VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

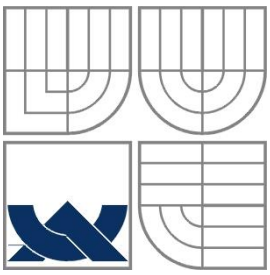
EXTENTED FUNCTIONALITY OF HONEYPOTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

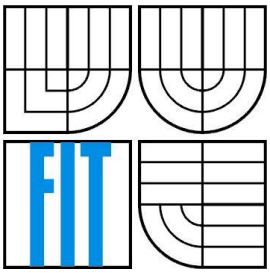
AUTOR PRÁCE
AUTHOR

PETER SOÓKY

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍŘENÉ FUNKCE HONEYPOTŮ

EXTENTED FUNCTIONALITY OF HONEYPOTS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETER SOÓKY

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR MATOUŠEK, Ph.D.

BRNO 2015

Abstrakt

Bakalářská práce se zabývá implementaci rozšíření funkce honeypotu. Práce obecne popisuje honeypoty a zabývá s algoritmy a metodami používanými u implementaci rozšíření. Následně se zabývá testováním implementovanych časti a prezentaci výsledků. V další části práci je provedeno vyhodnocení výsledků.

Abstract

This thesis deals with the implementation of extended functionality of honeypots, which includes the description of honeypots in general and it also deals with the algorithms and methods used in the implementation process. Furthermore, this paper summarizes testing the implemented parts and presenting the accomplished results. Evaluation of these results is also presented in this work.

Klíčová slova

Honeypot, Honeynet Project, Conpot, Glastopf, ICS/SCADA, Webový honeypot

Keywords

Honeypot, Honeynet Project, Conpot, Glastopf, ICS/SCADA, Web application honeypot

Citace

Peter Soóky: Extented functionality of honeypots, bakalářská práce, Brno, FIT VUT v Brně, 2015

Extented functionality of honeypots

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Petra Matoušeka. Další informace mi poskytla pani Katarina Durechova. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Peter Soóky
May 20, 2015

Poděkování

Chtěl bych poděkovat za trpělivost, nápady a náměty vedoucímu práce Ing. Petrovi Matoušekovi a dále pani Katarine Durechovej za poskytnutí odborné pomoci a nezbytnou podporu při zpracování této práce.

Contents

1 Introduction

Day by day, more and more computer systems are connected into networks all over the world. As the accessibility of these systems is growing rapidly, information security is becoming a major concern in business environments. An increasing number of complex attacks demand improved early warning detection capabilities and better countermeasures that prevent intruders to gain access to computer systems. The traditional approach to information security has been defensive so far, but new strategies in a more aggressive forms of defense appear to supplement the existing methods. One of these methods involves the use of various honeypots or honeynets to detect and learn from cyber-attacks to improve security. Understanding our vulnerabilities and our adversaries' capabilities, allows us to create better defensive and offensive plans. This idea is very similar to the concepts of conventional warfare.

The aim of this thesis is to provide basic explanation about honeypots, to portray the extension process of already existing honeypot software and present the enhanced functionality in different scenarios.

1.1 Problem description

This paper will present an overview of honeypots and honeynets in general, explain some of the functionality and concepts of these systems, furthermore it will examine the implementation process of various extensions to already existing honeypots.

Honeypots can be involved in different aspects of information security, such as detection, prevention and information gathering. A honeypot is a security resource, whose value lies in unauthorized or illicit use of that resource. Honeypots have no production value, anything going to or from a honeypot is likely a probe, attack or compromise, ensued from its concept: nobody should use a honeypot, therefore any transaction or interactions with a honeypot are unauthorized. This approach excludes the possibility of false alerts whether positive or negative, granting advantage against other network monitoring techniques, like intrusion detection systems.

Honeypots have tremendous potential for the security community, and these systems can accomplish goals few other technologies can. However like any other technology, they have some challenges to overcome. The significance of this paper is to help to address some of these issues in the development of honeypot systems and contribute the possible solutions to the white hat community.

1.2 Motivation

The motivation of this thesis is to understand how various security systems work and provide protection against different kinds of attacks. We will learn the concepts and architectures of these security systems. Once the extensions are implemented, we will test it against various attacks representing real-life scenarios. Therefore, we will acquire knowledge including various fields as networking, security and information gathering.

1.3 Goals

The perspective is to solve the problems related to security, explore and demonstrate methods that security systems use. Furthermore, we will look into strategies to increase efficiency of honeypots. The goal of this paper is to familiarize the reader with the basic theory of honeypots and present the enhancements' significance to the community. One main aspect is to implement numerous extensions to an already existing honeypot, test it against different attack methods, evaluate the results, and merge the newly created modules with the main project on GitHub. Additionally, the expediency of the realized improvements will be discussed with members of the CZ.NIC association.

1.4 Restrictions

We will use a variety of noncommercial software throughout the thesis during the implementation and evaluation process, e.g. Ubuntu, Debian, Conpot, Glastopf, PLCscan, IPMItool, tcpdump etc. We won't look into the hardware properties in detail. The provided tests on the honeypots are also restricted by available equipment, which determines limits of the experiment.

1.5 Report structure

The thesis report is followed by a chapter describing honeypots in general, the concepts and architecture of such systems. Moreover, we specify the possible types of honeypots, the intended usage and purpose of each type along with their advantages and disadvantages. The next two chapters sum up the specified software we will be working with, as well as the security problems related to the honeypots including the approach of implementation and testing of the solutions. In the last section we conclude and provide our opinion about the results and the future of honeypots.

2 Honeypots and honeynets

In this section we will explain what a honeypot is, what is its purpose and what are the benefits to use one. First we will describe honeypots in general and the terms used in the realm of these systems. In the next subsection we divide honeypots into categories and specify each of them, while in the third subsection we explain the idea behind honeynets. Finally in this chapter we mention the organization behind the concepts of honeypot systems and characterize the two honeypots, for what this thesis provides the detailed report of extensions.

2.1 Honeypot

A honeypot is a deception trap, designed to entice an attacker into attempting to compromise an information system. These decoy servers or systems are setup to collect valuable information regarding the intruder during an attack. A deployed honeypot can serve as an early-warning and advanced security surveillance tool. Honeypots can minimize the risks from attacks on systems and networks, analyze the intruders' attempts to compromise the information system, which provides detailed insight into the system's potential loopholes. The main purpose of honeypots is to detect, deflect or to provide countermeasures in some manner against non-authorized users of the information system.

Honeypots are typically virtual machines, designed to emulate real machines, creating the appearance of running services and applications with open ports that might be found on a usual system or network. It works as a bait by luring attackers to compromise a fake system and fooling them into believing it is a legitimate system. Although it seems like to be a part of a network, it's isolated, monitored and observed covertly. In a scenario where an intruder attempts to compromise the honeypot, valuable attack-related information are gathered. This provides useful information for analyzing attacking techniques and methods, which helps improve the protection of the real system or network.

Often honeypots are used in conjunction with intrusion detection systems (IDS) to extend the other system's capabilities. IDSes and honeypots differ fundamentally in the way they attempt to detect malicious traffic. IDSes have the advantage of monitoring all traffic, flagging threats through a combination of known attack signatures and statistical anomalies. However there is a possibility of false positive alerts due to pattern mismatch or false negative alerts on actual attacks. Using honeypots to complement the IDSes can fill the gaps left by conventional IDSes and can result in a dramatically improved intrusion detection.

2.2 Classification of honeypots

The following subsections will describe the different kinds of honeypots and categorize them according to their properties. The type of the honeypot also defines their goal and usage. Honeypots may vary according to their interaction level, purpose, implementation principle, service emulation, etc.

One of the main specifications of a honeypot system is its level of interaction and general purpose. These qualifications of a honeypot define the system's basic architecture, the stage of risk the honeypot introduces to its environment and the concept of its implementation.

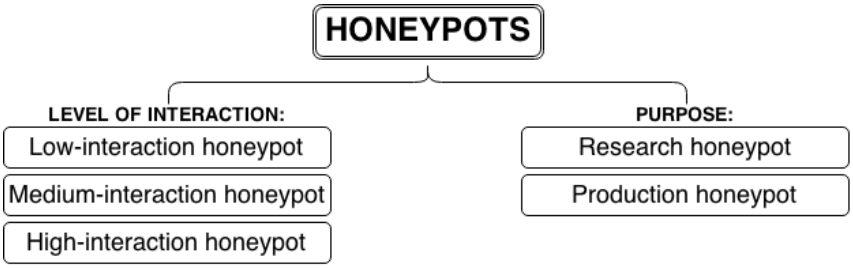


Figure 1: Classification of honeypots

This categorization in the image above can be divided into smaller sections differentiating the type of service emulation honeypots provide, such as server-side or client-side emulation. Server-side emulation honeypots are imitating network services. These honeypots have the same architecture as a server would have, while passively waiting for an attack engagement. Client-side emulation honeypots, by contrast, are more aggressive, and with a client software, like a web browser are actively looking for suspicious servers or other dangerous Internet locations that attack clients. The main target of client honeypots are browsers, but any client that interacts with servers can be part of a client honeypot, including FTP, SMTP, SSH, etc. [1]. Whereas all accesses to a more traditional server based honeypot are malicious, the client-side honeypot must discern which server is malicious and which is benign. This classification is achieved by analyzing the interaction with the servers via static analysis and signature matching based on pre-established definition of "malicious" activity. A different approach is to use a dedicated operating system as an actual vulnerable client and search for unauthorized state changes in the OS after each interaction with the server. However, this approach requires an additional component with some sort of containment strategy to prevent successful attacks from spreading beyond the client. This is accomplished through the use of firewalls and virtual machine sandboxes.

A more detailed classification can contain a section of honeypots divided into groups according to their implementation concept. In accordance with these characteristics the honeypots can be segregated as physical and virtual systems. Physical honeypots are running on a real machine, suggesting that it could be high level of involvement system and able to be compromised completely. Physical honeypots are expensive to maintain, difficult to install and configure, making them impractical to deploy for large

address spaces. Virtual honeypots use only one real machine to run several virtual machines that act like honeypots. This concept reduces the cost of management compared to physical honeypots. However the use of virtual machines are limited by the hardware virtualization software and the host operating system. It is also easier to fingerprint a virtual honeypot, as opposed to physical ones deployed with real hardware, hence the presence of virtualization software and signatures of the virtual hardware emulated by a virtualization software.

Honeytokens are also considered as another type, even if they don't belong to the mentioned categories. These are fake digital entities that have no authorized use, giving them the same power as any other honeypot has. These entities could be embedded into databases, file servers or some other repositories and then intrusion detection systems can have signatures customized to look for these specific tokens. This thesis has no intention to provide detailed information on this topic, hence its minimal correlation with the thesis' aim. Further information on honeytokens can be found in [2].

2.2.1 Level of interaction

Honeypots can be differentiated according to their levels of interactions into high-interaction, medium-interaction and low-interaction honeypots. This degree of interaction determines the level of involvement allowed between an intruder and the system itself. Furthermore, the level of interaction defines the deployment strategies, the grade of necessary maintenance and the risks introduced to the existing network.

2.2.1.1 High-interaction honeypots

High-interaction honeypots are complex systems imitating the activities of a production system that host a variety of services. It involves real operating systems, services and applications that attackers can interact with, and therefore it involves higher risks. The goal of a high-interaction honeypot is to provide the attacker with a system, where nothing is restricted, hence producing more accurate and valuable data for further analysis. Although, giving the intruders real systems to interact with, there are no restraints imposed on attack behavior, in a worst case scenario the attackers can compromise and take over a honeypot system, this allowing the intruders to use it for further network penetration. In this case the honeypot might need to be completely disconnected from the network to prevent serious damage. In general high-interaction honeypots provide more security and are less likely to be detected, though are highly expensive to maintain. A notable drawback to these systems is that the number of

honeypots in a network is limited and the risk associated with high involvement honeypots is significantly higher.

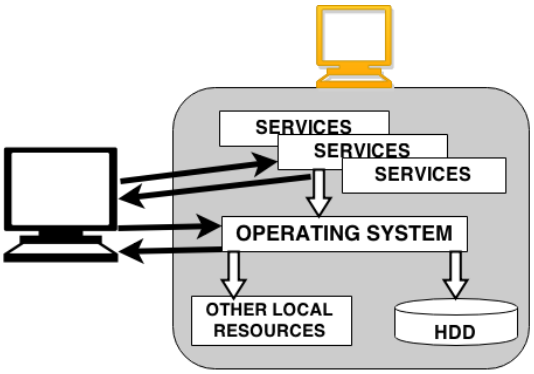


Figure 2: High level of interaction honeypot

2.2.1.2 Medium-interaction honeypots

Medium-interaction honeypots are less sophisticated than high-interaction honeypots, but more advanced than low-interaction honeypots. These honeypots don't have a real operating system, however the simulated services are more complex comparing to low-interaction honeypots and the probability of the aggressor compromising the system is still low. The key feature of medium interaction honeypots is application layer virtualization. These honeypots provide sufficient responses that known exploits await, trick them into sending their payload. Once the payload has been received, the shell code is extracted and analyzed. Then the honeypot emulates the actions the shell code would perform.

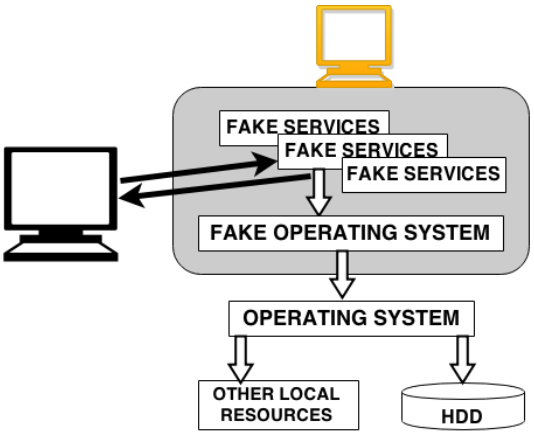


Figure 3: Medium level of interaction honeypot

2.2.1.3 Low-interaction honeypots

Low-interaction honeypots simulate only certain services while consuming relatively few resources though with limited interaction. The intruders' activities are limited to the level of emulation provided by the honeypot, also the obtainable information is restricted to this level as well. On this level there is no operating system for the intruder to interact with [3]. It allows the simulation of virtual network topologies using a routing mechanism that mimics various network parameters. Low-interaction honeypots are easier to deploy and maintain, however their architecture only provides a restricted

framework within which emulation is carried out. Due to the limited number of emulated services, it is easier to fingerprint such system.

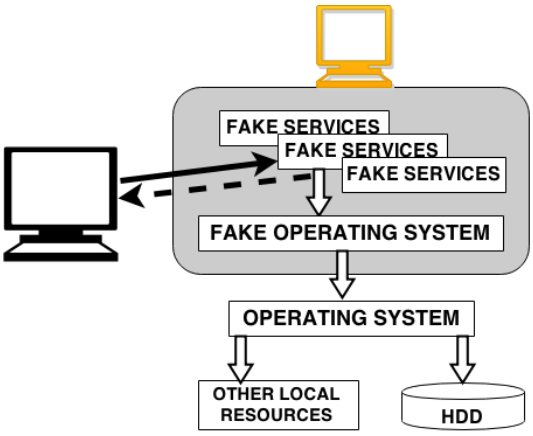


Figure 4: Low level of interaction honeypot

2.2.2 Purpose of honeypots

2.2.2.1 Research honeypots

Research honeypots are mostly used by military, research or government organizations. They are designed to gather valuable information on the general threat organizations may face. Their main function is to obtain extensive information about attack vectors, motives, behavior, various methods and techniques, hence provide more accurate insight about the attacks. Research honeypots are complex systems, requiring more skill and time to deploy and maintain.

2.2.2.2 Production honeypots

Production honeypots are used within an organization's environment mainly for risk mitigation. These are easier to build and deploy, thus providing less functionality. These honeypots often emulate specific services of a production network to entice attackers to interact with them. The data provided by these honeypots can be used for fixing system loopholes, and to build better defenses and countermeasures against potential future threats. Typically these systems work as extension to IDSes performing a more advanced detection function. Some production honeypots are capable to slow down or even shut down different attacks, and in this way these can be used as reconnaissance or deterrence tools.

2.3 Honeynets

"A honeynet is a network of high interaction honeypots that simulates a production network and configured such that all activity is monitored, recorded and in a degree, discreetly regulated [4]."

