpt of such a resulting FSM is listed below.

```
<S [TCP:31337:] f:6 kids:1 label_len:158 new:3 >
{ <TR REG f:6,\#r:1> |F|
 <S [TCP:31337:1] f:2 kids:1 label_len:6 new:4 >
 { <TR REG f:2,\#r:13> |F||Mf||F||Mf||F||Mf||F||Mf||F||Mf||F||Mf||F|
  <S [TCP:31337:1|1] f:2 kids:1 label_len:16 new:0 >
  { <TR NULL f:2>
   <S [TCP:31337:1|1|1] f:0 kids:0 label_len:1054 new:2 >
```

It contains information about the used protocol, the observed states and edges as well as the respective transitions, where
S is the self-identifier (i.e., the protocol and port) followed by the path,
f is frequency of the state,
kids describes the number of transitions the state has,
label_len is the length of the state labels,
new is the amount of conversations and
TR (Type Region) describes the identified region type. A region can thereby be *NULL* describing a transient state, i.e., a state with an outgoing NULL transition. That is, a state that immediately leads to a new future state after label generation without expecting a client request. In addition a region can be identified as *Fixed* (containing repeatedly the same data), *Mutating* (containing varying data) or as *REG* (i.e., a regular expression). In turn, a region described via a regular expression may consist of several regions having varying characteristics. This is optionally indicated via corresponding flags, such as *"F"* (fixed region) or *"Mf"* (mutating region). Finally every generated FSM is inspected and basic information such as port, IP address and the identified protocol is extracted for further processing (e.g., a rule-based decision, whether the FSM for the respective port should be enabled in the service emulator). We incorporate the described functionality for traffic dissection and FSM generation into a small Python script (*SE-parsetraffic.py*) whose procedure is described in algorithm 6.3.1.

Algorithm 6.3.1: TRAFFIC DISSECTION AND FSM GENERATION($ip, port$)

$//Traffic-dissection(tcpdump)$:
$load\_tcpdump$
$extract\_ports\_for\_IP$
$//Generate\_FSM$:
for $p\_in\_portlist$:
  do
$extract\_port\_number(p)$
$load\_traffic\_dump - p < p >< IP >< file >$
if FSM_exists
  then $attach\_data\_to\_FSM - p < p >< IP >$
$build\_FSM < microclustering - threshold >< macroclustering - threshold >$
$write\_resulting\_FSM\_to\_file$
$//Parse\_FSM$:
for $\_p\_in\_portlist$:
  do $extract\_used\_protocol(IP, port)$
$write\_to\_file$: $"port < p > ".sh$

6.3.4 FSM Traversal and Script Building

In order to generate a service emulation script out of a given FSM, information about its elements is required. Specifically we need to determine the total number of states and for

each state

- all state labels (the messages sent by the server),
- the total number of edges (the number of possible transitions towards the next state) and for each edge
  - the respective edge label (the message representing the client request triggering the transition)

Thereby we assume that the first seen state label represents the initial message sent by the server (e.g., a service banner). Thus it is defined as the default state. To this end a certain FSM is inspected by traversing through its paths, which are derived from the identified regions. For a given path the corresponding strings and regular expressions representing the incoming and outgoing messages as identified by ScriptGen are extracted. Based on this information the sequence of regular expressions and the respective strings to respond with is reassembled. An excerpt of the FSM inspection is listed below.

```
<<< Incoming
> Fixed region
0000     50  41  53  53  20  31  32  33          PASS 123
0008      D   A                                  ..
%regexp: 'PASS\\ 123\\\r\\\n'

<<< Incoming
> Fixed region
0000     4E  49  43  4B  20  62  6F  74          NICK bot
0008     2D                                     -
> Mutating region
Content candidate
0000     69  68                                 ih
Content candidate
0000     6D  72                                 mr
> Fixed region
0000     77                                     w
> Mutating region
Content candidate
0000     79                                     y
Content candidate
0000     67                                     g
> Fixed region
0000      D   A  55  53  45  52  20  62          ..USER b
0008     6F  74  2D                             ot-
> Mutating region
Content candidate
0000     69  68                                 ih
Content candidate
0000     6D  72                                 mr
> Fixed region
0000     77                                     w
> Mutating region
Content candidate
0000     79                                     y
Content candidate
0000     67                                     g

>>> Outgoing (len: 16)
0000     50  49  4E  47  20  3A  35  44          PING :5D
0008     38  42  34  37  34  34   D   A          8B4744..
0010
```

```
**********************
:192.168.93.134 001 bot-ihwg :Welcome to the Botnet IRC Network bot-ihwg!
bot-ihwg@192.168.93.136
:192.168.93.134 002 bot-ihwg :Your host is 192.168.93.134, running version Unreal3.2.9
:192.168.93.134 003 bot-ihwg :This server was created Sat Nov  5 10:20:48 2011
:192.168.93.134 004 bot-ihwg 192.168.93.134 Unreal3.2.9 iowghraAsORTVSxNCWqBzvdHtGp
lvhopsmntikrRcaqOALQbSeIKVfMCuzNTGjZ
:192.168.93.134 005 bot-ihwg UHNAMES NAMESX SAFELIST HCN MAXCHANNELS=20 CHANLIMIT=\#:20
MAXLIST=b:60,e:60,I:60 NICKLEN=30 CHANNELLEN=32 TOPICLEN=307 KICKLEN=307 AWAYLEN=307
 MAXTARGETS=20 :are supported by this server
:192.168.93.134 005 bot-ihwg WALLCHOPS WATCH=128 WATCHOPTS=A SILENCE=15 MODES=12
 CHANTYPES=\# PREFIX=(qaohv)~\&@%+ CHANMODES=beI,kfL,lj,psmntirRcOAQKVCuzNSMTGZ
  NETWORK=Botnet CASEMAPPING=ascii EXTBAN=~,qjncrR
 ELIST=MNUCT STATUSMSG=~\&@%+ :are supported by this server
:192.168.93.134 005 bot-ihwg EXCEPTS INVEX CMDS=KNOCK,MAP,DCCALLOW,USERIP :are supported
by this server
:192.168.93.134 251 bot-ihwg :There are 2 users and 0 invisible on 1 servers
```