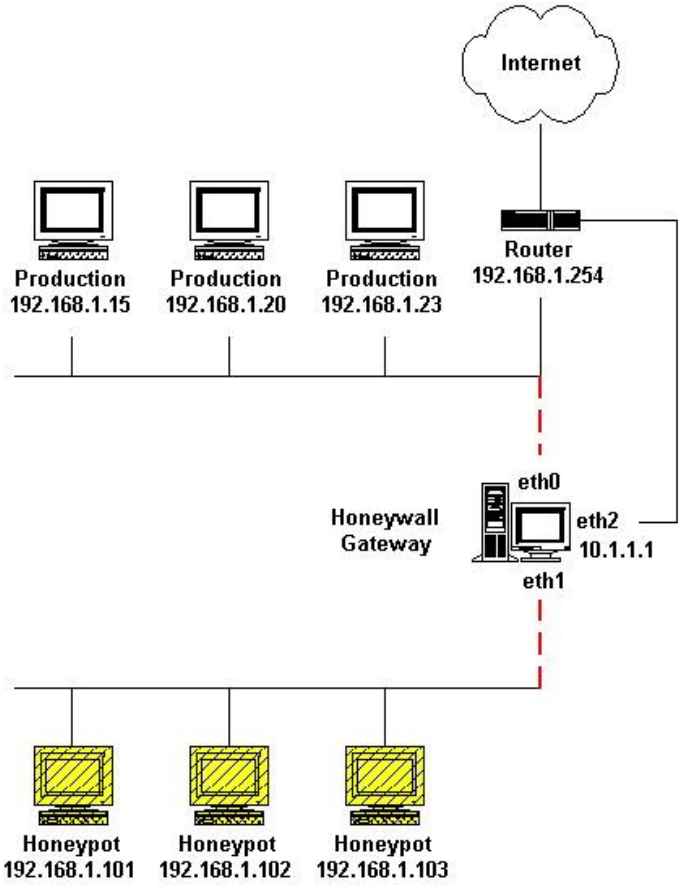


Figure 5: Example of a honeynet Source: [4]

The value of honeypots can be increased by building them into a network, providing attackers a realistic network of systems to interact with. In addition to the honeypots, a honeynet usually has real services and applications running, imitating a real network. Hence honeypots are no production systems, the honeynet itself has no production activity either. As a result, any interaction with a honeynet implies malicious or unauthorized activity. Any connections initiated inbound to a honeynet are most likely a probe, scan, or attack. Any unauthorized outbound connections from a honeynet imply someone has compromised a system and has initiated outbound activity [4].

Deploying such system requires at least a honeypot and a honeywall. In a scenario where the attacker is given a honeypot with real operating system, there is a chance the attacker can obtain full access to the system. To reduce such risks a firewall is configured on the honeywall, which limits the outbound connections and restricts the access to the production network. The honeywall maintains an IDS, which monitors all traffic on the honeypot. While honeynets allow the simulation of realistic productive environments, the risk is significantly higher opposed to a single honeypot.

2.4 The Honeynet Project

Honeynet Project is one of the leading organizations dedicated to computer security research and information sharing. The organization was founded in 1999, and according to their website The Honeynet Project's purpose is to: raise general awareness about threats and vulnerabilities, share information about attackers with the security community, and to share technology and methods with organizations that want to develop similar security research. The group informally began in April, 1999 as the Wargames mailing list. As the group has grown, it became the Honeynet Project in June, 2000. During the years of development the Honeynet Project created several publications on honeypot technologies and introduced efficient ways of building honeypots. Today it includes dozens of active chapters all around the world with many security professionals dedicated to information security[1].

In this chapter we described the general terms related to the thesis's topic, we also explained the differences between various types of honeypots. The next sections reveal the honeypots we will be working with, then we clarify the architecture and the idea behind these systems.

2.5 Conpot

In the following subsections we introduce the various software used in this thesis. We show the concepts behind the programs and proceed to explain the structure of the honeypots in detail.

Conpot is a low interactive server side Industrial Control Systems honeypot designed to be easy to deploy, modify and extend. The honeypot provides a range of common industrial control protocols capable to emulate complex infrastructures to convince an adversary that he just found a huge industrial complex. To improve the deceptive capabilities, Conpot also provides the possibility to serve a custom human machine interface to increase the honeypots attack surface. The response times of the services can be artificially delayed to mimic the behavior of a system under constant load. The honeypot provides complete stacks of the protocols, allowing Conpot to be accessed with productive HMI's or extended with real hardware. Conpot is developed by the Honeynet Project in collaboration with other organizations[2].

The first release was published in May 2013, since then the project has gone a long way and even today the project is still in active development. Conpot is an object-oriented implementation of a low-interaction ICS/SCADA honeypot written in Python. Industrial Control Systems (ICS) are computer-based systems that monitor and control different industrial processes. The largest subset of ICS is

[1] http://www.honeynet.org/about
[2] http://conpot.org

supervisory control and data acquisition systems (SCADA), which are complex systems operating with coded signals over communication channels to provide control of a remote equipment, generally referring to control systems that span a large geographical area.

2.5.1 The structure of Conpot

This subsection lists the main individual parts of the master branch of Conpot release 0.3.1. The root directory contains the informational texts in separate files, such as the Changelog.txt, LICENSE.txt, README.rst, requirements.txt and also the necessary installation scripts. Beside these files the root directory includes three other subdirectories as well: bin, conpot, docs.

The bin/ directory holds the main script for running Conpot, and other several other scripts, which are not necessary for running the software, however they provide extended functionality to the honeypot.

The conpot/ directory contains the basic configuration file and other directories with the source codes of protocol and emulator implementations, utilities, testing suites and templates.

The docs/ directory includes the documentation files and all the necessities for creating and maintaining Conpot's documentation.

2.6 Glastopf

Glastopf is a dynamic, low interactive web application honeypot which emulates thousands of vulnerabilities to gather data from attacks targeting web applications. The project was founded in 2008 by Lukas Rist. The principle of Glastopf is similar to a normal web server: upon a received request it gets processed, and a response is returned. The key to success is to provide proper reply for every request from the attacker exploiting the web application and convince the aggressor that it is a vulnerable, but legit web server.

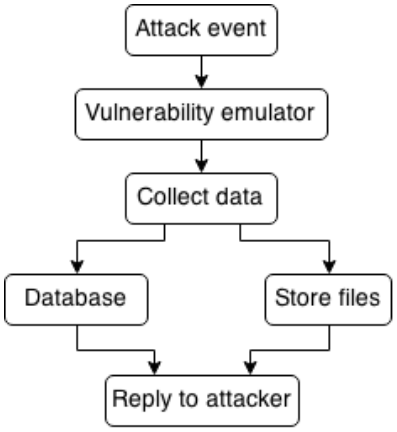


Figure 6: General principle of Glastopf

Glastopf is a minimalistic web server written in Python. The honeypot collects information about web application-based attacks like remote file inclusion, SQL injection, and local file inclusion attacks. Glastopf scans the incoming request for predefined strings, in case of a match, the honeypot tries to download and analyze the file and respond as close as possible to the attacker's expectations. The collected data about the attack vector and the aggressor is stored in a database [10].

2.6.1 Structure of Glastopf

This subchapter portrays the structure of Glastopf release 3.1.2, and describes the main parts of the master branch on Github. The root directory contains general information about Glastopf in separate text files and also the installation scripts. The root directory contains four additional subdirectories as well: bin, glastopf, docs.

The bin/ directory contains the glastopf-runner script, which serves as the main script of Glastopf. It handles the argument parsing, prepares a specific environment, creates an instance of the Glastopf honeypot and the Web Server Gateway Interface of Glastopf and starts up the logging procedure.

The glastopf/ directory includes additional scripts and configuration files to the Glastopf honeypot. Furthermore it contains three subdirectories: modules/, sandbox/ and testing/. The modules/ directory has all the scripts responsible for the service emulation, event logging and data storing procedure. The sandbox/ directory stores the files necessary to create and run a sandbox environment in order to contain and prevent the malicious code to cause damage. The sandbox is further used to analyze and evaluate the actions of the captured malicious software. The testing/ directory has the testing suites for each implemented feature of Glastopf.

The docs/ directory contains the sphinx documentation files and all the necessary scripts for creating and maintaining Glastopf's documentation.

This section discussed the honeypots that will be developed during the thesis along with their structure and the concepts behind these software. In the next chapter we will describe the realization of the provided solutions for both honeypot. We will segregate the individual issues and provide an in depth explanation about the problem and its solution.

3 Implementations within Conpot

The following chapters will describe the analysis and the implementation process of the realized extensions for Conpot and Glastopf. First, the problem and the related terms will be explained. Next, the analysis of the issue will clarify the possible solutions along with their evaluation. The implementation process of the chosen option will be presented together with the explanation of the more important fragments from source code. Finally the realized solution will be tested as a part of the honeypot and the result will be demonstrated in a subchapter.

3.1 Introduction to the extensions

The aim of the thesis to present the enhancements implemented to the honeypots, which will be described in the subsequent parts of this paper along with their evaluation. We will look into various issues listed on the honeypots' GitHub repositories and try to provide a solution for each of them acceptable to be part of the final version of the honeypot software. The selected issues to be solved on Conpot's part are the following:

- Fixing the problem with Conpot MIB cloner's methods
- Adding a feature to Conpot to make it able to change the MAC address
- Separating the STIX and TAXII configuration for clarification
- Resolving the issue with the UID handling in the MODBUS protocol implementation
- Adding a support for BACnet protocol
- Investigating the emulation under IPMI protocol

The other main part of this project is to implement solutions to the listed problems in Glastopf:

- Fixing the issue with the comments on Glastopf's emulated website
- Modifying the database schema containing information about the attacker
- Resolving the problem with the HTTP response status codes
- Implement logging in Glastopf's profiler

3.1.1 Development environments

This subsection is dedicated to the description of the development environment used to solve the security problems related to the honeypots.

All the extensions of Conpot were implemented and tested in a virtualized environment of x86 architecture running distribution of GNU/Linux Ubuntu 12.04.4 LTS (Precise Pangolin) Desktop developed by Canonical/Ubuntu Foundation. The Ubuntu build is based on Debian's architecture

version (Wheezy) and shipped with linux-kernel version 3.11 by default. Glastopf's enhancements were implemented on a virtual instance of Debian 7.0 (Wheezy) with amd64 (64-bit) architecture running linux-kernel version 3.2. This thesis is not supposed to cover the installation process of the operating systems, only to provide general information about the systems the results were achieved. For a detailed guide see the Ubuntu Documentation Project[3] and the Debain Wheezy Installation Guide[4].

The upcoming sections will discuss the issues in depth to provide an overview about each problem. The description of approach is the following: first, the problem analysis and the related terms will be covered. After the illustration the problem's occurrences will be presented. Finally, the implementation and testing of the extensions will be documented here as well.

3.2 Issue with Conpot's cloner module

3.2.1 Problem description

This problem occurs when Conpot's cloner module tries to execute the walk_command method implemented in a built-in SNMP client located at conpot/tests/helpers/snmp_client.py. The purpose of this command is to obtain information from a Management Information Base (MIB) described in the SNMP protocol's template file. The module's initial feature is to be able to clone Conpot's configuration just from crawling the honeypot.

The next subsections will explain some of the terms related to this problem more in depth and portray this specific issue with Conpot's functionality. Furthermore the provided solution will be elucidated with commented snippets from the source code.

3.2.2 Conpot's walk_command method

Conpot's SNMP client uses the pySNMP library, which is a cross-platform, pure-Python SNMP engine implementation. It features fully-functional SNMP engine capable to act in Agent, Manager and Proxy roles, talking SNMP v1/v2c/v3 protocol versions over IPv4/IPv6 and other network transports[5].

The implemented method creates an instance of an SNMP manager and calls a CommandGenerator class's method in pySNMP, which performs a sequence of chained GETNEXT requests automatically to query a network entity for a subtree of management values.

[3] https://help.ubuntu.com/12.04/installation-guide/index.html
[4] https://www.debian.org/releases/stable/installmanual
[5] http://pysnmp.sourceforge.net/index.html

3.2.3 Problem analysis

Reproduction of the mentioned issue can be achieved by running the bin/conpot_cloner script against Conpot version 0.3.1. The next part will show the recreation process as well as the analysis of the issue.

First we start up Conpot as root with the default template:

root@ubuntu-VirtualBox:/home/ubuntu/conpot# conpot -t default

After Conpot initializes and starts the logging procedure, we run the cloner module against it:

ubuntu@ubuntu-VirtualBox:/opt/conpot/bin$ python2.7 conpot_cloner

Debugging conpot_cloner shows that after calling the cloner.snmp_walk on line 132 the execution is continued in the definition of snmp_walk, that eventually results in forced termination of the cloner.

```
129    if __name__ == "__main__":
130        cloner = ConpotCloner()
131        #cloner.modbus_worker()
132        cloner.snmp_walk()
```

After stepping into the method an instance of the client is created successfully. However, the execution fails on line 126 at evaluating attribute self.result.

```
122        def snmp_walk(self):
123            client = snmp_client.SNMPClient(host='127.0.0.1', port=161)
124            OID = ((1, 3, 6, 1, 2, 1, 25, 2, 3, 1, 1), None)
125            client.walk_command(OID, callback=self.mock_callback)
126            print self.result
```

The error message indicates that ConpotCloner object's result attribute does not exist, even though it is supposed to be created in the mock_callback method.:

```
ubuntu@ubuntu-VirtualBox:/opt/conpot/bin$ python2.7 conpot_cloner
    Traceback (most recent call last):
        File "conpot_cloner", line 132, in <module>
            cloner.snmp_walk()
        File "conpot_cloner", line 126, in snmp_walk
            print self.result
    AttributeError: 'ConpotCloner' object has no attribute 'result'
```

Upon further investigation it seems the arguments passed to client.walk_command are incorrect. The detailed study of conpot's SNMP client reveals once the correct parameters are passed to the method, the client's behaviour is appropriate. This leads to the conclusion: the error is caused by the conpot_cloner itself not the SNMP module in conpot/tests/helpers/snmp_client.py. Looking at the execution of conpot_cloner shows the mock_callback method is not called. This method is responsible for conveying data between the conpot_cloner and the SNMP module. A callback method is an executable code, which is passed as an argument in another method and which is invoked after some kind of event. The "call back" nature of the argument is that, once its parent method completes, the

function which this argument represents is then called; that is to say that the parent method 'calls back' and executes the method provided as an argument. The invocation may be immediate as in a synchronous callback or it might happen at later time, as in an asynchronous callback. In this case the mock_callback method, which checks for errors and handles the output delivery is left behind by the snmpEngine's dispatcher, which should receive queries and send responses.

By examining examples of implementation of the pySNMP module's API, we can verify, that Conpot is using an older implementation of a now obsolete version, causing the data handling in the mock_callback method implementation improper, resulting in the conpot_cloner's failure.

3.2.4 Implementation of the bugfix

After the analysis above the implementation process can be divided into two separate parts:

1. Fixing the synchronization issue between cloner's client and snmpEngine's dispatcher
2. Updating the mock_callback method, which uses pySNMP's API

The first part can be fixed by starting the dispatcher in the snmp_walk method to wait for mock_callback, and signalling the different states during the execution in method mock_callback. After obtaining the desired information the snmp_walk method should close all running transport objects including the sockets, and remove all callbacks from the dispatcher:

```
145     def snmp_walk(self):
146         client = snmp_client.SNMPClient(host='127.0.0.1', port=161)
147         OID = ((1, 3, 6, 1, 2, 1, 1, 1, 0), None)
148         client.walk_command(OID, callback=self.mock_callback)
149         client.snmpEngine.transportDispatcher.runDispatcher()
150         print self.result
151         client.snmpEngine.transportDispatcher.closeDispatcher()
```

The second part can be achieved by modifying the mock_callback method to fit the current version of the pySNMP API and altering the Object Identifier inside the snmp_walk method to start the retrieval process from the beginning of the MIB. We create cloner's result attribute to hold the information received by reading the given MIB:

```
90     def mock_callback(self, sendRequestHandle, errorIndication, errorStatus,
                                                  errorIndex, varBindTable, cbCtx):
91         if not hasattr(self, 'result'):
92             self.result = ""
```

The new response may containt SNMPv1 noSuchName errors or SNMPv2c exceptions, so we need to handle these along with other error indications.

```
94          if errorIndication:
95              print(errorIndication)
96              return
97          if errorStatus and errorStatus != 2:
98              print('%s at %s' % (
99                   errorStatus.prettyPrint(),
100                  errorIndex and varBindTable[-1][int(errorIndex)-1] or '?'
101                  )
102              )
103              return  # stop the execution on error
```

The next step is to update the handling of the retrieved data according to the new version of pySNMP. The signalization to the snmpEngine dispatcher is realized by returning the states of the execution. We differentiate two states, which represent the continuation of execution and the end of the MIB:

```
104     for varBindRow in varBindTable:
105         # walk command
106         if type(varBindRow) is list:
107             for oid, val in varBindRow:
108                 if isinstance(val, EndOfMibView):
109                     self.result+='%s\n' % (val.prettyPrint())
110                     return 0 # signal dispatcher about the last element of MIB view
111                 self.result+='%s = %s\n' % (oid.prettyPrint(), val.prettyPrint())
112         # get command
113         elif type(varBindRow) is tuple:
114             oid, val = varBindRow
115             if isinstance(val, EndOfMibView):
116                 self.result+='%s\n' % (val.prettyPrint())
117                 return 0 # signal dispatcher about the last element of MIB view
118             self.result+='%s = %s\n' % (oid.prettyPrint(), val.prettyPrint())
119     return 1 # signal dispatcher to continue
```

We have to separate the cases where the invocation of the callback function is initiated by the snmp_get or snmp_walk commands. The reason behind this separation is, that the pySNMP library returns a different data structure according to the invoked command. In case of a snmp_walk command, we have to walk through all the rows in the given data structure called varBindTable. This table contains a sequence of varBinds. Each varBind of varBinds in the varBindTable represents a set of Managed Object Names and Managed Object Values. As we walk through this data structure we store these names and values after proper formatting. In case of calling the snmp_get command, pySNMP returns a similar varBindTable, yet it only holds a single python tuple containing information about one element of an MIB. The data retrieval is resolved in a similar way, just like in the case of a snmp_walk command call. This separation is necessary to ensure we do not cause other issues by overhauling the conpot_cloner module.