We find, that we can recognize our previously exchanged messages, i.e., the launched commands and their according responses. In particular we conclude, that ScriptGen is able to map also variable values correctly. For instance it identifies *"USER bot-"* and *"NICK bot-"* as fixed regions while defining single letters and their combinations as mutating regions. This matches our configuration instructing the bot to use a variable nick consisting of random letters prefixed by "bot-". Thus, ScriptGen abstracts the protocol semantics correctly in this example. The output of the FSM traversal is then included in the service emulation script file. We implemented this functionality in Python (*SE-FSM-traversal.py*). To this end the static part of the script is created in a first step consisting of a header and some basic functions for echoing fake messages. For basic data exchange we rely upon the "base.sh" script, which is shipped with Honeyd and resides in /usr/share/honeyd/scripts/misc/. In a second step we created a modified version of the traverse function as originally implemented in ScriptGen so that it now traverses through a given FSM path and extracts the regular expressions of the exchanged messages. This basic procedure is outlined in

algorithm 6.3.2.

Algorithm 6.3.2: FSM TRAVERSAL AND SCRIPT BUILDING($FSM, path$)

```
//FSM_traversal :
current_state = fsm
for all_states_in_path
  do if first_seen_server_response
  then dump_current_state_label
state_count++
default_state = current_state
//If_not_the_first_seen_server_response :

  else dump_current_state_label
state_count++
for all_transitions_of_current_state
  do if len(t)! = 1
  then break
//no_more_transitions_for_current_path
  else
t = t[0]
if t.type == t.TYPE_NULL
  then //transient_state
state_count++
current_state = current_state + 1

  else if t.type == t.TYPE_REGION
  then grab_matching_regex_describing_current_state
//Count_states_and_edges_and_determine_content_of_region_type :
for r_in_t.content
  do if r.type == r.TYPE_FIXED
rtype_f = rtype_f + 1
Type = Fixed : (r.content[0])

  else if r.type == r.TYPE_MUTATING
rtype_m = rtype_m + 1

  else if r.type == r.TYPE_REGEXP
rtype_r = rtype_r + 1
Type = Regex : (r.content[0])
//Proceed_with_next_state :
current_state = current_state + 1
```

6.3.5 Results

As a result of the steps described above a service emulation script is generated and can be used with Honeyd. Thereby the regular expressions intended to cover a given class of requests are of special interest. Hence an excerpt of the corresponding sections of the script is listed below.

```
# State S0)
#S0)
        if [[ $_ =~ /'PASS\\ 123\\\r\\\n' ]]; then
                echo -e "  OK "
        else
                 echo -e "ERROR"

 state="S2"
        fi
 ;;
# State S2:
S2)
        if [[ $_ =~ /'NICK\\ bot\\-untuv\\\r\\\nUSER\\ bot\\-untuv\\ 0\\ 0\\ \\:bot\\-un' ]]; then
                echo -e "  ^@^@^@^@^@^@"
         else
                 echo -e "ERROR"

 state="S3"
        fi
 ;;

# State S3:
# transient state (immediately leads to a new future state without expecting a client request)
S3)
# therefore no response is generated -> traverse directly to S4
 state="S4"
 ;;

# State S9:
S9)
        if [[ $_ =~ /'USERHOST\\ bot\\-untuv\\\r\\\n' ]]; then
                echo -e "  :192.168.93.134 302 bot-untuv :bot-untuv=+"
         else
                 echo -e "ERROR"

 state="S10"
        fi
 ;;

# State S10:
S10)
        if [[ $_ =~ /'JOIN\\ \\#MyMasterThesisBotnet\\#\\ 123\\\r\\\n' ]]; then
                echo -e "  :192.168.93.134 302 bot-untuv :bot-untuv=+"
         else
                 echo -e "ERROR"

 state="S11"
        fi
 ;;
```

By interacting with the emulated IRC service we found that it is capable of generating appropriate responses to a set of very basic requests. However, slight deviations of these basic requests cause the emulated service to not respond at all thus leading to insufficient emulation. We believe, that this is related to the limited amount of traffic, that we have generated for this experiment. In fact, the size of the generated traffic dump file is less

than 100kB, which seems clearly insufficient for ScriptGen to learn all interactions properly. Thus further experimentation is necessary and we are confident, that a larger amount of traffic will produce more accurate results. However, we found that generating service emulation scripts using the ScriptGen approach is essentially possible. Specifically we believe, that the basic assumption of ScriptGen (i.e., an exploit performs a limited number of execution paths) can be applied to our use case of service emulation for C&C traffic, since a bot performs a limited number of commands as well. Thus we conclude that the application of ScriptGen within the service provisioning component of our presented approach is feasible.