3.2.5 Evaluating the achieved results

After implementing the changes mentioned in the previous subsection, the conpot_cloner module is functional and its execution results in expected behavior. Hence before the implementation of the bug fix the conpot_cloner couldn't provide results the only reference we can use for evaluation is other implementation of the same concept. In this section we compare the results achieved by conpot_cloner to the net-snmp's one.

First we run the updated conpot_cloner against Conpot. Next we invoke net-snmp's snmpwalk command to see if there are any deviation. Comparing the program outputs we look for unwanted behavior, which should appear as different representation of the same MIB. The testing will be considered successful if the structure of the printed out MIB is same at both.

Running conpot_cloner against Conpot provides the following results:

```
ubuntu@ubuntu-VirtualBox:/opt/conpot/bin$ python2.7 conpot_cloner
1.3.6.1.2.1.1.1.0 =Siemens, SIMATIC, S7-200
1.3.6.1.2.1.1.2.0 = 1.3.6.1.4.1.20408
1.3.6.1.2.1.1.3.0 = 114
1.3.6.1.2.1.1.4.0 = Siemens AG
1.3.6.1.2.1.1.5.0 = CP 443-1 EX40
1.3.6.1.2.1.1.6.0 = Venus
1.3.6.1.2.1.1.7.0 = 72
...
1.3.6.1.2.1.11.30.0 = 1
1.3.6.1.2.1.11.31.0 = 0
1.3.6.1.2.1.11.32.0 = 0
No more variables left in this MIB View
```

As seen above the updated conpot_cloner prints out an MIB with the OIDs pairing its assigned values. The net-snmp's snmpwalk command's result should provide the same MIB structure, though the result may differ in the formatting. We invoke the snmpwalk command while running an instance of Conpot in the background:

```
ubuntu@ubuntu-VirtualBox:/opt/conpot/bin$ snmpwalk -Os -v 1 -c public localhost
iso.3.6.1.2.1.1.1.0 = STRING: "Siemens, SIMATIC, S7-200"
iso.3.6.1.2.1.1.2.0 = OID: iso.3.6.1.4.1.20408
iso.3.6.1.2.1.1.3.0 = Timeticks: (182) 0:00:01.82
iso.3.6.1.2.1.1.4.0 = STRING: "Siemens AG"
iso.3.6.1.2.1.1.5.0 = STRING: "CP 443-1 EX40"
iso.3.6.1.2.1.1.6.0 = STRING: "Venus"
iso.3.6.1.2.1.1.7.0 = INTEGER: 72
...
iso.3.6.1.2.1.11.30.0 = INTEGER: 1
iso.3.6.1.2.1.11.31.0 = Counter32: 0
iso.3.6.1.2.1.11.32.0 = Counter32: 0
End of MIB
```

Comparing the output of the two programs, shows the same MIB structur, indicating the updated conpot_cloner is capable to cycle through the whole MIB tree and convey the retrieved information to the user.

Conpot is logging requests from conpot_cloner, as well as from snmpwalk. The log file created by the honeypot shows that the logging procedure was successful. The log file shows the same results in both cases:

```
2014-11-22 23:54:17,014 New snmp session from 127.0.0.1 (be99c159-e880-4915-b4a0-3cb58778f47b)
2014-11-22 23:54:17,014 SNMPv3 GetNext request from ('127.0.0.1', 46038): 1.3.6.1.2.1.1.1.0
2014-11-22 23:54:17,015 SNMPv3 response to ('127.0.0.1', 46038): 1.3.6.1.2.1.1.2.0 1.3.6.1.4.1.20408
2014-11-22 23:54:17,018 DataBus: Get value from key: [Uptime]
...
2014-11-22 23:54:19,205 SNMPv3 response to ('127.0.0.1', 46038): 1.3.6.1.2.1.11.32.0 0
2014-11-22 23:54:19,308 SNMPv3 GetNext request from ('127.0.0.1', 46038): 1.3.6.1.2.1.11.32.0
2014-11-22 23:54:19,309 SNMPv3 response to ('127.0.0.1', 46038): 1.3.6.1.2.1.11.32.0
```

We repeat this testing process with different kind of MIBs to prove that the implementation of the bug fix was successful. In each test case the conpot_cloner's and the snmpwalk's output resulted in the MIB structure.

3.3 Adding support to change the MAC address

This section will explain the implementation of a separate module, which provides additional functionality to Conpot – the ability to change the MAC address on a given interface.

3.3.1 MAC address

A media access control address is a globally unique identifier assigned to network interfaces for communications on the physical network segment, often referred as hardware or physical address. It is hardwired or hard-coded onto a computer's network interface card and it is unique. MAC addresses are 6-byte (48-bits) long, the first three octets of the address identify the organization in the Organizationally Unique Identifier. The following octets are serial number assigned by the manufacturer.

In terms of a honeypot the MAC address sometimes might give hints to the attackers that the system is running on top of a virtualization software. It does not declare that the system is actually a honeypot, but it can give doubts to an aggressor. The reasoning behind that, if the attacker is on the same network segment as the honeypot, there are possible ways to fingerprint the honeypot at layer 2. By looking at

ARP responses or trying to reach remote NetBIOS services the attacker will be able to get the MAC address and guess that the system is a virtualized guest, by examining the OUI part of the MAC address and searching for a match in the IEEE standards. This way the attacker can identify the assigned MAC address to an organization, which may be a provider of virtualization software. To avoid this kind of fingerprinting where the virtualization software does not support the configuration of the MAC address it is necessary to change the hardware address other way. To achieve this protection in Conpot we implemented the feature of spoofing a MAC address.

3.3.2 Problem analysis

While the physical MAC address is a manufacturer-assigned hardware address, it can be modified by the user via several mechanisms that allow modification of the MAC address reported by the operating system.

Changing the MAC address can be achieved in numerous ways including both hardware and software-based solutions. Since a hardware-based solution is not utilized in this scenario, we are limited to software-based solutions. The appropriate answer to this problem is dependent on the operating system installed on the computer running Conpot. Hence Conpot's development is focused mainly on Linux distributions, specifically Ubuntu and Debian; the newly implemented module will be based around these distributions as well.

The methods used to spoof the MAC address can affect the persistency of the change. Permanent changes require to edit configuration files, which can cause unnecessary problems later on, and wouldn't provide any advantages over non-permanent changes in this case. Taking this into consideration, we are looking for a solution providing non-permanent changes. The next upcoming issue concerns the user's privileges. There are number of techniques for changing the MAC address of a computer with or without root privileges on UNIX based systems. Without root privileges, this can be achieved with ptrace system call or LD_PRELOAD. LD_PRELOAD allows to override the value returned by a library call. We could use this feature to implement the module to act as a wrapper around an application and load a small library that overloads the ioctl() function. The MAC address of an interface can be obtained using the ioctl command SIOCGIFHWADDR. When called, we alter the value returned by the system call with the wanted MAC address' value. The solution could be achieved in a similar manner with PTRACE_SYSCALL to alter the behavior of ioctl(). However, these options would be unnecessarily complicated implementations to the solution, just to bypass the need for root privileges. It is possible to run Conpot as root without causing further issues, since the implementation of Conpot forces the running instance of Conpot to drop the root privileges after they are not needed. Using this ability of Conpot we can implement an easier and more effective realization of the module

to provide an answer for MAC address changing via ifconfig, a system administration utility in Unix-like systems supported by both Ubuntu and Debian.

3.3.3 Implementation of the module

This part is about the implementation of the extension module and importing the newly created module into Conpot.

First we want to create a module that is capable to change the MAC address on an interface either on Ubuntu or Debian distributions. To achieve this the ifconfig utility has been chosen, for the reasons explained above in this particular problem's description. For the ifconfig utility we need to provide an interface whose address we want to change as well as the new MAC address, which will be spoofing the original:

ifconfig [-v] interface [aftype] options | address …

However, the ifconfig utility for these changes requires to have root privileges. After the script starts the execution a test is needed to check if the necessary privileges are provided:

```
14      def is_root():
15              return os.getuid() & os.getgid() == 0
```

The superuser account on Linux systems always has a UID and GID equals zero as this is coded into the kernel of the OS.

If the conditions are met, and the utility can be run with the needed privileges, then the next step is to invoke a set of ifconfig commands which will execute the changes in the operating system. This is done using python's subprocess module, which allows to spawn new processes, and manage their IO.
The check_call function used in this implementation waits for the ifconfig commands to complete and obtains their return codes.

```
17      def set_hwaddr(device,mac):
18          subprocess.check_call(["ifconfig", "%s" % device, "down"])
19          subprocess.check_call(["ifconfig", "%s" % device, "hw", "ether","%s" % mac])
20          subprocess.check_call(["ifconfig", "%s" % device, "up"])
```

First the desired interface has to be shut down, then a new MAC address has to be specified. After that we need to activate the modified interface. After executing this method a temporary change of MAC address should been established, although it will revert to the default on the next system reboot. However, this is not an issue in this case, since it will be run at Conpot's startup procedure every time needed. In case the attempt to change the MAC address is a failure the module simply reverts to the original MAC address by storing the original address during the startup procedure and in case of a caught exception invoking a second set_hwaddr() function with the original hardware address as parameters.

To check if the modification was successful, we need to compare the possibly modified interface's MAC address to the desired MAC address. We have to call the ifconfig utility again on the interface, and parse through the output to find the actual MAC address on the active interface. In case that they are the same, the modification was successful:

```
22      def get_hwaddr(device):
23              output = subprocess.Popen(["ifconfig", "%s" % device],
                                stdout=subprocess.PIPE).communicate()[0]
24              index = output.find('HWaddr') + len('HWaddr ')
25              localAddr = output[index:index+17].lower()
26              return localAddr
```

Once this is done, the module is functional for changing the MAC address on a given interface. The next step is to import the module into Conpot and provide some logging information to the user. Conpot's configuration file located in conpot/ now has to contain the settings for the module, such as the desired interface and MAC address. In case the activation of this module is not needed it can be deactivated in the configuration file as well:

```
[change_mac]
enabled = True
interface = eth0
hwaddr = 00:c0:9f:a1:9d:4a
```

The newly created python script is considered as a utility to Conpot and its location has been stated in conpot/utils/ as set_mac.py. The module has to be imported into the main script and started there at every execution to be functional:

```
49              from conpot.utils import set_mac
...
223             # Get MAC address from config file
224             MAC_address = None
225             if config.getboolean('change_mac', 'enabled'):
226                     MAC_address = set_mac.set_mac(config)
```

After merging the newly created utility into the main honeypot the implementation of the feature to change the MAC address on a desired interface is finished. This subsection discussed various methods to provide ability to Conpot to change the MAC address and showed the implementation process of a specific module along with importation procedure of the separate utility into Conpot.

3.3.4 Testing and error handling

This subsection will illustrate few of the possible outcomes using the newly acquired utility with Conpot. The important lines will be highlighted for better readability. The testing will be provided by comparing the MAC address of the development VM during different scenarios, but always in two states: before and after the startup procedure of Conpot.

First we look up the virtual machine adapter's MAC address using ifconfig:

```
ubuntu@ubuntu-VirtualBox:~$ ifconfig eth0
eth0          Link encap:Ethernet  HWaddr 08:00:27:2d:44:05
              inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
              inet6 addr: fe80::a00:27ff:fe2d:4405/64 Scope:Link
              ...
```

Next we run Conpot with different configuration file setups, which will contain both valid and invalid MAC addresses and we will compare the MAC addresses before and after each start up.

Running Conpot without superuser privileges causes Conpot to log the specific error and terminate the execution without any changes made showing no deviation from the expected behavior:

```
ubuntu@ubuntu-VirtualBox:/usr/local/lib/python2.7/dist-packages/Conpot-0.3.1-
py2.7.egg/bin$ python2.7 conpot -t default
...
2015-01-19 19:13:18,441 Could not set MAC address: None missing root priviliges
```

After this we run Conpot as a superuser with proper settings in the configuration file:

```
ubuntu@ubuntu-VirtualBox:/usr/local/lib/python2.7/dist-packages/Conpot-0.3.1-
py2.7.egg/bin$ sudo python2.7 conpot -t default
[sudo] password for ubuntu:
...
2015-01-19 19:22:29,740 MAC address changed to 00:0c:6e:a2:5f:ab on interface eth0
```

To ensure that changes were actually made we check the hardware address using the ifconfig utility. The address of the VM should correspond to the change reported by Conpot:

```
ubuntu@ubuntu-VirtualBox:~$ ifconfig eth0
eth0          Link encap:Ethernet  HWaddr 00:0c:6e:a2:5f:ab
              inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
              inet6 addr: fe80::20c:6eff:fea2:5fab/64 Scope:Link
              ...
```

In case the configuration file contains an invalid MAC address or a non-existent interface Conpot logs the errors and reverts to the original hardware address before termination:

```
...
SIOCSIFHWADDR: Cannot assign requested address
2015-01-19 19:37:54,222 Unable to set MAC address 01:0c:6e:a2:5f:ab on interface eth0

...
eth1: ERROR while getting interface flags: No such device
2015-01-19 19:36:51,216 Unable to set MAC address 00:0c:6e:a2:5f:ab on interface eth1
```

This issue cannot be resolved if the implementation uses the ifconfig utility. The only option we have is either terminate the program with proper exit codes as seen above or continue the execution without any changes made. We decided to go with the first option to ensure the honeypot we are running definitely works as it is configured, giving less attack surface to intruders.

3.4 Separating STIX and TAXII configuration

3.4.1 Problem analysis

The issue with Conpot's configuration is, that both STIX and TAXII related settings are defined under the same section. The same problem occurs in several modules of Conpot related to either STIX or TAXII. This problem is mainly about locating files and parts of the source code using the specified section of the configuration file and modifying the scripts to be able to get the necessary information for proper functioning. After the alteration of the modules, Conpot should not show any kind of deviation in behavior.

In the following subchapters we will briefly explain STIX and TAXII. Next we will discuss the modifications made in Conpot along with the testing of this fix.

3.4.2 STIX and TAXII

STIX is a standardized language to represent structured cyber threat information. The STIX language intends to convey a full range of potential cyber threat information. This feature allows to display event sessions captured by Conpot in a structured format, which eases the integration of the honeypot into existing consumer infrastructures.

TAXII defines a set of services and message exchanges that, when implemented, enable sharing of actionable cyber threat information across organization boundaries. With the implemented TAXII client, Conpot is capable to transport the collected data in real time to the interested parties.

3.4.3 Locating the issue

Excluding the configuration file of Conpot, we searching through the contents of the scripts and filter the modules using the problematic section:

```
ubuntu@ubuntu-VirtualBox:~/development/conpot$ grep --include=\*.py -rwn . -e "taxii"
./conpot/tests/test_taxii.py:41:                ...
./conpot/core/loggers/taxii_log.py:32:          ...
./conpot/core/loggers/log_worker.py:84:         ...
./conpot/core/loggers/stix_transform.py:50:     ...
```

Inspecting the source code of these files show that log_worker.py is responsible for the creation and configuration of the individual logging modules. In this case the TaxiiLogger class, which is located in taxi_log.py. The TaxiiLogger is a designated logger module that converts the events logged by Conpot to a STIX compatible XML format, wraps the message in a TAXII envelope and calls the TAXII web

service to transmit the data. The TaxiiLogger reads the data necessary for the transmission from the configuration file:

```
31      def __init__(self, config, dom):
32              self.host = config.get('taxii', 'host')
34              self.port = config.getint('taxii', 'port')
35              self.inbox_path = config.get('taxii', 'inbox_path')
36              self.use_https = config.getboolean('taxii', 'use_https')
```

Observing the source code of stix_transform.py reveals, that the remaining settings under the "taxii" section belong to the StixTransformer. The StixTransformer class is responsible for the appropriate conversion to STIX XML in the TaxiiLogger and other modules as well. The retrieval of configuration data is implemented in a similar way to the TaxiiLogger's solution:

```
49      def __init__(self, config, dom):
50              self.config = config._sections['taxii']
...
88      def transform(self, event):
89              self._set_namespace(self.config['contact_domain'],
                                    self.config['contact_name'])
```

Using this information we can separate the settings of TAXII from the STIX in the configuration file.

```
[taxii]
enabled = False
host = taxiitest.mitre.org
port = 80
inbox_path = /services/inbox/default/
use_https = False
contact_name = conpot
contact_domain = http://conpot.org/stix-1
```

```
[taxii]
enabled = False
host = taxiitest.mitre.org
port = 80
inbox_path = /services/inbox/default/
use_https = False

[stix]
enabled = False
contact_name = conpot
contact_domain = http://conpot.org/stix-1
```

Simply by segregating the data into two different sections and modifying the function parameters in source codes we can resolve this issue. The new separated sections should improve the readability of the configuration file and the source code as well.