6.4 Summary

In this chapter we presented our proof of concept implementation leveraging the output of ScriptGen for use within the service provisioning component of our presented approach. In particular we evaluated the feasibility of using ScriptGen for generating service emulation scripts intended to spawn an emulated C&C service. To this end we introduced the functionality of ScriptGen and discussed some implications of its approach. Next, we described the setup used for generating C&C traffic, subsequently used by ScriptGen to abstract the observed behavior. Our setup consists of three distinct machines (i.e., a C&C server, a victim host and a service emulation host), which are used to deploy a minimal botnet using IRC as C&C channel. We build a customized malware using the source code of the Agobot / Phatbot malware family in order to keep full control over all test parameters and to assure reproducibility. Our data set consists of manually generated traffic, which was recorded from launching commands to the bot via IRC. Based on this data set we created a FSM using ScriptGen and accordingly a service emulation script. The original C&C server was then replaced by the service emulation host running Honeyd along with the generated script. We evaluated the results by interacting with the emulated IRC service and found that it is capable of generating appropriate responses to a set of very basic requests. However, slight deviations of these requests cause the emulated service to not respond at all thus leading to insufficient emulation. Thus we conclude that generating service emulation scripts using the ScriptGen approach is essentially possible, although further experimentation is necessary to produce more accurate results.