One of the testing suites also uses these configuration settings and need to be changed. The file can be located in conpot/tests/ directory under the name of test_taxii.py. This script contains the tests for transmitting data to a TAXII server and validating a STIX xml. Further inspection of this file shows that the test cases this script contains should be separated into two files, because the tests cover different modules of Conpot. Creating a new test suite for the STIX tests obtained from test_taxii.py, and modifying the necessary parameters in the invocation of configuration file parsing methods should solve the issue of separation.

The testing of the modifications is accomplished by running the test_taxii.py and test_stix.py suites, which cover the functionality of all the altered modules. The tests show no deviation in behavior, hence the implementation of the changes are considered as a success.

3.5 UID handling in the MODBUS protocol

The subsection will focus on the improvement of the MODBUS protocol implementation within Conpot, especially the MODBUS server's slave addressing and response handling according to the MODBUS's description. First we will explain the terms necessary to understand the problem with Conpot's MODBUS protocol implementation.

3.5.1 MODBUS protocol

MODBUS protocol is a messaging structure developed by Modicon in 1979, used to establish master-slave/client-server communication between programmable logic controllers (PLCs). MODBUS is typically used to transmit signals from instrumentation and control devices to a main controller or data gathering system[6].

Each device intended to communicate using MODBUS is given a unique address. There are several versions of the MODBUS protocol, which cause differences in the behavior of the devices. In serial and MODBUS+ networks, only the node assigned as the Master may initiate a command. On Ethernet, any device can send out a MODBUS command, although usually only one master device does so. Aside from the differences mentioned above, there are a few alterations in the message content as well. A MODBUS command contains the MODBUS address of the device it is intended for. Only the intended device will act on the command, even though other devices might receive it as well. Basic MODBUS commands can instruct a Remote Terminal Unit (RTU) to execute actions, such as read or write values in its registers or control the I/O ports [5]. Remote terminal units are microprocessor-controlled electronic devices, which connect to sensors in the process and convert the signals to digital data and transmit it to the supervisory system. Moreover it is capable to receive commands from the supervisory system and evaluate Boolean logic operations. An RTU message usually contains a Slave Identification Address, a Function Code along with the Data and the CRC for error checking. To transmit this message with a TCP/IP wrapper there are necessary alterations in the message structure. The new message has to contain an additional MODBUS Application Header (MBAP) and the Slave Identification has to be changed to a Unit Identifier. The Unit Identifier is used for intra-system routing, as it is identifies the MODBUS slave address of a remote device connected on a MODBUS+ or MODBUS serial line sub-

[6] http://www.simplymodbus.ca/faq.htm

network. The UID is set by the client in the request and the same value has to be returned by the server in its response.

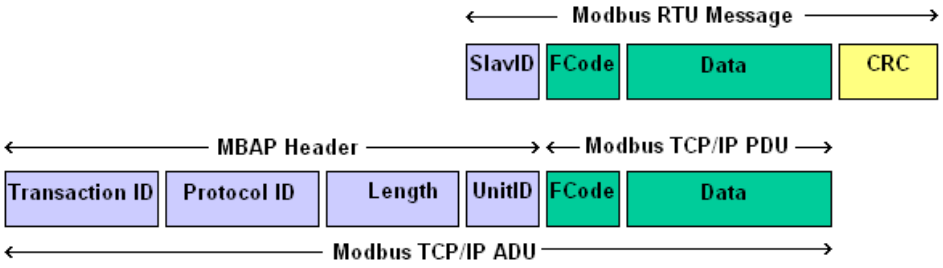


Figure 7: Image MODBUS message structure Source: (http://www.simplymodbus.ca/TCP.htm)

MODBUS messaging on TCP/IP provides a client/server communication between the connected devices with real time information exchange. The client-server model is based on four types of messages:

- MODBUS Request
- MODBUS Confirmation
- MODBUS Indication
- MODBUS Response

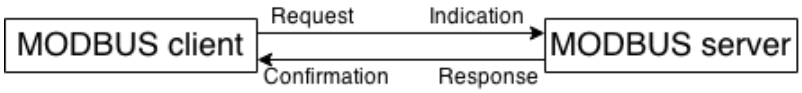


Figure 8: MODBUS client-server communication

As illustrated in the picture above the participant devices can act in two roles:

- The MODBUS client allows the user application to control information exchange with a remote device. The client builds a MODBUS request sends it to the server and waits for a MODBUS confirmation.
- The MODBUS server on reception of a request proceed to take proper actions, once it is done the server builds a response depending on device context and sends it back to the clients.

The forthcoming sections will analyze the issue with Conpot's MODBUS server's UID handling and explain the solution for improved UID management.

3.5.2 Problem analysis

The problem with Conpot's MODBUS implementation's response handling is, it does not fulfill the MODBUS messaging protocol's requirements. The current version of Conpot (v0.3.1) responds to any MODBUS slave address, while as stated in Modbus Messaging Implementation Guide there should be different response handling according to the client-server/master-slave connection:

1. "If the MODBUS server is connected to a MODBUS+ or MODBUS Serial Line sub-network and addressed through a bridge or a gateway, the MODBUS Unit identifier is necessary to

identify the slave device connected on the subnetwork behind the bridge or the gateway. The destination IP address identifies the bridge itself and the bridge uses the MODBUS Unit identifier to forward the request to the right slave device. The MODBUS slave device addresses on serial line are assigned from 1 to 247 (decimal). Address 0 is used as broadcast address [5]."

2. "When addressing a MODBUS server connected directly to a TCP/IP network, it's recommended not using a significant MODBUS slave address in the "Unit Identifier" field. In the event of a re-allocation of the IP addresses within an automated system and if an IP address previously assigned to a MODBUS server is then assigned to a gateway, using a significant slave address may cause trouble because of a bad routing by the gateway. Using a nonsignificant slave address, the gateway will simply discard the MODBUS PDU with no trouble. 0xFF is recommended for the "Unit Identifier" as nonsignificant value. The value 0 is also accepted to communicate directly to a MODBUS/TCP device [5]."

This Guide shows that we have to differentiate the two modes of emulation in Conpot's MODBUS server. The mode of the connection has to be stored in an accessible file, like Conpot's configuration file or in the template file of the MODBUS server, so the user can easily change the emulation's behavior. Furthermore we have to modify Conpot's template file defining the emulated MODBUS slaves. The information regarding to the slaves are stored in the modbus.xml file located in conpot/templates/default/modbus/ directory. To alter the mentioned XML file we need to modify the according XSD file containing the description of this XML document. The XML Schema Definition can be found in the protocol's implementation directory: conpot/protocols/modbus/. As described in the Guide above, on TCP/IP connection the addressing of devices is managed by using IP addresses, therefore the MODBUS Unit Identifier is useless. The value 0x00 or 0xFF has to be used. On a serial line the addressing is accomplished using slave UIDs. The assigned UIDs have to be between 0 and 247, while UID=0 is specified as a broadcast address. We have to create new slaves to represent both types of connection. The newly created slaves' identifiers have to be in accordance with the former statements. Conpot's documentation describes the customization of the default profile containing MODBUS's template as well, for more information see Appendix A. Once the new slaves are defined, we need to add them to the general template file containing information about Conpot's default profile along with the key value mappings for the databus module. The databus serves as a global data storage and retrieval module within Conpot. Alteration of the mentioned template ensures that the defined slaves are created and initialized at the honeypot's start up procedure. The next part of the problem is the proper handling of these slaves according to their UID.

In accordance with the preset connection settings in the MODBUS template file we have to filter the messages and convey them to the proper UIDs. We shouldn't be able to address a slave with a UID that is nonexistent within the sub-network. First we have to acquire the connection settings and check for

its validity. The next problem to resolve is to filter the messages requesting invalid addresses. After filtering the received MODBUS requests through several conditional expressions, further modification is needed to handle broadcast communication to the slaves. As stated in [6] the mode of the communication with the slaves is highly dependent on the UID addressing:

"The master node issues a MODBUS request to the slave nodes in two modes:

1. In unicast mode, the master addresses an individual slave. After receiving and processing the request, the slave returns a message (a 'reply') to the master.

In that mode, a MODBUS transaction consists of 2 messages: a request from the master, and a reply from the slave. Each slave must have a unique address (from 1 to 247) so that it can be addressed independently from other nodes.

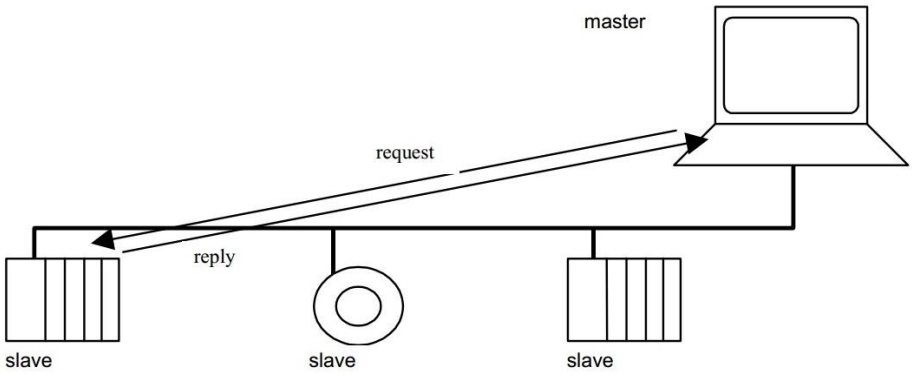


Figure 9: Unicast mode Source [6]

2. In broadcast mode, the master can send a request to all slaves. No response is returned to broadcast requests sent by the master. The broadcast requests are necessarily writing commands. All devices must accept the broadcast for writing function. The address 0 is reserved to identify a broadcast exchange [6]."

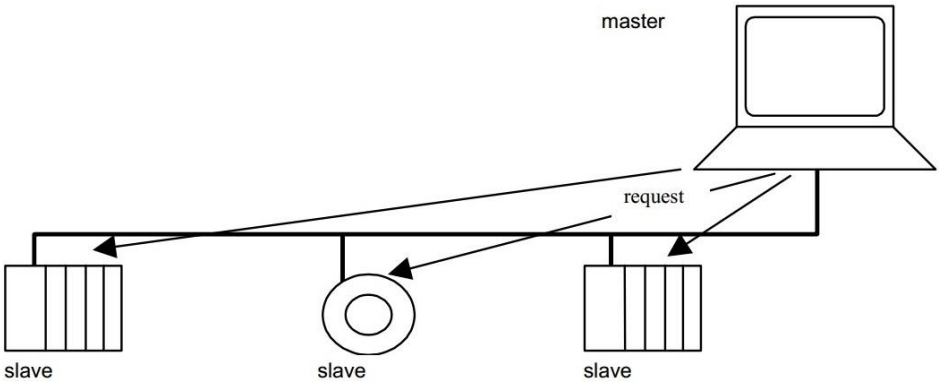


Figure 10: Broadcast mode Source: [6]

Taking this into consideration we have to modify the response message generation as well, since the emulated slaves are responding to every received request. The creation of the proper response type has to be in accordance with the communication mode, which is dependent on the master-slave/client-server connection's type and the device addressed in the MODBUS message. Following a request, there are four possible outcomes from the slave in the MODBUS unicast communication service:

1. The request is successfully processed by the slave and a valid response is sent
2. The request is not received by the slave, therefore no response is sent
3. The request is received by the slave with an error, the slave ignores the request and sends no response
4. The request is received without an error, but cannot be processed further; the slave replies with an exception.

The options above describe the behavior in case of communication over MB serial connection with any slave beside UID=0. The MODBUS broadcast communication service describes the behavior when communicating over MB serial line addressing slave with UID=0, or any MODBUS TCP/IP communication. Since the MODBUS broadcast communication service is essentially unconfirmed, meaning that no response is sent from the slaves, this issue is vastly simplified on the slaves' side. Detailed investigation of the request handling implementation of individual slaves in Conpot shows that the request/response handling is working properly on the slaves' level. However the addressing of the slaves has to be modified in the slave databank to meet the requirements stated above. This databank is responsible for request handling on higher level, slave addition and slave addressing. Once the changes are made Conpot's MODBUS implementation should fulfill the protocol's requirements. To evaluate the achieved result we use a non-commercial tool which is capable to scan PLC devices over s7comm and MODBUS protocols and Conpot's test_modbus_server.py test suite. Conpot's modified configuration simulates a basic Siemens SIMATIC S7-200 PLC with 4 slaves (2 for each type of connection), hence this tool and test suite combined should provide valid results.

The next few sections will outline the implementation process of the significant changes, with more in depth explanation towards the realization of UID filtering.

3.5.3 Modifying the MODBUS implementation

In this section we will provide an in-depth explanation of the modifications discussed in the problem analysis above. The description of the implementation process will follow the steps of the problem resolution in the previous subchapter.

As stated in the problem analysis, we have to create new slaves with variety of UIDs in the MODBUS template. In terms of resolving the MODBUS server's issue the only relevant information regarding the slaves are their identifiers. For this reason the definition of the new slaves will be similar to the slaves described in the original MODBUS template, withx modified slave identifiers and register starting addresses. Minor changes has to be made in Conpot's slave databank implementation to allow adding the new slaves. Only slaves with ID between 1 and 255 can be created, however as stated in the previous subsection above, the address 0 is used for broadcast on serial line connection and as a

nonsignificant value on TCP/IP. After altering the databank, the newly created slaves are added to the databus and generated at the startup procedure.

In the previous subsection we explained how the response generation is in correlation with the connection settings of the server. We need to adjust the modbus_server.py located at conpot/protocols/modbus/ to read the necessary information from the MODBUS template file and pass this information to the function responsible for the addressing in the slave databank. We use the mode of the connection along with the Unit Identifiers received in the requests as conditions to acquire information from the slaves or to inform the server of invalid addressing and ignoring the request. The data retrieval is accomplished by using XPATH expressions. The issue of slave addressing is resolved by applying multiple conditional expressions to check if the server's connection settings and the slaves' IDs in the received MODBUS messages are acceptable according to the protocols specification. Different combination of these values require different actions on behalf of the slaves, however the Modbus_tk library used in Conpot's slave implementation provides solution to handle all relevant scenarios. In case of improper addressing the MODBUS server receives an empty response from the databank. This either means invalid addressing in the MODBUS request or broadcasting over MB serial connection. The only issue remaining is to implement the turnaround delay in MB serial broadcasts and to terminate the pending connections due unconfirmed requests. The former can be resolved by defining a time delay in the template file, acquiring this information and put the MODBUS server's gevent thread into a blocking sleep. This time delay allows the slaves to process the current request before sending a new one. During this time the master is not able to send another request. The problem with the connection termination can be solved by calling shutdown() on the socket to close the underlying connection to deny reading and writing access to the socket descriptor and invoking a close() for deallocating the handler. After implementing these adjustments the server's behavior should fulfill the requirements.

3.5.4 Evaluating the results

In this part we will illustrate the improved UID handling of the honeypot software. For testing purposes we will use the plcscan utility capable to scan PLC devices over s7comm or MODBUS protocols and Conpot's test suite for the MODBUS server implementation.

To test the enhanced MODBUS implementation, we start up a version of Conpot without the implemented improvements and initiate a full network scan using the plcscan tool. We will use the output of plcscan to illustrate the behavior of the original MODBUS implementation. Conpot at default configuration only contains two slaves with UIDs 1 and 2. However by adding more slaves the issue

with UID handling becomes visible, especially if the UIDs are outside the range specified by the protocol. We define new slaves with UIDs 0, 248, 254 and initiate the network scan:

```
ubuntu@ubuntu:/opt/plcscan-read-only$ python plcscan.py --ports=502 --brute-uid
127.0.0.1
Scan start...
127.0.0.1:502 Modbus/TCP
  Unit ID: 0
    Device: Siemens SIMATIC S7-200
  Unit ID: 255
    Device: Siemens SIMATIC S7-200
  Unit ID: 1
    Device: Siemens SIMATIC S7-200
  Unit ID: 2
    Device: Siemens SIMATIC S7-200
  Unit ID: 3
    Device info error: SLAVE DEVICE FAILURE
...
  Unit ID: 247
    Device info error: SLAVE DEVICE FAILURE
  Unit ID: 248
    Device: Siemens SIMATIC S7-200
...
  Unit ID: 253
    Device info error: SLAVE DEVICE FAILURE
  Unit ID: 254
    Device: Siemens SIMATIC S7-200
```

In TCP mode Conpot should reply only on UID 0 and 255. In serial mode the honeypot should limit its responses to maximum UID 247. The results of the scan show that before applying the changes to the honeypot, Conpot's MODBUS server is suffering from a slave addressing issue.