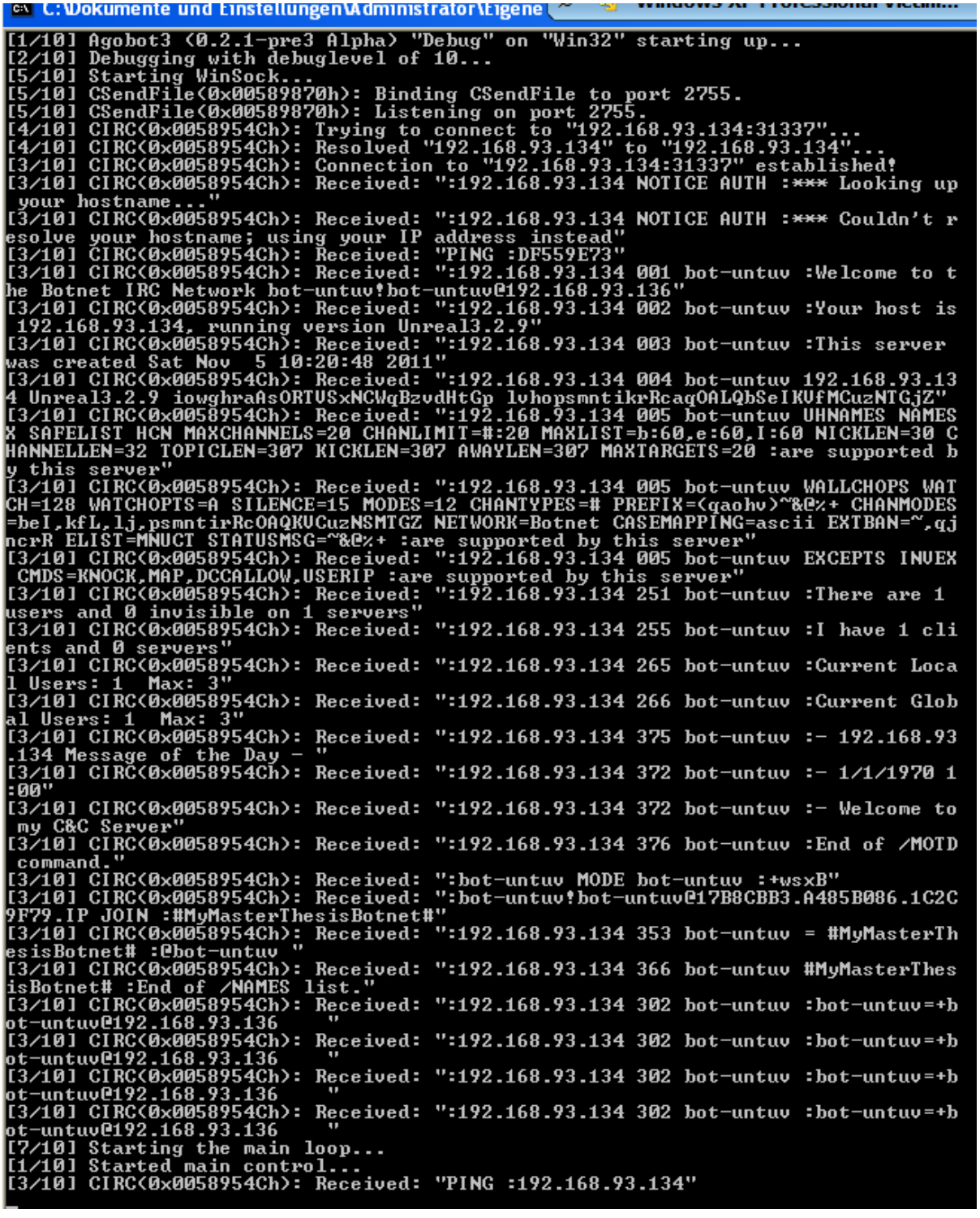

Figure 6.4: Agobot Initialization

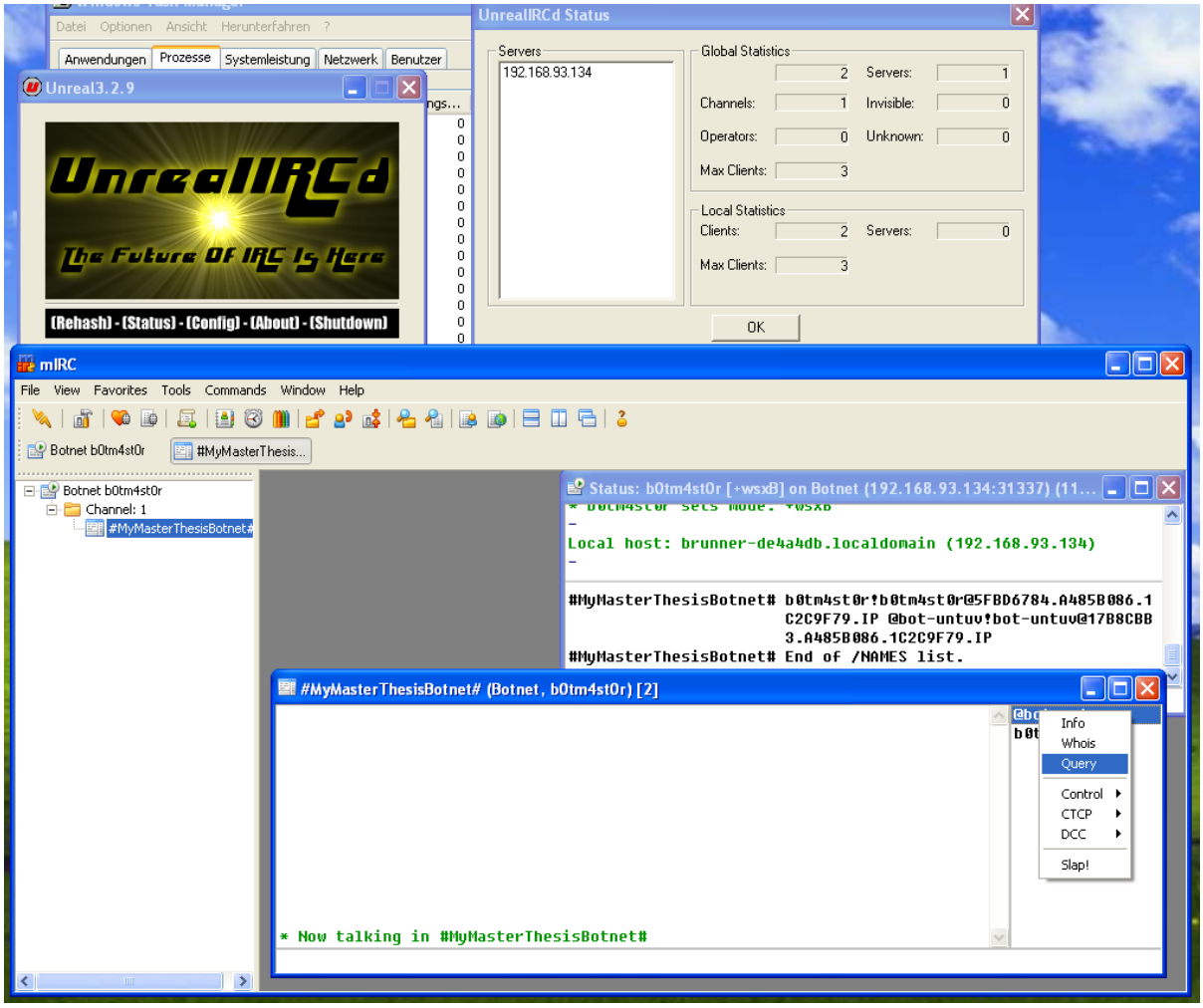

Figure 6.5: Instructing the Bot via the Defined IRC Channel

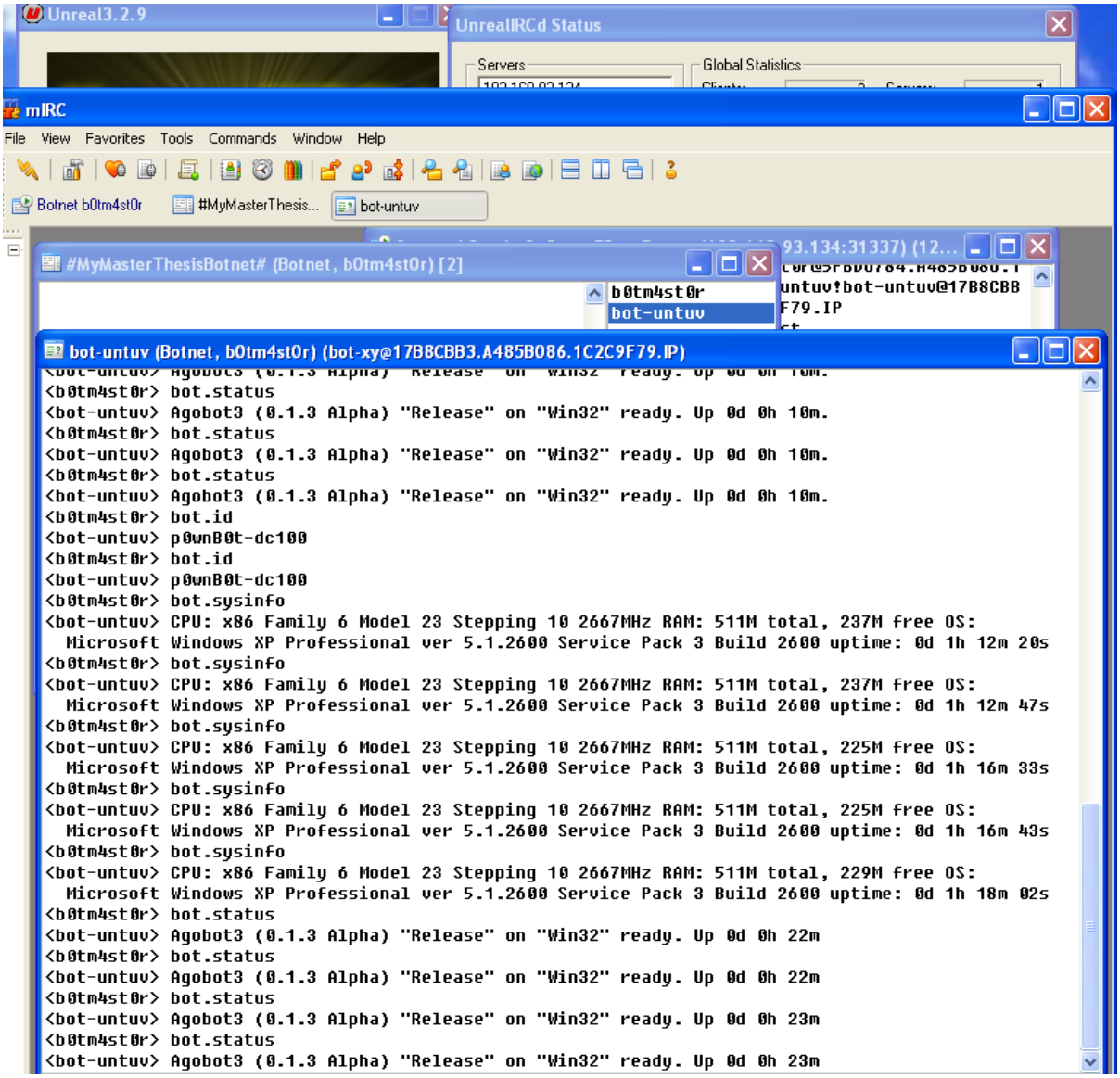

Figure 6.6: Communication Between the Bot and the C&C Server

Chapter 7

Summary

7.1 Conclusion

In this Master's thesis we proposed a holistic approach for integrated malware collection and analysis. For designing the approach we researched the underlying challenges and identified necessary prerequisites. To this end we defined an attacker model and introduced the threat posed by today's malware as well as the corresponding background on cyber crime to the extend it is relevant for the thesis. As a result we specified the life-cycle of modern malware, that has to be considered for a comprehensive analysis. Next, we surveyed the state of the art in honeypot based malware collection and analysis. In particular we described all - to our knowledge - existing honeypot implementations and created an corresponding taxonomy. Accordingly, we compared existing approaches for malware analysis as a decision basis for our approach. Based on the findings we developed a holistic approach for integrated honeypot based malware collection and analysis which addresses identified limitations of existing similar work, namely the separation of collection and analysis, the limitations of service emulation and the operational risk of HI honeypots. The overall goal is to capture and dynamically analyze malware at a large-scale while covering the entire execution life-cycle of a given malware. The key contribution of the approach is the design of the framework as well as the integration and extension of the stated tools. We found that the approach of HI honeypots and dynamic malware analysis fits best to our needs. Specifically we chose to rely on the taint-map based approach of the ARGOS honeypot for malware collection and on the virtual machine introspection framework Nitro for malware analysis. Additionally we developed a concept for providing emulated services to a malware sample under analysis thereby fulfilling possible legal and liability constraints. While the service provisioning element raises several technical issues regarding protocol identification, connection proxying, transparent redirection, payload modification as well as detection, observation and interception of malicious (C&C-) traffic, most of these issues can be addressed based on existing work. We referred to related research in this area and focused on the issue of handling unknown traffic patterns, such as C&C protocols. To this end we presented a proof of concept implementation leveraging the output of ScriptGen for use within the service provisioning component of our presented approach. Using this proof of concept we evaluated the feasibility of using ScriptGen for