Running the same scan on the improved version of the honeypot should provide results illustrating proper slave addressing, where the communication is limited to UIDs 0 and 255, or UIDs 1-247 depending on Conpot's configuration. First we set Conpot's MODBUS server to TCP mode in the template file, and then execute a scan while running the honeypot:

```
ubuntu@ubuntu:/opt/plcscan-read-only$ python plcscan.py --ports=502 --brute-uid
127.0.0.1
Scan start...
127.0.0.1:502 Modbus/TCP
  Unit ID: 0
    Device: Siemens SIMATIC S7-200
  Unit ID: 255
    Device: Siemens SIMATIC S7-200
```

As seen in the output of the scan above the communication is limited to Unit Identifiers desired in TCP connection mode, meaning that only slaves with UID 0 and 255 respond. We run the scan again, only we change the mode of connection in the server's settings:

```
ubuntu@ubuntu:/opt/plcscan-read-only$ python plcscan.py --ports=502 --brute-uid
127.0.0.1
Scan start...
127.0.0.1:502 Modbus/Unknown Protocol
  Unit ID: 1
    Device: Siemens SIMATIC S7-200
  Unit ID: 2
    Device: Siemens SIMATIC S7-200
  Unit ID: 3
    Device info error: SLAVE DEVICE FAILURE
...
  Unit ID: 247
    Device info error: SLAVE DEVICE FAILURE
```

The results show no difference from the desired values described in the MODBUS implementation guide. Plcscan sends a MODBUS Read Device Identification request, which is non-broadcastable. This means that slave with identifier of 0 should not respond, only slaves within range of UIDs between 1 and 247. With the changes made Conpot logs the traffic information when the scan tries to map the network:

```
ubuntu@ubuntu:/usr/local/lib/python2.7/dist-packages/Conpot-0.4.0-py2.7.egg/bin#
python conpot –t default
...
2015-05-08 08:22:59,896 New modbus session from 127.0.0.1 (4fe0c252-4cfe-44c5-a737-
863e18a1f56c)
2015-05-08 08:22:59,896 New connection from 127.0.0.1:43392. (4fe0c252-4cfe-44c5-a737-
863e18a1f56c)
2015-05-08 08:22:59,896 Client disconnected. (4fe0c252-4cfe-44c5-a737-863e18a1f56c)
2015-05-08 08:22:59,896 New connection from 127.0.0.1:43393. (4fe0c252-4cfe-44c5-a737-
863e18a1f56c)
2015-05-08 08:22:59,896 Function 43 can not be broadcasted
2015-05-08 08:22:59,897 Connection terminated with client 127.0.0.1.
2015-05-08 08:22:59,897 New connection from 127.0.0.1:43394. (4fe0c252-4cfe-44c5-a737-
863e18a1f56c)
2015-05-08 08:22:59,897 Client ignored due to invalid addressing. (4fe0c252-4cfe-44c5-
a737-863e18a1f56c)
2015-05-08 08:22:59,898 New connection from 127.0.0.1:43395. (4fe0c252-4cfe-44c5-a737-
863e18a1f56c)
2015-05-08 08:22:59,898 Modbus response sent to 127.0.0.1
2015-05-08 08:22:59,898 Client disconnected. (4fe0c252-4cfe-44c5-a737-863e18a1f56c)
2015-05-08 08:22:59,898 New connection from 127.0.0.1:43396. (4fe0c252-4cfe-44c5-a737-
863e18a1f56c)
...
```

Further testing is required to ensure that the addressing is functional in scenarios different from Read Device Identifications requests (MODBUS function code 43). The modified template defines four types of slave blocks: coils, discrete inputs, analog inputs and holding registers. Conpot's test suite initiates to read information from these slave blocks with a specified MODBUS read message for each type. Furthermore the test suite also covers the case of broadcasting over MB serial line represented as specific MODBUS write requests. The improved MODBUS implementation should respond to requests according to [6] and ignore any kind of invalid messages. The logging procedure of attack events is necessary in both cases. We start up Conpot and run the test script against it. The script uses the features of the unittest library. After establishing the connection with the server, it sends a specific type of

MODBUS read request to a specified slave. Once a response is obtained, it checks for deviation using assertions. During the testing process, we alter the type and the addressing of the MODBUS request to increase the code coverage of the testing suite. The results of the tests are in accordance with the expected values specified in the template file. Further testing shows that requesting non-existent blocks or initiating invalid commands are handled either on the individual slave level or by the Modbus-tk library. Since these features are working properly, we will not include these cases in this paper. Finally we request information from an existing slave block only the mode of connection and the addressing of the slave is set not to fulfill the MODBUS specification protocol's expectations. Conpot ignores the request's command, logs the intrusion and terminates the connection with the client.

Analyzing the results, we can consider the implementation of the improved UID handling of Conpot's MODBUS implementation a success. Before the improvements the improper slave handling did not meet the requirements of the MODBUS protocol causing higher probability of fingerprinting the honeypot during an attack and unwanted behavior on the user side.

3.6 Implementing the BACnet protocol

In the following subchapters we will learn the basics of BACnet communication necessary to implement this protocol. We explain the implementation of this protocol with samples from the source code along with the evaluation of the achieved results.

3.6.1 BACnet protocol

BACnet is a building automation and control networking protocol, which was designed to meet the communication needs of devices and their associated equipment in building automation systems. A building automation system (BAS) encompasses a highly complex network of control systems. This ASHRAE, ANSI, and ISO standard protocol is designed to facilitate communication of building automation and control systems for applications such as heating, ventilating, air-conditioning control, lighting control, access control, and fire detection systems. The BACnet protocol utilizes the infrastructure for the various BASs to exchange in a way to maximize the efficiency of the connected members. BASs are deployed across all levels ranging from small building segments to larger building establishments. A typical building automation network of devices includes a primary and secondary bus that is connected to various nodes in the system.

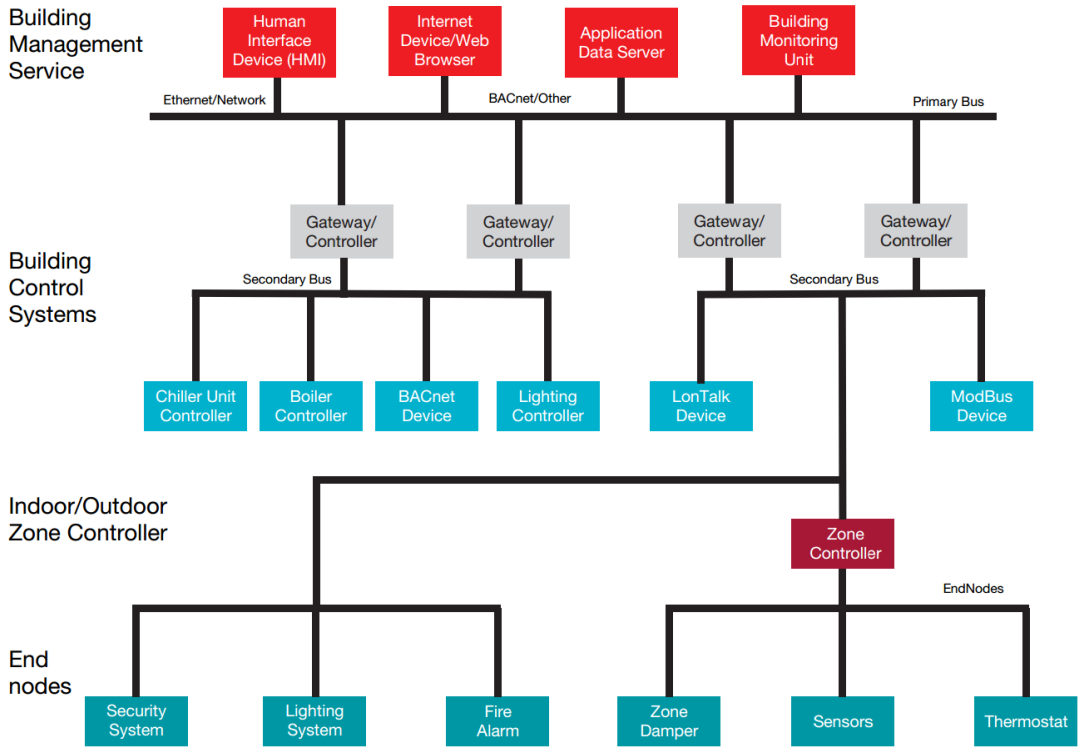


Figure 11: BACnet topology Source: [8]

The BACnet protocol specifies a way to transmit messages using wired or wireless standard protocols over data link and physical layers, such as Ethernet, RS-485, RS-232, ARCNET, LonTalk, UDP/IP, and HTTP. The most common serial version is called BACnet MS/TP, while the dominant Ethernet-based version is BACnet/IP. The BACnet/IP version has been developed to allow the BACnet protocol to use TCP/IP networks to enable system owners, facility managers, or even external suppliers to access BACnet networks and manage the devices and systems remotely.

BACnet has a layered protocol architecture based on a subset of ISO Open Systems Interconnections (OSI) Basic Reference Model. The Presentation, Session and Transport layers' functionality are either implemented in the BACnet Application layer, or completely eliminated in case if they're not needed for the BACnet application. BACnet/IP encapsulates application and network layer messages within a BACnet virtual link layer (BVLL) encapsulated in UDP/IP frames allowing to be used within IP based infrastructures. We will be focusing on the implementation of the BACnet/IP version, since it is the dominant Ethernet based version and provides the fastest transport of messages.

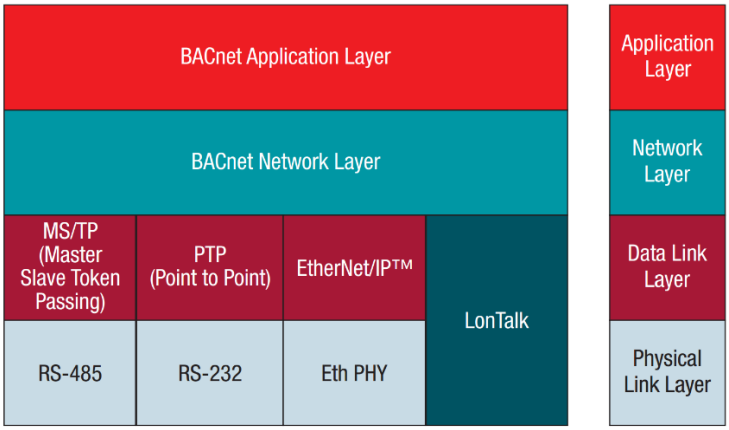


Figure 12: BACnet layer stack Source: [8]

The communication between BACnet devices is represented as an exchange of abstract service primitives, which are typically commands issued by BACnet devices for reading and manipulating information at the application layer. The BACnet protocol defines four service primitives:

- Request
- Indication
- Confirmation
- Response

The contained information is transmitted in the form of Protocol Data Units (PDUs), whose structure is defined by the type of the specified service. The standard services described in the BACnet protocol can be categorized into several groups:

- Confirmed requests are unicast messages and expected to be acknowledged,
- Unconfirmed requests are broadcast messages and does not require an acknowledgment,
- Acknowledgements,
- Error PDUs are used to indicate the reason why a previously confirmed service request failed,
- Reject PDUs are sent to deny received confirmed request PDUs based on syntactical flaws or other protocol errors that prevent the PDU from being interpreted or the requested service from being provided.

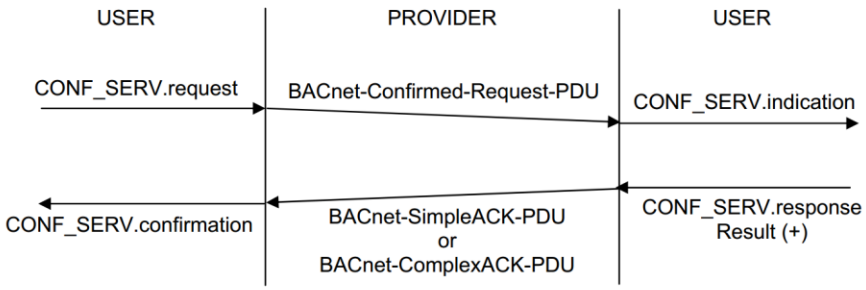


Figure 13: Time diagram for a confirmed request service Source: [7]

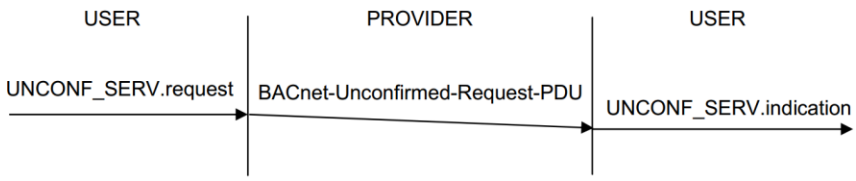


Figure 14: Time diagram for an unconfirmed request service Source [7]

Unconfirmed services like Who-Is, I-Am, Who-Has, I-Have are used for Device and Object discovery within the network. Confirmed services such as Read-Property and Write-Property are used for data sharing. A BACnet device is not required to support all services except the mandatory Read-Property service, which must be supported by every BACnet device.

The internal design and configuration of a BACnet device is different for each vendor. To resolve issues related to this diversity, the protocol defines a collection of abstract data structures called BACnet Objects. These Objects represent various aspects of the hardware, software of a BACnet device, like physical or virtual information, control algorithms, special application and calculations within the device. The Objects are acted upon by the services described above. This way the user can access information from the device without requiring knowledge of the details of the internal design; the interpretation of the requests is handled by the communication software in the specified BACnet device. Each Object has a type and set of required and optional properties that are used to get information from the Object, or give information to the Object.

We showed the general concept behind the BACnet protocol and explained the requirements to implement a basic device emulation, which is capable to act like a server side BACnet application alongside with additional logging abilities for the honeypot.

3.6.2 Problem analysis

This section will resolve some of the issues during the implementation process of BACnet support, mainly focusing on the theory behind the choices made at the time of realization.

As stated in the previous chapter the BACnet devices communicate using UDP rather than TCP, so the devices do not need to implement a full IP stack. Taking this into consideration the first issue is to implement a datagram server with a BACnet device representing the PLC Conpot simulates. Once the device is created, it requires some configuration, for instance assigning a Device Name, Device Identifier or other communication parameters. Since a BACnet device usually contains a series of BACnet Objects, the next step is to create instance of Objects along with their properties defined by the user, and add these newly created objects to the BACnet device. These configuration data together with the Objects' and their properties' information should be easily accessible and modifiable by the

user, therefore there is a demand for creation of a template for the BACnet protocol, which holds all this data in a structured format. BACnet devices range from motion sensors in lighting control systems, through smoke detectors to programmable thermostats in heating and cooling systems. From this wide variety of devices it can be seen, that some of these devices provide analog output that are dynamically changing during a short period of time. If the emulated device only provides static data, it would cause a vulnerability in the honeypot in terms of fingerprinting. To overcome this issue we need to allow the user to predefine a range of values, which can be randomized to provide more realistic data during the emulation. Finally we have to implement the communication aspect of the server:

1. an "indication" method decoding the received requests, calling the execution of the service, setting a proper response PDU, and logging the valuable information regarding the sender,
2. a "response" method encoding the created response PDU, determining the mode of transmission and in case of demand conveying the message back.

The implemented stand-alone server application should be able to process received requests from other BACnet devices, send proper responses according to the incoming messages, and behave in agreement with the BACnet protocol specification in overall.

The implemented BACnet application will use the BACpypes library to facilitate some parts of the process. This library provides the necessary application layer and network layer for BACnet application development. We will explain the realization procedure in the upcoming chapters following the description above.

3.6.3 Implementing the server side

This section portrays the implementation of a stand-alone BACnet server application, which is capable to handle some specified incoming requests, provide the correct responses while logging valuable information about the aggressor. The integration of the stand-alone application into the honeypot will be described in this chapter as well.

As stated in the previous subsection we need to create the XSD and XML files describing the simulated device including its objects and properties. Starting with the XML Schema Definition we need to store information about the device object and its optional objects. The device object consist of mandatory information about the BACnet device's name and identifier, the maximum acceptable APDU length, the segmentation support during the communication and the vendor's identification number. Furthermore to effectively handle the objects of a device, we incorporate them into an object list containing all the objects of the device for each object defining its compulsory properties. To manage the wide variety of additional properties of an object we are bound to use `<any>` elements inside the schema. This type of element allows the user to extend the XML document with elements not specified

by the XSD. This could cause some issues with the XML validation later on, however it is necessary to define all the properties, which can differ from one object to another. We create the XML template file defining the emulated PLC with additional objects according to the constructed XSD file and fill it with data for testing purposes only, the values given in the template file does not have to correspond to any vendor's BACnet device. Moreover, vendor specific objects and properties are completely ignored during the testing. The reason behind this decision is that the BACpypes library only contains BACnet Objects and Properties described in the BACnet specification. Furthermore we do not have access to any BACnet device during this thesis for detailed analysis, which limits our capabilities in terms of emulating an existing BACnet device.