generating service emulation scripts intended to spawn an emulated C&C service. We setup a minimal botnet using customized malware in order to generate the corresponding C&C traffic. Out of this recorded traffic we derived FSMs, which were then used to generate a Honeyd service emulation script incorporating the abstract protocol behavior. After replacing the original C&C server with the service emulation host we launched different commands to evaluate, whether the created script can emulate sufficient responses. We found, however, that slight deviations of the used commands cause the emulated service to not respond at all thus leading to insufficient emulation. While we were able to demonstrate that generating service emulation scripts using the ScriptGen approach is essentially possible, further experimentation will be necessary to produce more accurate results.

7.2 Outlook and Future Work

From a conceptional perspective the main limitation of our approach is that it can only capture samples of autonomous spreading malware due to the use of ARGOS. The server based approach of ARGOS, i.e., passively waiting for incoming exploitation attempts, implies that this type of honeypot can not collect malware, which propagates via other propagation vectors such as Spam messages or drive-by downloads. With respect to the outlined paradigm shift in attack vectors we will need to consider such propagation vectors as well in future. However, since this is a sensor issue, this limitation may be overcome by integrating corresponding honeypot types (i.e., client honeypots) into our approach. This is left for future work. Moreover, due to the ongoing spread of IP-enabled networks to other areas (e.g., mobile devices and SCADA environments) our approach will be required to integrate malware sensors covering these attack vectors as well in future.

Beside the necessary further experiments to improve the accuracy of service emulation for C&C traffic, enhancements in malware analysis need to be tested. In particular the interaction between all stated components will be evaluated, once all of them are readily deployed. At the time of writing the evaluation of system calls produced by Nitro needs to be finished and measures for checking malware initiated outbound attempts need to be evaluated. In a next step we will test the use case of tracking and intercepting encrypted C&C protocols.

Bibliography

[ABFM10] M. Apel, J. Biskup, U. Flegel, and M. Meier. Early warning system on a national level - project amsel. In *Proceedings of the European Workshop on Internet Early Warning and Network Intelligence (EWNI 2010)*, January 2010.

[BCH+09] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium, NDSS*, San Diego, California, USA, Feb 2009. The Internet Society.

[BEH+10] Martin Brunner, Michael Epah, Hans Hofinger, Christopher Roblee, Peter Schoo, and Sascha Todt. The fraunhofer aisec malware analysis laboratory - establishing a secured, honeynet-based cyber threat analysis and research environment. Technical report, Fraunhofer Research Institution for Applied and Integrated Security, September 2010.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[BHK+10] Martin Brunner, Hans Hofinger, Christoph Krauss, Christopher Roblee, Peter Schoo, and Sascha Todt. Infiltrating critical infrastructures with next-generation attacks – w32.stuxnet as a showcase threat. Technical report, Fraunhofer Research Institution for Applied and Integrated Security, 2010.

[BHKW05] Paul Baecher, Thorsten Holz, Markus Koetter, and Georg Wicherski. Know your enemy: Tracking botnets. The Honeynet Project, 2005.

[BKH+06] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The nepenthes platform: An efficient approach to collect malware. In *Proceedings of the 9 th International Symposium on Recent Advances in Intrusion Detection (RAID)*, number 4219 in Lecture Notes in Computer Science, pages 165–184. Springer, 2006.

[BSI11] BSI. Die lage der it-sicherheit in deutschland 2011. Bundesamt fuer Sicherheit in der Informationstechnik, May 2011.

[BSSW11] Georg Borges, Joerg Schwenk, Carl-Friedrich Stuckenberg, and Christoph Wegener. *Identitaetsdiebstahl und Identitaetsmissbrauch im Internet: Rechtliche und technische Aspekte*. Springer, 1st edition, January 2011. ISBN 9783642158322.

[BY07] Paul Barford and Vinod Yegneswaran. An inside look at botnets. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 171–191. Springer US, 2007. 10.1007/978-0-387-44599-1 8.

[CAM+08] Xu Chen, J. Andersen, Z.M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177 –186, june 2008.

[CG08] Davide Cavalca and Emanuele Goldoni. Hive: an open infrastructure for malware collection and analysis. In *Proceedings of the 1st workshop on open source software for computer and network forensics*, pages 23–34, 2008.

[CGKP11] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring pay-per-install: the commoditization of malware distribution. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[Che92] Bill Cheswick. An evening with berferd in which a cracker is lured, endured, and studied. In *Proc. Winter USENIX Conference*, pages 163–174, 1992.

[Chu03] Anton Chuvakin. An overview of unix rootkits. iDefense Labs, February 2003. http://www.rootsecure.net/content/downloads/pdf/unix_rootkits_overview.pdf.

[CJM05] Evan Cooke, Farnam Jahanian, and Danny McPherson. The zombie roundup: understanding, detecting, and disrupting botnets. In *Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop*, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

[CLLK07] Hyunsang Choi, Hanwoo Lee, Heejo Lee, and Hyogon Kim. Botnet detection by monitoring group activities in dns traffic. In *Computer and Information Technology, 2007. CIT 2007. 7th IEEE International Conference on*, pages 715 –720, oct. 2007.

[Coh87] Fred Cohen. Computer viruses : Theory and experiments. *Computers & Security*, 6(1):22 – 35, 1987.

[DGLL07] David Dagon, Guofei Gu, Christopher P. Lee, and Wenke Lee. A taxonomy of botnet structures. In *Proc. of the 23 Annual Computer Security Applications Conference (ACSAC'07)*, 2007.

[DRSL08] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, New York, NY, USA, 2008. ACM.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29 Issue 2:198–208, Mar 1983.

[DZL06] David Dagon, Cliff Zou, and Wenke Lee. Modeling botnet propagation using time zones. In *Proceedings of the 13 th Network and Distributed System Security Symposium NDSS*, 2006.

[Eck09] Claudia Eckert. *IT-Sicherheit - Konzepte, Verfahren, Protokolle (6th edition)*. Oldenbourg, 2009. ISBN 978-3-486-58999-3.

[Eri06] Jon Erickson. *Forbidden code*. Mitp, 2006. ISBN 9783826616679.

[ESKK08] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[Fer06] Peter Ferrie. Attacks on virtual machine emulators. In *AVAR Conference, Auckland*, pages 128–143. Symantec Advanced Threat Research, December 2006.

[FHW05] Felix C. Freiling, Thorsten Holz, and Georg Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proceedings of 10 th European Symposium on Research in Computer Security, ESORICS*, pages 319–335, 2005.

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.stuxnet dossier. Technical Report 1.4, Symantec, February 2011.

[FPPS07] Jason Franklin, Adrian Perrig, Vern Paxson, and Stefan Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 375–388, New York, NY, USA, 2007. ACM.

[Fuc11a] Christian Martin Fuchs. Deployment of binary level protocol identification for malware analysis and collection environments. Bacherlor's thesis, Upper Austria University of Applied Sciences, Bacherlor's degree programme Secure Information Systems in Hagenberg, May 2011.

[Fuc11b] Christian Martin Fuchs. Designing a secure malware collection and analysis environment for industrial use. Bacherlor's thesis, Upper Austria University of Applied Sciences, Bacherlor's degree programme Secure Information Systems in Hagenberg, January 2011.

[FW09] Michael Freeman and Andrew Woodward. Smartpot: Creating a 1st generation smartphone honeypot. In *Australian Digital Forensics Conference. Paper 64.*, 2009. http://ro.ecu.edu.au/adf/64.