After all the required information about the device is available we create a datagram server listening on port 47808 using the gevent library. This server greenlet will be merged into Conpot later as a separate lightweight pseudothread starting with the other protocol implementations. During the server initialization process we create an instance of the emulated device and its objects via BACpypes library methods. The objects and properties defined by the user can be validated to a certain degree. During the instantiation of the objects and properties, the BACpypes library provides partial solution to the XML validation problem mentioned above. The validation is resolved by python's object oriented philosophy; transforming the XML object element names into a format of BACpypes.Object class names we will have a list of valid callable Object class names. Using python's built-in getattr() and setattr() functions we can create instances of BACpypes Objects and their properties. In case of invalid object types, property types, value datatypes, etc. the BACpypes methods raise various errors. Using this beneficial property of the library it is sufficient to catch these exceptions and convey the error messages to the user. The remaining issue we have to address is to ensure there is no redundancy in the information belonging to the objects.

The BACnet protocol supports segmentation of messages, hence the application has to ensure to recover the whole requests from the UDP packets in the order they are sent even simultaneously from multiple clients. The issues related to simultaneous data retrieval are resolved by using non-blocking sockets. The correct order of the received data is ensured by the BACnet protocol itself. Once the received data is assembled, we have to decode and categorize the requests. The BACpypes library implements the full BACnet stack and provides different functions that are capable to decode the specific PDU. Since we are implementing the BACnet/IP version of the protocol our only concerns are the Application and Network PDUs.

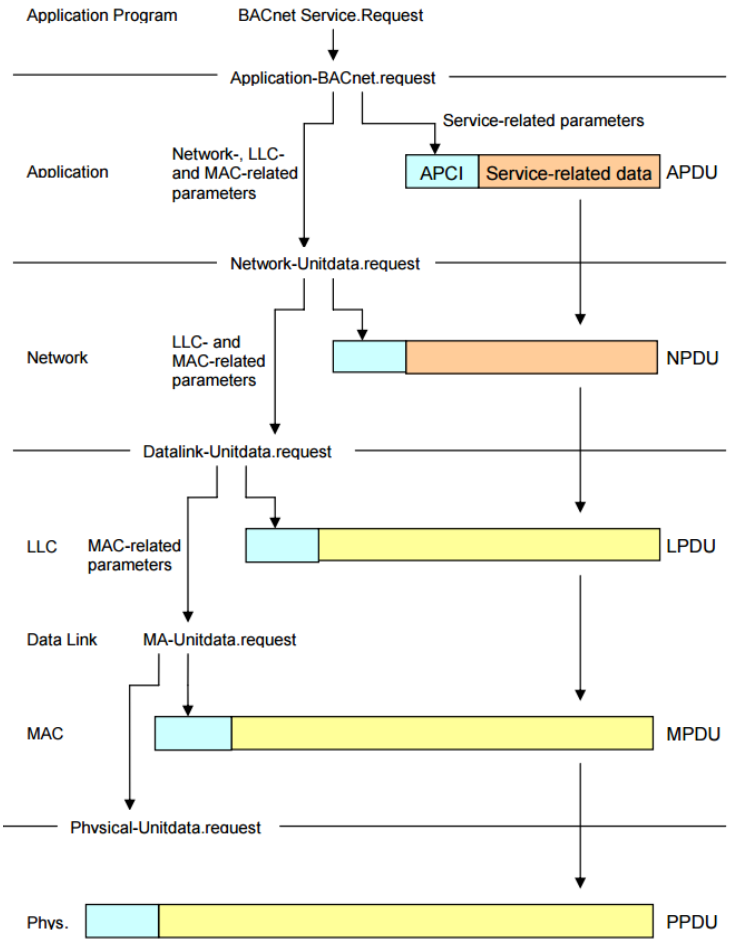


Figure 15: BACnet protocol stack Source: [7]

For providing a proper reply the application has to be able to differentiate the decoded request. An efficient way to classify the incoming traffic is to look at the received request‘s BACnet Protocol Data Unit (PDU) header part for information about the PDU type. According to this information we can determine the invocation of the proper BACpypes functions to decode the APDU request and retrieve the transmitted message. Each PDU type has a numerical value, defined in the BACnet protocol specification:

PDU Type (Code)	APDU (Structure)
0x0X	BACnet-Confirmed-Request-PDU
0x1X	BACnet-Unconfirmed-Request-PDU
0x2X	BACnet-SimpleACK-PDU
0x3X	BACnet-ComplexACK-PDU
0x4X	Segment ACK
0x5X	Error PDU
0x6X	Reject-PDU
0x7X	Abort PDU
0x8X-0xfX	Reserved

The PDU type is in correlation with the type of the requested service. Service types have numerical identifications in the same way as PDU types, making the implementation to filter according to the requested service types similar. The type of the service determines the response generation and the mode of transmission.

After implementing the stand-alone server application we have to merge it to be a part of the honeypot, by implementing additional methods to start and terminate the server. Finally we have to import the server module to the main file and add the created server's class to the protocol's list to create an instance of the BACnet module during the thread spawning.

This section described the implementation of BACnet protocol to improve Conpot's functionality. BACnet became a national standard in more than 30 countries across the globe and the vendors and manufacturers show interest in exploring and supporting the BACnet technology, this enhancement can give additional power to organizations using Conpot or honeypots in general.

3.6.4 Testing and evaluation of BACnet

This subsection will illustrate the functionality of the implemented BACnet protocol. Since we have no access to BACnet devices to setup a production network and test with the honeypot, we create a testing suite that will mimic the communication of the devices.

To test the implemented server's functionality we implement a standalone testing script which is capable to transmit specified BACnet messages to the server. The implemented "clients" are able to send ReadProperty, Who-Is and Who-Has request. The implementation of the test cases is not covered in this thesis. We run the tests and monitor the outputs of the script and the log files of the honeypot. The test cases initiate valid request, which can be used to confirm that the functionality of the server is in accordance with the BACnet protocol specification.

An example of a client executing a ReadProperty request for the Present Value property of an Analog Input object with Object Identifier of 14 is illustrated in the code fragment below. The code snippet shows the encapsulation of the request and the encoded response from the BACnet server:

```
BACpypes ReadProperty request without encoding:
    pduExpectingReply = 1
    pduNetworkPriority = 0
    apduType = 0
    apduMaxResp = 1024
    apduService = 12
    apduInvokeID = 101
    objectIdentifier = ('analogInput', 14)
    propertyIdentifier = 85
    pduData = x''
```

```
BACpypes ReadProperty request encapsulated into PDU:
    pduExpectingReply = 1
    pduNetworkPriority = 0
    pduData = x'00.04.65.0C.0C.00.00.00.0E.19.55'

Received data from the server:
'0e\x0c\x0c\x00\x00\x00\x0e\x19U>DB\x88\x00\x00?'
```

Conpot is actively logging the attack events; it identifies the attack's source address and logs the communication with the BACnet server:

```
2015-03-22 13:43:53,646 New bacnet session from 127.0.0.1 (5fb5e713-6c4a-42dc-9427-
535b55c3fecc)
2015-03-22 13:43:53,647 New connection from 127.0.0.1:55765. (5fb5e713-6c4a-42dc-
9427-535b55c3fecc)
2015-03-22 13:43:53,647 Bacnet PDU received from 127.0.0.1:55765.
(ConfirmedRequestPDU)
2015-03-22 13:43:53,647 Bacnet indication from 127.0.0.1:55765.
(ReadPropertyRequest)
2015-03-22 13:43:53,648 Bacnet response sent to ('127.0.0.1', 55765)
(ComplexAckPDU:ComplexAckPDU)
2015-03-22 13:43:53,649 Bacnet client disconnected 127.0.0.1:55765. (5fb5e713-6c4a-
42dc-9427-535b55c3fecc)
```

After decoding the received response on the client side we can analyze the contents of the message to ensure that the communication is proper to the BACnet specification:

```
pduDestination = ('127.0.0.1', 55765)
pduExpectingReply = 0
pduNetworkPriority = 0
apduType = 3
apduService = 12
apduInvokeID = 101
objectIdentifier = 14
propertyIdentifier = 'presentValue'
propertyValue
        <bacpypes.primitivedata.Tag(real) instance at 0x04c20750>
            tagClass = 0 application
            tagNumber = 4 real
            tagLVT = 4
            tagData = '42.88.00.00'
pduData = x''
```

Next we alter some of the test cases to generate some invalid requests. According to the request's type it should be either ignored or dealt with an error response. Using the same script we send a ReadProperty request to the server asking for a nonexistent object. Conpot outputs the event information:

```
2015-03-22 13:46:28,523 New connection from 127.0.0.1:55770. (5fb5e713-6c4a-42dc-
9427-535b55c3fecc)
2015-03-22 13:46:28,523 Bacnet PDU received from 127.0.0.1:55770.
(ConfirmedRequestPDU)
2015-03-22 13:46:28,523 Bacnet indication from 127.0.0.1:55770.
(ReadPropertyRequest)
2015-03-22 13:46:28,523 Bacnet ReadProperty: no object found
2015-03-22 13:46:28,524 Bacnet client disconnected 127.0.0.1:55770. (5fb5e713-6c4a-
42dc-9427-535b55c3fecc)
```

Finally we test the implemented Unconfirmed-Request type commands, like Who-Is and Who-Has. First we send a Who-Is request and wait for the honeypot to log the attack:

```
2015-03-22 13:45:09,191 New connection from 127.0.0.1:55766. (5fb5e713-6c4a-42dc-
9427-535b55c3fecc)
2015-03-22 13:45:09,191 Bacnet PDU received from 127.0.0.1:55766.
(UnconfirmedRequestPDU)
2015-03-22 13:45:09,191 Bacnet indication from 127.0.0.1:55766. (WhoIsRequest)
2015-03-22 13:45:09,192 Bacnet response sent to *:*
(UnconfirmedRequestPDU:IAmRequest)
2015-03-22 13:45:09,192 Bacnet client disconnected 127.0.0.1:55766. (5fb5e713-6c4a-
42dc-9427-535b55c3fecc)
```

Next we transmit a Who-Has request and monitor the output:

```
2015-03-22 13:45:44,617 New connection from 127.0.0.1:55767. (5fb5e713-6c4a-42dc-
9427-535b55c3fecc)
2015-03-22 13:45:44,618 Bacnet PDU received from 127.0.0.1:55767.
(UnconfirmedRequestPDU)
2015-03-22 13:45:44,618 Bacnet indication from 127.0.0.1:55767. (WhoHasRequest)
2015-03-22 13:45:44,618 Bacnet response sent to *:*
(UnconfirmedRequestPDU:IHaveRequest)
2015-03-22 13:45:44,618 Bacnet client disconnected 127.0.0.1:55767. (5fb5e713-6c4a-
42dc-9427-535b55c3fecc)
```

Analyzing the results above show that Conpot's BACnet server implementation is capable to classify the incoming request and provide the attacker a reply, imitating a legit network of BACnet devices.

3.7 Adding support for IPMI protocol

3.7.1 IPMI protocol

The Intelligent Platform Management Interface (IPMI) provides autonomous monitoring and recovery features implemented directly in platform management hardware and firmware. The main aspect of IPM is that all the implemented features, like inventory, monitoring, logging, and recovery control functions are available independent of the main processors, BIOS, and operating system even, when the system is in a powered down state. Platform status information can be obtained and recovery actions initiated under situations where system management software and other management mechanisms are unavailable.

The autonomous computer subsystem of an IPMI typically consists of multiple components:

- a Baseboard Management Controller (BMC),
- an Intelligent Platform Management Controller (IPMC),
- an Intelligent Chassis Management Bus (ICMB),
- a System Event Log (SEL),

- several Sensor Data Record Repositories (SDR),
- and Field Replaceable Unit Information (FRU).

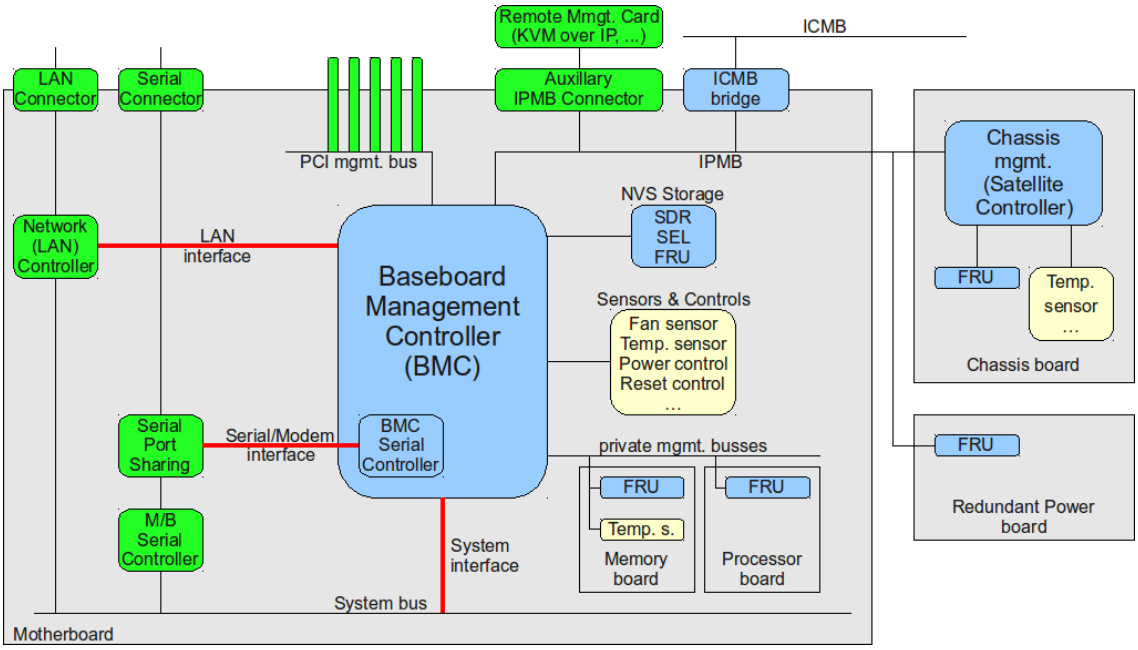


Figure 16: IPMI topology[7]

The main component of the IPMI architecture is a microcontroller called the BMC, which manages the interface between system management software and the platform management hardware, provides autonomous monitoring, event logging, and recovery control, and serves as the gateway between system management software and the IPMB/ICMB. The IPMB is an I^2C based serial line bus using byte-level transport for transferring IPMI messages between intelligent devices, typically microcontrollers. The ICMB is a standardized interface using character-level transport for inter-chassis communication between intelligent chassis.

The BMC is supposed to be always on when the system is plugged in, or even if the system is off. The management system communicates with the management controller; the BMC provides a normalized interface to all the sensors, events, and FRU data in the system. The IPMI stores all the information related to these interfaces in a separate memory areas. The SEL is a central, non-volatile container for event related information. This type of storage permits reading and deleting data via IPMI commands. Since the memory for SEL is limited, it must be periodically checked and deleted, so additional events can be stored and documented. The event log typically belongs to the BMC. The SDR repositories store information about the type and number of sensors and other components of the system. The SDRs are stored in a central, non-volatile storage area. The BMC must have a main SDR repository, which has to be writable. The last type of storage is the FRU data container. This storage contains information for several modules in the system, usually serial numbers, port numbers, model numbers and other asset

[7] https://www.thomas-krenn.com/en/wiki/IPMI_Basics

tags. In case of a problem or a change occurs in the system, the BMC handling the sensor detecting the change may issue an event. This allows management software to detect these problems or changes without having to poll every sensor constantly. More information about the architecture of IPMI can be found in the IPMI Specification [9].

3.7.2 Communication service in IPMI

IPMI allows messaging through various interfaces, including system interfaces for local access (like Keyboard Controller Style, System Management Interface Chip, Block Transfer, SMBus System Interface), serial interface, LAN interface, ICMB and PCI Management Bus. All IPMI messages share the same fields in the message regardless of the interface that they're transferred over. Messages are available over every IPMI-specified interface with different encapsulation according to the needs of the particular transport. This enables the conversion between interfaces by changing the underlying driver for the particular transport.

IPMI communication service uses a request/response messaging. IPMI request messages are usually commands, allowing the requestor to perform several managing tasks on the system. The use of a request/response protocol facilitates multi-master operation on busses like the IPMB and ICMB, allowing messages to be interleaved and multiple management controllers to directly intercommunicate on the bus.

A typical IPMI message consists of a Network Function Code, a Request/Response Identifier, a Requester's ID, a Responder's ID, an IPMI command and additional data. IPMI commands are grouped into different classes by function, using a field called the Network Function Code. This functional grouping simplifies the organization and management of the assignment and allocation of command values. The Network Function Code values are described in Table 5 of the IPMI Specification [9]. The Request/Response ID field differentiates the IPMI message's type. The Requester's and Responder's IDs identify the source and destination addresses of the IPMI messages. The Command field holds a one-byte value that specifies the actions desired to be executed by the requestor. More information about the command value assignments can be found in Appendix G of [9]. The Data field contains any additional parameters for a specified message.