[Goe08] Jan Goebel. Amun: A python honeypot. Technical report, Laboratory for Dependable Distributed Systems, University of Mannheim, 2008.

[Gor07] Christian Gorecki. Trumanbox - improving malware analysis by simulating the internet. Master's thesis, RWTH Aachen University, 2007.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, NDSS, pages 191–206, San Diego, California, USA, 2003.

[GSN+07] Julian B. Grizzard, Vikram Sharma, Chris Nunnery, Brent ByungHoon Kang, and David Dagon. Peer-to-peer botnets: overview and case study. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 1–1, Berkeley, CA, USA, 2007. USENIX Association.

[Gut07] Peter Gutmann. The commercial malware industry. In *DEFCON 15*, 2007.

[GZL08] Guofei Gu, Junjie Zhang, and Wenke Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *NDSS*. The Internet Society, 2008.

[HEF08] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. Technical report, University of Mannheim, Laboratory for Dependable System, 2008.

[HEF+09] T. Holz, M. Engelberth, F. Freiling, J. Göbel, C. Gorecki, P. Trinius, and C. Willems. Frühe warnung durch beobachten und verfolgen von bösartiger software im deutschen internet: Das internet-malware-analyse system (inmas). In Bundesamt für Sicherheit in der Informationstechnik, editor, *Sichere Wege in der vernetzten Welt (Tagungsband zum 11. Deutschen IT-Sicherheitskongress)*, 2009.

[Hol05] Thorsten Holz. A short visit to the bot zoo. *IEEE Security and Privacy*, 3:76–79, May 2005.

[Hol09] Thorsten Holz. *Tracking and Mitigation of Malicious Remote Control Networks*. PhD thesis, University of Mannheim, 2009.

[HSD+08] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 9:1–9:9, Berkeley, CA, USA, 2008. USENIX Association.

[IHF08] Ali Ikinci, Thorsten Holz, and Felix C. Freiling. Monkey-spider: Detecting malicious websites with low-interaction honeyclients. In *Sicherheit'08*, pages 407–421, 2008.

[ISO] Iso/iec 27000:2009.

[Ken07] McMillan C. Kendall, K. Practical malware analysis. In *Black Hat Conference*, USA, 2007.

[KKB06] Christopher Kruegel, Engin Kirda, and Ulrich Bayer. Ttanalyze: A tool for analyzing malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research (EICAR 2006) Annual Conference*, 4 2006. Best Paper Award.

[KL01] David Karig and Ruby Lee. Remote denial of service attacks and countermeasures. Technical Report CE-L2001-002, Princeton University Department of Electrical Engineering, 2001.

[Kle04] Tobias Klein. *Buffer Overflows und Format-String-Schwachstellen: Funktionsweisen, Exploits und Gegenmassnahmen*. Dpunkt.Verlag GmbH, 2004. ISBN 9783898641920.

[KPL10] Engin Kirda, Veikko Pankakoski, and Tobias Lauinger. Honeybot, your man in the middle for automated social engineering. In *Proceedings of the 3rd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 2010)*, 4 2010.

[KWK+11] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson. Gq: practical containment for measuring modern malware systems. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, IMC '11, pages 397–412, New York, NY, USA, 2011. ACM.

[LD07] Corado Leita and Marc Dacier. Sgnet: A distributed infrastructure to handle zero-day exploits. Research Report RR-07-187, Institut Eurécom - Department of Corporate Communications, 2007.

[LD08] Corrado Leita and Marc Dacier. Sgnet: A worldwide deployable framework to support the analysis of malware threat models. *Seventh European Dependable Computing Conference*, 0:99–109, 2008.

[LKMC11] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection (RAID) Symposium*, 2011.

[LMD05] Corrado Leita, Ken Mermoud, and Marc Dacier. Scriptgen: an automated script generation tool for honeyd. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 203–214, Washington, DC, USA, 2005. IEEE Computer Society.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[MBGL06] Alexander Moshchuk, Tanya Bragin, Steven D Gribble, and Henry M Levy. A crawler-based study of spyware on the web. *Proceedings of the 2006 Network and Distributed System Security Symposium*, pages 17–33, 2006.

[McC03] Bill McCarty. Botnets: Big and bigger. *IEEE Security and Privacy*, 1:87–90, 2003.

[MFHM08] Michael Mueter, Felix Freiling, Thorsten Holz, and Jeanna Matthews. A generic toolkit for converting web applications into high-interaction honeypots. University of Mannheim, 2008. Citeseerurl 10.1.1.89.5000.

[Mit02] Kevin D. Mitnick. *The Art of Deception*. Wiley, 1st edition, October 2002. ISBN 0471237124.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[Nas05] Troy Nash. An undirected attack against critical infrastructure - a case study for improving your control system security. Case study series: Vol 1.2, Vulnerability & Risk Assessment Program (VRAP), Lawrence Livermore National Laboratory, September 2005.

[Naz09] Jose Nazario. Phoneyc: a virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'09, pages 6–6, Berkeley, CA, USA, 2009. USENIX Association.

[NBH08] Kara Nance, Matt Bishop, and Brian Hay. Virtual machine introspection: Observation or interference? *IEEE Security and Privacy*, 6(5):32–37, September 2008.

[Neu66] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, Champaign, IL, USA, 1966.

[Nor09] Norman. Norman green book on analyzing malware. Whitepaper, 2009.