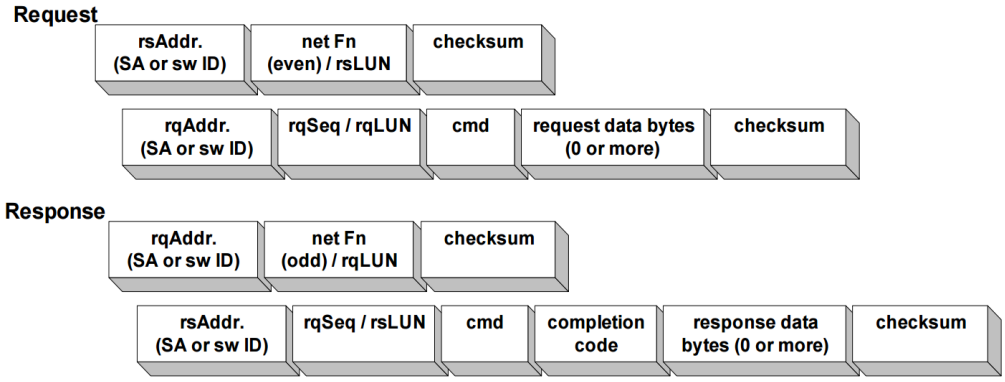


Figure 17: IPMI message format Source: [9]

An IPMI messaging connection to the BMC is accomplished by creating a session-less connection or establishing a single-session/multi-session connection. Session-less connections are unauthenticated, usually used over system interfaces and IPMB, while single-session/multi-session connections require user authentication. To implement an IPMI server module in Conpot, we have to classify incoming traffic and establish an IPMI session for new participants. The session activation procedure of the IPMI protocol will not be described in this paper, since detailed information can be found in section 6.12.7 and 13.15 of [9]. However we will refer to some events during the process, it is advised to acquire basic knowledge about the flow of the events (Figure 18.).

The deviation between IPMI v1.5 and v2.0 startup procedure lies in security improvements. In IPMI v2.0 after finishing the Discovery the participants exchange an RMCP+ Open Session Request/Response and a series of RAKP Messages. These messages contain random generated numbers, identification information and other data necessary to ensure communication in a protected session. Once the connection between the client and the server is active, the server should execute the desired commands on a "fake", emulated BMC in a way to deceive the attacker and convey a response back to the client. To make this approach feasible, we have to implement an additional emulated BMC beside the IPMI server. This BMC should be able to handle basic operations like system power management and similar.

The upcoming section will describe the realization of the IPMI server module of Conpot.

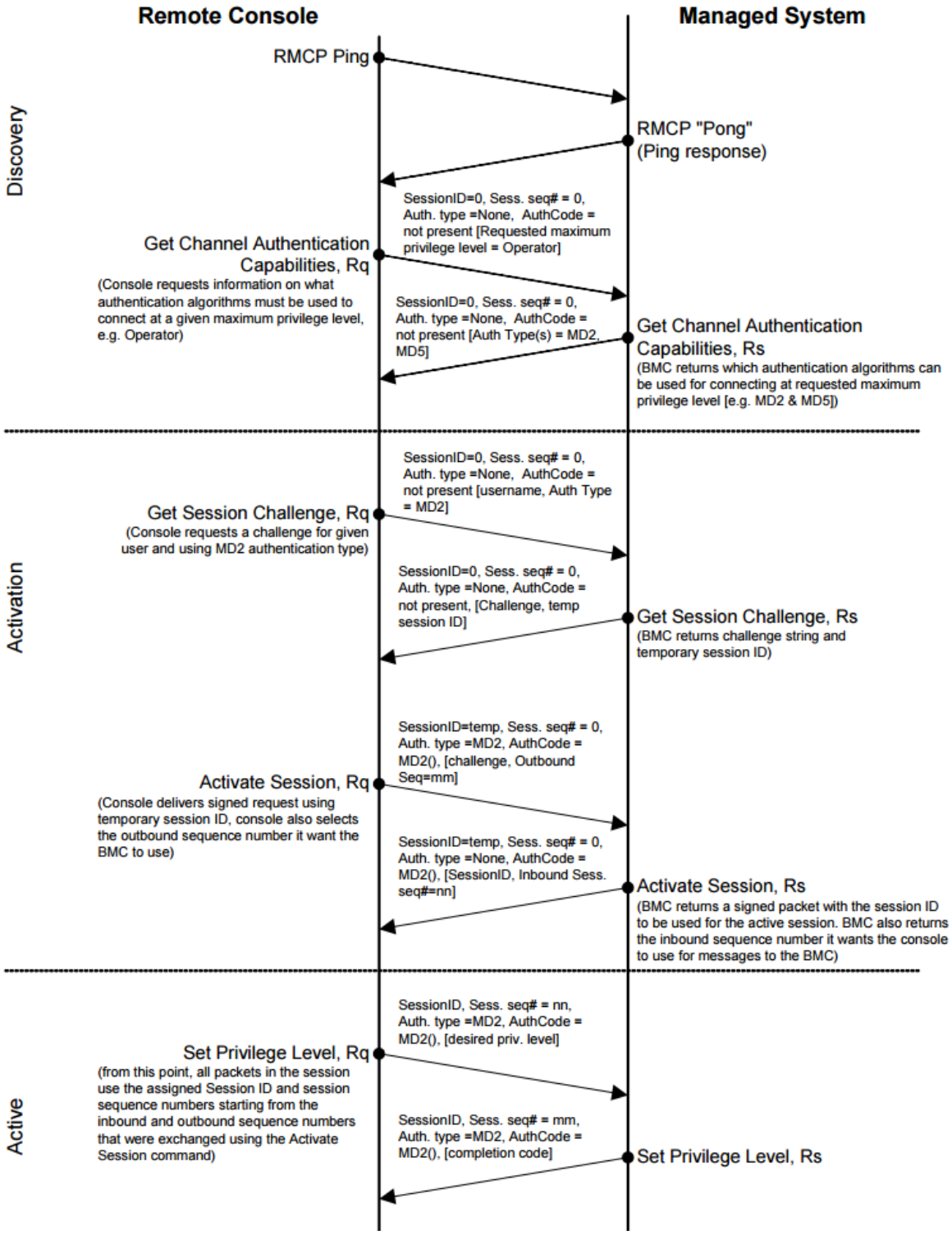


Figure 18: Session Startup for IPMIv1.5 Source: [9]

3.7.3 Implementation

We create a stand-alone Datagram Server running on port 623. On incoming traffic the server has to be able to determine if the request is part of an active session, a pending connection establishment or session-less communication. To overcome this problem we use python dictionaries containing the clients and their session. In case of the source of the request does not have an active or pending session,

a new session instance is created and assigned to the client. The classification of the session is resolved by using a staging feature. The stage of a session is a value representing a certain state of connection establishment. By altering the stage's value on a given session we can ensure that the communication will be handled correctly.

We create a session module, which will be responsible for parsing the received messages once it was classified by the server and building a proper response to any kind of requests on the application level. To avoid implementing functions for parsing and constructing RMCP and RMCP+ messages we will use the pyghmi library's IPMI implementation with modifications to suit our needs in the IPMI server. This library is capable to handle both IPMIv1.5 and IPMIv2.0 interfaces, widening the area of use for the honeypot. Using pyghmi's abilities we can obtain the received request's payload type. The payload type identifies the type of the message, which is used to define the server's behavior. Using the staging feature and the payload type identification the server can establish connection with multiple clients via RAKP messages.

Once the connection establishment is resolved, the server has to classify the commands in the IPMI requests and provide a proper reply to the attacker. For this to be feasible we create a BMC module, which will obtain the requests addressed to the BMC from the server. We initialize the BMC with properties that will represent various aspects of the hardware and software of the device. These attributes symbolize "permanent" changes to the emulated BMC after the attacker terminated the connection. Finally we define the IPMI server's behavior for additional IPMI commands, like Get User Access and Get User Name. The reason these commands were chosen, is that the mentioned commands play important roles during several known attacks for IPMI devices. More information about the IPMI protocol's vulnerabilities [8] regarding this topic can be found in Dan Former's research. These commands allow the user to retrieve the active user accounts registered on the BMC. Using the mentioned commands in combination with additional IPMI commands and external password cracking programs the aggressor can exploit the managed system by hijacking password hashes sent by the BMC or by injecting a new user account with elevated access privileges serving as a backdoor. To implement specific behavior to the server for these IPMI commands we have to decode the incoming messages once the session is activated and filter according to their Net Function Code and Command Code. The proper content of the response messages for supported IPMI commands are described in section 22 of [9]. The user accounts have to be created in a way to provide a certain level of customizability to the honeypot users. There's a demand for creating a template file and its definition to hold these data. The server has to send back these data to the requestor after proper encoding. The pyghmi library is used to handle the connection termination. The only issue we have to resolve, is to remove the specified session

[8] https://community.rapid7.com/community/metasploit/blog/2013/07/02/a-penetration-testers-guide-to-ipmi

from our active session list, in case of the same client sends another request. After that we have a stand-alone IPMI server that is capable to respond to a subset of IPMI requests. The usage of the libraries and the way of the realization allows further improvements in the implementation of the server, so the subset of commands can be easily broadened by filtering out more messages by their Net Function Code and Command Code in the handle_client_request() function of the IPMI server module. After creating the proper response message for the given request described in the IPMI specification, by passing the created response message with the corresponding response code to the _send_ipmi_net_payload() function, the server handles the rest of the communication.

Merging the stand-alone server into Conpot is achieved by implementing additional functions to the server, which will be responsible for the starting and stopping of the gevent Datagram Server. Furthermore we have to add the IPMI server class into the protocol mapping list of Conpot, which will be used to create an instance of the mentioned class during the startup procedure of the honeypot software.

3.7.4 Evaluating the results

This subchapter will show some of the functionality of the implemented IPMI server. For testing purposes we will use a non-commercial Linux utility called IPMItool. IPMItool provides a command-line interface for both IPMI v1.5 and v2.0 interfaces. Using the utility we will send various commands to the emulated BMC and analyze their communication using the verbose options on both software. However, this paper will only show a small subset of the testing, with limited description of the results, due to the length of the outputs provided by both of the software.

First we start the testing with the emulated BMC. Using IPMItool we initiate each command in the implemented set of supported IPMI messages and analyze the output and the communication procedure on both Conpot's and IPMItool's side. The current implementation of the BMC is capable to handle power and boot management instructions along with some of the chassis management commands. A fragment from the output of the specific "chassis status" command will be shown below to illustrate the logging capabilities and response generation of Conpot's IPMI server module.

```
debian@debian:~$ ipmitool -I lanplus -H 127.0.0.1 -p 623 -U Administrator -P
Password chassis status
System Power         : off
Power Overload       : false
Power Interlock      : inactive
Main Power Fault     : false
Power Control Fault  : false
Power Restore Policy : always-off
Last Power Event     :
Chassis Intrusion    : inactive
Front-Panel Lockout  : inactive
Drive Fault          : false
Cooling/Fan Fault    : false
```

Conpot without verbose logging only logs the basic information about the attack, such as time of occurrence, source of the attack, type of the request and the end of the communication:

```
...
2015-05-08 08:44:56,665 New IPMI traffic from ('127.0.0.1', 38102)
2015-05-08 08:44:56,668 IPMI BMC Get_Power_State request.
2015-05-08 08:44:56,668 IPMI Session closed 2695013284
...
```

During the testing procedure for the power management instructions we compare the IPMItool's output with the emulated BMC attributes' values, for ensuring that any changes made by a client are properly stored. This way, in case an attack is initiated form several source addresses, the chances of successfully fingerprinting the IPMI server would be smaller. Executing the implemented commands against Conpot while monitoring and analyzing the communication procedure shows that the new IPMI server of Conpot fulfills the requirements of the IPMI protocols specification.

Next we test a different set of commands implemented as additional instructions to the emulated BMC. We start IPMItool to send a "user list" command to Conpot to retrieve the user accounts registered on the emulated BMC.

```
debian@debian:~$ ipmitool -I lanplus -H 127.0.0.1 -p 623 -U Administrator -P
Password user list
ID  Name             Callin  Link Auth   IPMI Msg   Channel Priv Limit
1   Operator         true    false       false       OPERATOR
2   Administrator    true    true        true        ADMINISTRATOR
3   User2            true    false       false       USER
4   User3            true    true        true        CALLBACK
5   User1            true    true        true        USER
```

Conpot's IPMI server using verbose logging prints out the whole communication and generates the proper reply seen above.

```
...
2015-05-08 08:38:42,701 IPMI server started on: ('0.0.0.0', 623)
2015-05-08 08:38:45,483 New IPMI traffic from ('127.0.0.1', 47914)
2015-05-08 08:38:45,483 New IPMI session initialized for client (('127.0.0.1',
47914))
2015-05-08 08:38:45,483 Connection established with ('127.0.0.1', 47914)
2015-05-08 08:38:45,483 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,484 Incoming IPMI traffic from ('127.0.0.1', 47914)
2015-05-08 08:38:45,484 IPMI open session request
2015-05-08 08:38:45,484 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,485 Incoming IPMI traffic from ('127.0.0.1', 47914)
2015-05-08 08:38:45,485 IPMI rakp1 request
2015-05-08 08:38:45,485 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,485 Incoming IPMI traffic from ('127.0.0.1', 47914)
2015-05-08 08:38:45,486 IPMI rakp3 request
2015-05-08 08:38:45,486 IPMI rakp4 sent
2015-05-08 08:38:45,486 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,486 Incoming IPMI traffic from ('127.0.0.1', 47914)
2015-05-08 08:38:45,487 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,487 IPMI response sent (Set Session Privilege) to ('127.0.0.1',
47914)
```

```
...
5-08 08:38:45,493 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,493 IPMI response sent (Get User Access) to ('127.0.0.1', 47914)
2015-05-08 08:38:45,493 Incoming IPMI traffic from ('127.0.0.1', 47914)
2015-05-08 08:38:45,494 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,494 IPMI response sent (Get User Name) to ('127.0.0.1', 47914)
2015-05-08 08:38:45,494 Incoming IPMI traffic from ('127.0.0.1', 47914)
2015-05-08 08:38:45,494 IPMI response sent to ('127.0.0.1', 47914)
2015-05-08 08:38:45,495 IPMI response sent (Close Session) to ('127.0.0.1', 47914)
2015-05-08 08:38:45,495 IPMI Session closed 2695013284
...
```

For any non-supported command Conpot logs the communication and sends a proper message indicating that the requested command is not supported.

Evaluating the results above we came to the conclusion that the realized IPMI server is an acceptable solution to broaden Conpot's area of use. The only issue with the implemented server is the limited subset of commands it is capable to handle. However this issue can be resolved in the future by adding more commands to the currently filtered ones.

4 Implementations within Glastopf

In this chapter we will show the implementations to Glastopf's issues previously listed in section 3.1. The used development environment is also described in the section mentioned above. The next subchapters will describe the analysis and implementation of the solution to the given problems along with their evaluation.

4.1 Issue with Glastopf's web server

Using the default configuration of Glastopf, the honeypot emulates a vulnerable web server on port 80. Glastopf's template files located at glastopf/modules/handlers/emulators/data/templates/ define the web application's interface, implementing a basic login form, a main content part and a comments section by default. The problem with the web server is the following: after posting a comment, it does not shows up in the comment section.

4.1.1 Problem analysis

Following a post, there are no visual feedback or error messages provided on the interface, assuming that the implementation of the module responsible for posting the comments on the web application interface is not working properly. Investigating the log files generated by Glastopf, shows that the posting procedure is actually logged: the event information is stored in the glastopf.db database and the content of the comment is also saved to a separate text file.

To confirm the assumption above, we have to examine implementation of the visualization process of the comments block on the web server's interface. The comment handling is located in the glastopf/modules/handlers/emulators/comments.py file. Debugging the module shows that the comments stored in the display_comments string variable does not get included into the generated response on line 62. Upon studying python's standard libraries, we can confirm that the issue is related to the safe_substitute() function call. This method instead of generating a KeyError or ValueError exception, it silently ignores malformed templates containing errors and returns a usable string.

We identified the issue with the web server's interface. In the next subsection we describe the implementation of the solution to Glastopf comment visualization.

4.1.2 Resolving the issue

As we stated above, the problem is generated by python's safe_substitute() function call, which looks for a placeholder nonexistent in the template files. Glastopf template files are using the jinja2 library,

which is a fast, secure and designer-friendly template engine, modelled after Django's template system. It features an optional sandboxed template execution environment and automatic HTML escaping for applications. Glastopf's web interface is defined by two template files: base.html and index.html, both located at glastopf/modules/handlers/emulators/data/templates/. The implementation of the templates use the features of Jinja, which allows to set up template inheritance. The base template provides the skeleton structure of the standard page via defining a number of blocks. The index.html is a more specific template, which inherits from the base template and extends it by specifying the contents of each block.