[Oll11] Gunter Ollmann. Behind todays crimeware installation lifecycle: How advanced malware morphs to remain stealthy and persistent. Whitepaper, Damballa, 2011.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[PGPL11] Daniel Plohmann, Elmar Gerhards-Padilla, and Felix Leder. Botnets: Detection, measurement, disinfection & defence. European Network and Information Security Agency (ENISA), 2011.

[PH08] N. Provos and T. Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Addison-Wesley, 2008. ISBN 9780321336323.

[PMP08] Michalis Polychronakis, Panayiotis Mavrommatis, and Niels Provos. Ghost turns zombie: exploring the life cycle of web-based malware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 11:1–11:8, Berkeley, CA, USA, 2008. USENIX Association.

[Pol10] Bartlomiej Polot. Adapting blackhat approaches to increase the resilience of whitehat application scenarios. Master's thesis, Technische Universität München, June 2010.

[Pro04] Niels Provos. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*, pages 1–14, 2004.

[Pro05a] The Honeynet Project. Know your enemy: Genii honeynets, May 2005. accessible from: http://old.honeynet.org/papers/gen2/index.html.

[Pro05b] The Honeynet Project. Know your enemy: Honeywall cdrom roo, August 2005. accessible from: http://old.honeynet.org/papers/cdrom/roo/index.html.

[Pro06] The Honeynet Project. Know your enemy: Honeynets, May 2006. accessible from: http://old.honeynet.org/papers/honeynet/.

[PSB06] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks. In *Proc. ACM SIGOPS EUROSYS'2006*, pages 15–27, Leuven, Belgium, April 2006. ISBN 1-59593-322-0.

[PSE11] Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, November 2011.

[PW10] Vern Paxson and David Wagner. Worms, botnets and the underground economy. Lecture CS 161 - Computer Security, UC Berkeley, April 2010.

[RMEM08] Jamie Riden, Ryan McGeehan, Brian Engert, and Michael Mueter. Know your enemy: Web application threats. The Honeynet Project, 2008.

[RSL+10] Konrad Rieck, Guido Schwenk, Tobias Limmer, Thorsten Holz, and Pavel Laskov. Botzilla: detecting the "phoning home" of malicious software. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1978–1984, New York, NY, USA, 2010. ACM.

[Rut04] J. Rutkowska. Red pill... or how to detect vmmusing (almost) one cpu instruction, 2004. http://invisiblethings.org.

[Rut06] Joanna Rutkowska. Introducing stealth malware taxonomy. COSEINC Advanced Malware Labs, November 2006.

[RVKM10] Lukas Rist, Sven Vetsch, Marcel Kossin, and Michael Mauer. Know your tools: Glastopf - a dynamic, low-interaction web application honeypot. The Honeynet Project, 2010.

[SBY+08] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper.*, Hyderabad, India, December 2008.

[SGCC+09] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 635–647, New York, NY, USA, 2009. ACM.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86. In *Proceedings of CCS 2007*. ACM Press, 2007.

[Shi00] R. Shirey. Request for comments: 2828 internet security glossary, May 2000.

[SL04] Stephen M. Specht and Ruby B. Lee. Distributed denial of service: taxonomies of attacks, tools and countermeasures. In *Proceedings of the International Workshop on Security in Parallel and Distributed Systems, 2004*, pages 543–550, 2004.

[Spa88] Eugene H. Spafford. The internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-2004, 1988.

[Spi02] Lance Spitzner. *Honeypots. Tracking Hackers*. Addison-Wesley, 2002. ISBN 978-0321108951.

[SS06] Christian Seifert and Ramon Steenson. Capture - honeypot client (capture-hpc), 2006. Available from https://projects.honeynet.org/capture-hpc.

[SSH+07] Christian Seifert, Ramon Steenson, Thorsten Holz, Bing Yuan, and Michael A. Davis. Know your enemy: Malicious web servers. The Honeynet Project, August 2007.

[SWK06a] Christian Seifert, Ian Welch, and Peter Komisarczuk. Honeyc - the low-interaction client honeypot, 2006. Citeseerurl 10.1.1.61.6882.

[SWK06b] Christian Seifert, Ian Welch, and Peter Komisarczuk. Taxonomy of honeypots. Technical Report 11, VICTORIA UNIVERSITY OF WELLINGTON, School of Mathematical and Computing Sciences, June 2006. Technical Report CS-TR-06/12.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, February 2005. ISBN: 0-321-30454-3.

[Tri07] Philipp Trinius. Omnivora: Automatisiertes sammeln von malware unter windows. Master's thesis, RWTH Aachen University, September 2007.

[WBJ+06] Yi-min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. Automated web patrol with strider honeymonkeys : Finding web sites that exploit browser vulnerabilities. *Microsoft Research, Redmond*, pages 35–49, 2006.

[WHF07] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

[Wic06] Georg Wicherski. Medium interaction honeypots, April 2006.

[ZDS+08] Li Zhuang, John Dunagan, Daniel R. Simon, Helen J. Wang, and J. D. Tygar. Characterizing botnets from email spam records. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 2:1–2:9, Berkeley, CA, USA, 2008. USENIX Association.

[ZHH+07] Jianwei Zhuge, Thorsten Holz, Xinhui Han, Chengyu Song, and Wei Zou. Collecting autonomous spreading malware using high-interaction honeypots. In *Proceedings of the 9th international conference on Information and communications security*, ICICS'07, pages 438–451, Berlin, Heidelberg, 2007. Springer-Verlag.

[ZZQL03] Feng Zhang, Shijie Zhou, Zhiguang Qin, and Jinde Liu. Honeypot: a supplemented active defense system for network security. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 231 – 235, aug. 2003.