These features allows us to define a separate block for the comment section and replace the block section a valid placeholder. It is desirable to have Jinja output the actual placeholder name and not handle it as a variable. To achieve this we have to mark the comments block as a raw, this way we can ensure that it is not going through an escape filter. Not using auto-escaping could cause more issues, since it creates additional security loopholes, for example vulnerability to Cross Site Scripting (XSS) attacks. XSS is a security exploit, where the attacker injects malicious client-side scripts into web applications. This way the intruder can gain elevated access-privileges to sensitive content, session cookies, and a variety of other information maintained by the web application on behalf of the user.

To work around this issue we have to ensure that the posted comments are going through an escape filter before they get stored and processed. After implementing these changes we make sure that the comments section is working as expected and the logging features of Glastopf are still appropriate.

4.1.3 Testing the vulnerabilities

The testing process includes posting different texts as a comment using the web application's interface while monitoring Glastopf behavior. We are expecting the posted comments showing up on the page, the action stored as an event into the database and the comments' contents saved into a text file.

During the testing we try various texts, including normal texts and JavaScript code snippets to make sure that the honeypot is protected against XSS attacks. After several attempts we can confirm that the comment section is functional and Glastopf behaves as expected; the honeypot stores the attack events into a database and the posted comments into a text file.

```
...
2015-03-22 14:35:24,493 (glastopf.glastopf) 127.0.0.1 requested POST /comments on
debian:80
2015-03-22 14:35:24,561
(glastopf.modules.handlers.emulators.dork_list.database_sqla) Done with insert of 1
dorks into the database.
...
```

The contents of the text file also confirms the proper functionality:

```
debian@debian:~/glastopf$ sudo cat data/comments.txt
<br/><br/>Testing comments section in Glastopf's web application interface
<br/><br/><script>alert(document.cookie)</script>
...
```

Evaluating the results above, the implementation of the solution can be considered as a success. The posted comments show up on the web interface and also get saved in a text file. The event logging in Glastopf stores the attack related information in the database. Trying simple XSS attacks on Glastopf shows no vulnerability, thus providing some protection against basic automated tools.

4.2 Storing the attacker's IP address and port

The issue described in this chapter includes the modification of the sqlite3 database of Glastopf, where the attack related information is stored. Furthermore, creation of a migration script, which is capable to execute the modification mentioned above is desired.

4.2.1 Problem analysis

To resolve the problem, first we have to study the methods that store attack related data in Glastopf. Upon investigating the implementation of the procedure, it has been clarified that in case of an interaction with the honeypot, the software creates a new instance of the AttackEvent class, which will be the container for all the valuable information. The class itself, as described in the glastopf/modules/events/attack.py file, holds the data about the time of the occurrence of the event, the IP address and port number of the source, and some information about the type of the attack as well in form of a python dictionary.

The dictionary containing the information gets stored in a database created at the initialization process of Glastopf. The creation of the database and the insertion of the data are defined in the glastopf/modules/reporting/main/log_sql.py file.

The whole procedure is controlled by the glastopf/glastopf.py script. This file holds the GlastopfHoneypot class, which is responsible for the functionality of the honeypot feature of the software. In order to implement the modifications in the database schema, we have to modify the creation, data insertion methods. Moreover, we need to alter the source code in modules using the database. Finally we have to implement a utility tool that is capable to transform an existing database to the desired format.

4.2.2 Altering the storing procedure

In the first part of the solution we have modify the schema of the database to store the IP address and the port number of the source in separate columns. The creation of the related part of database is executed in the log_sql.py file. The Database class found in this file is responsible for the construction of the "events" table of the database. This table contains all the attack related data, like timestamps, addresses, patterns and information about the executed requests. By altering the method that generates the table, we can modify table to the desired schema.

The information about the attacks are stored as attributes of an instance of the AttackEvent class during the logging procedure. Before saving the data in the database, all the information about each event get stored in a separate python dictionary. Later these dictionaries are used to insert the events' data to the main database. By modifying the class's attributes and methods using these data we can ensure that the insertion procedure will be in accordance with the altered table.

Finally we have to modify the logging procedure of Glastopf and locate all the files that are accessing the database in some way. For all the located code fragments we have to separate the attacker's source address into an IP address and a port number.

The second part of the solution is about to construct a migration script, which is capable to modify a database created according to the old schema and transform it to the new version. The chosen approach involves creation of a python script. The script uses the sqlite3 library and executes SQL commands, which will modify the "events" table's schema and copy the existing data sorted into the new table. Hence the SQLite library lacks the ability to remove a single column from an existing table we have to find a workaround to solve this issue. As stated in the library's website a possible solution could be to recreate the table according to the desired schema[9]. Copying all the required data into a temporary table and dropping the old, unwanted table creates the desired table. By simply renaming the temporary table the above procedure's result is the same as dropping one column from the original table.

4.2.3 Analyzing the results

The implementation will be considered as a successful solution to the issue if Glastopf's logging functionality is proper to the modified database, meaning that the honeypot is not going to terminate with an error message indicating a problem with the workflow of the altered database in any case. To test the functionality on a wider scale to cover a larger amount of modules using the database logging feature of Glastopf, we will run the test suites of the software and attack the honeypot in different ways

[9] http://www.sqlite.org/faq.html#q11

described in the "Attack examples and samples" section of [10]. During the attack events we will monitor Glastopf's behavior and it's interaction with the database. Executing various attacks shows expected behavior: the honeypot logs the attack related information properly into the modified database and sends a proper reply to the aggressor.

The testing process of the migration script lies in executing the script over a database created with the old format. Once the execution is finished we ensured that the data stored in the modified database represent the same logical relations as in the original database.

Evaluating the achieved results we can state that both extensions accomplished to solve the given issue and provide the desired outcomes.

4.3 Enable logging in Glastopf's profiler

This section will provide explanation for the implementation of a solution to Glastopf's profiler issue. The problem regarding Glastopf's profiler module is that lacks any kind of ability to log events occurred and handled by the module. This problem prevents the user from getting information about attacks handled by the profiler module. The upcoming subsections will describe the issue in detail and the implementation of the designated logger for the profiler.

4.3.1 Problem analysis

The profiler module of Glastopf is responsible for analyzing the attack and its source. It stores relevant information about the attacker, like source IPs, country codes, amount of requests from a specific IP, and much more in a database. The database is updated regularly in predefined time periods, however no information is conveyed to the user about the event. Furthermore, information about occurrences of new attack sources, possible errors encountered within the module are not logged either. To resolve the issue we have to create a logger within the profiler module. We will be gathering relevant information and use the mentioned logger to convey the messages to the user. These messages will contain information about occurrences of regular database updates along with the timestamp for the closest upcoming future update. Moreover, information about attack events will be logged in the time of occurrence. Additional functionality of the profiler is disabled due to the lack of logging capabilities. A possible solution to issue is to implement a designated logger module that will eliminate the problem by handling the specific events that require the mentioned functionality. The upcoming subsection will portray the implementation process of the generic and the designated logger as well.

4.3.2 Resolving the issue

To implement the generic logger in the profiler module of Glastopf we use python's built-in logging API provided by a standard library module. This API provides a large amount of functionality to handle different kind of events from informational messages through warnings to error reports. We instantiate a generic logger via module-level function of the API called getLogger(). We use this logger to convey messages to the user about the database update events and real time events that invoke some functions of the profiler. We modify each function of the module to log relevant information, for the update events a useful information can be considered the date and time of the next forthcoming database update, another example can be the occurrence of a new source IP address during an attack. For this to be feasible we have to search through the database for each attack event and look for the IP address in the specific table. IP addresses are used as primary keys in the mentioned table, so redundancy and efficiency is not an issue with this method. Each time a new IP source is found, we insert it to the database and add it to a FIFO queue that holds all the occurred events from the last update. During the next update the queue will be emptied and attacker related information processed and updated in the database table. After these changes the profiler logs the events that resulted in invoking the module.

The next issue is enabling the additional functionality mentioned above. Besides traditional logging the profiler should handle the storing procedure of the posted comments belonging to a specific IP address. We implement the get_comments() and add_comment() static methods in the profiler, which will retrieve a list of logger modules in Glastopf and in case of an instance of the designated logger exists they'll invoke the logger's appropriate functions. The comments related methods in the profiler are called from the glastopf/modules/handlers/emulators/comments.py file, which is responsible for processing, storing and retrieving posted comments using the honeypot's web interface.

To implement the designated logger we create a separate module that we inherit from an already implemented BaseLogger class. The BaseLogger class serves as a generic module which is used for the auxiliary loggers in Glastopf. In this module we implement the get_comments() and add_comment() functions, which will be responsible for retrieving all the posted comments by a specific IP address and inserting an additional comment to the database belonging to an IP address. First we have to modify the database scheme to store the comments belonging to an IP address. This can be achieved by altering the glastopf/modules/processing/ip_profile.py python script that handles the creation and initialization of the relevant table in the database. Next we realize the get_comments() and add_comment() methods using the sqlite3 library and embedded SQL queries in python. Once the methods are implemented the profiler module of Glastopf is able to log statistical information during updates and information about attack events in real time.

In the next section we will show some examples of the logging capabilities of the improved profiler module and test the storing and retrieval functionalities of the designated logger.

4.3.3 Testing the logging capabilities

For testing the logging features of the profiler we start up Glastopf with verbose logging. This allows the output of debugging messages, which were primarily used for logging during the implementation of the generic logger in the profiler. To simulate different types of attack and generate traffic on the emulated honeypot's emulated web server we use GNU Wget and cURL by initiating simple HTTP requests.

```
...
2015-05-08 08:51:31,145 (glastopf.glastopf) Glastopf started and privileges dropped.
2015-05-08 08:51:36,257 (glastopf.glastopf) 127.0.0.1 requested GET / on debian:80
2015-05-08 08:51:38,284 (glastopf.modules.processing.profiler) Attack event occured
from new IP source (127.0.0.1). A new profile got stored in the database.
2015-05-08 08:51:38,284 (glastopf.modules.processing.profiler) Attack event added to
a queue for processing.
2015-05-08 08:51:38,352 (glastopf.glastopf) 127.0.0.1 requested GET /style.css on
debian:80
2015-05-08 08:51:38,391 (glastopf.modules.processing.profiler) Attack event from
known IP (127.0.0.1). Profile updates will be provided in 0:00:02.750597
2015-05-08 08:51:38,391 (glastopf.modules.processing.profiler) Attack event added to
a queue for processing.
2015-05-08 08:51:38,413
(glastopf.modules.handlers.emulators.dork_list.database_sqla) Done with insert of 1
dorks into the database.
2015-05-08 08:51:46,171 (glastopf.modules.processing.profiler) Regular database
profile updating finished. Next update will be at 2015-05-08 08:52:11.141653
...
```

Glatopf's profiler module logs the occurred events along with the database updates. To test the comment related functions we use an SQLite database browser and a modified version of the comments.py module to insert comments into the database and to output comments belonging to an IP address. Conpot's profiler is capable to properly store and retrieve comments from the database, meanwhile logging the different kinds of database events.

Analyzing the results above, we can consider the enhancements of Glastopf's profiler a success. Before the improvements were implemented the profiler module did not convey any information to the user. After the changes Glastopf's profiler logs attack events in real time, informs the user about database events, furthermore it's able to store the posted comments in the database grouped by IP addresses and retrieve all the data as well.

4.4 Fixing the HTTP responses

In the following subchapters we will look into the implementation of Glastopf's web server emulation, locate the source of the problem with the HTTP responses' status code generation. The general

description of the issue is that the implemented feature of changing the HTTP response code of an individual service emulator gets ignored and a default value of "200 OK" gets transmitted independent on any predefined settings.

4.4.1 Analyzing the issue

The emulation services of Glastopf are relying on the BaseEmulator class implemented in base_emulator.py in the glastopf/modules/handlers/ directory. Each service is using this as a super class and inherits its attributes and methods. These emulators can be found in separate scripts under glastopf/modules/handlers/emulators. The individual modules override the parent's handle() method, which is responsible for the given module's request and reply handling once the request classification has finished. Each of the emulators use the same set_reponse() method to set the reply's attributes such as status code, http body and header information. The definition of this method is located in the glastopf/modules/HTTP/handler.py file. This file contains the HTTPHandler class, which uses a standard python library called BaseHTTPServer to properly parse the requests and generate a response.

```
102     def set_response(self, body, http_code=200, headers=(('Content-type',
                                                            'text/html'),)):
...
111         self.send_response(http_code)
112         for header in headers:
113             self.send_header(header[0], header[1])
114         self.end_headers()
115         self.wfile.write(body)
```

The execution of the library's function on line 111 is responsible for the appropriate HTTP status code's transmission. Assuming that the current behavior is not one of the library's flaws we have to look for the alteration of the status code between the call of the set_response() method and the invocation of the first function inside the library's boundaries.

4.4.2 Debugging the related methods

Using a python debugger we set several breakpoints at the crucial parts of the source code and monitor the execution process. We start by generating attack events and tracing back the execution to the point where the predefined status code alters.

Investigating the execution process shows that the parameters are passed correctly in each emulator module and the modification is not inside the set_response() method. From here the execution returns the proper values inside the glastopf/glastopf.py file. No modification is carried out to the mentioned values within this module. The glastopf.py returns the correct values to the glastopf/wsgi_wrapper.py. In this file the debugger indicates an alteration in the status code's value in the following snippet:

```
40    header, body = self.honeypot.handle_request(req_webob.as_text(),
                                                   remote_addr, sensor_addr)
41    for h in header.splitlines():
42        if ':' in h:
43            h, v = h.split(':', 1)
44            res_webob.headers[str(h.strip())] = str(v.strip())
45    #this will adjust content-length header
46    res_webob.charset = 'utf8'
47    res_webob.text = unicode(body)
```

While the "header" variable contains the proper values the res_webob object that is used for transmission has faulty data. By examining the source code fragment it became clear that the part of the header containing the information about the response codes is ignored. However this does not explains the status code of "200 OK" stored as an attribute of res_webob. The res_webob is an instance of the webob.Response object, which contains everything necessary to make a WSGI response. WebOb is a Python library that provides wrappers around the WSGI request environment, and an object to help create WSGI responses. The objects map the specified behavior of HTTP, including header parsing, content negotiation and correct handling of conditional and range requests. Further investigation of Glastopf's source code does not clarifies the source of the assigned values to res_webob's attributes. This creates the assumption that it is a default behavior of the WebOb library, even though a detailed study of the library's documentation does not show any indication about the statement. However looking into the library's source code[10] makes clear that the initialization process of a webob.Response object set's the status code to "200 OK" in case no other value is defined:

```
104         if status is None:
105             self._status = '200 OK'
106         else:
107             self.status = status
```

This information makes clear that the issue can be resolved by storing the HTTP response code in the corresponding attribute of res_webob. To achieve this we have to parse the returned header in the wsgi_wrapper.py and assign the conveyed value to the proper attribute.

In the next section we evaluate the results achieved by implementing the solution described above. The testing will involve generating several requests and modifying one of the emulator modules for testing purposes only that can be altered to provide a wide variety of replies.

4.4.3 Testing the implementation

First we have to alter an emulator module for testing the changes made in Glastopf's behavior. The newly created emulator will be integrated into Glastopf for the time of testing the implementation of the issue. The emulator will generate replies and respond to HTTP/1.1 protocol's OPTIONS method. By altering the values given as the http_code parameter highlighted in the source code fragment below, we can set the desired HTTP status code in the reply:

[10] https://github.com/Pylons/webob/blob/master/webob/response.py#L104-107

```
...
18    from glastopf.modules.handlers import base_emulator
19
20
21    class OPTIONSRequest(base_emulator.BaseEmulator):
22        def __init__(self, data_dir):
23            super(OPTIONSRequest, self).__init__(data_dir)
24
25        def handle(self, attack_event):
26            attack_event.http_request.set_response('', http_code=200,
                                headers=(('Allow', 'OPTIONS, GET, HEAD, POST'),))
```

Explicitly assigning a value to the http_code in a version of Glastopf before the implementation of the fix resulted in a reply with status code 200 without taking into account the stated value. We change the highlighted parameter to "http_code=301", and use the cURL Linux utility to initiate an OPTIONS method request:

```
debian@debian:~$ curl -i -X OPTIONS 127.0.0.1
HTTP/1.1 200 OK
Server: Apache/2.0.48
Date: Sun, 19 Apr 2015 15:18:44 GMT
Allow: OPTIONS, GET, HEAD, POST
Content-Type: text/html; charset=utf8
Content-Length: 0
```

The example above illustrates the issue with Glastopf's HTTP response generation. After implementing the changes explained in the previous subsections we initiate another OPTIONS request via cURL:

```
debian@debian:~$ curl -i -X OPTIONS 127.0.0.1
HTTP/1.1 301 Moved Permanently
Server: Apache/2.0.48
Date: Sun, 19 Apr 2015 15:20:12 GMT
Allow: OPTIONS, GET, HEAD, POST
Content-Type: text/html; charset=utf8
Content-Length: 0
```

The status code of the response is now altered according to the predefined value. To ensure that the changes made does not cause further issues with the honeypot's behavior we provide some additional testing using a network mapper and security scanner tool called nmap. We run nmap against Glastopf with various options to cover a wide area of use of the modified honeypot. During the scanning procedures we monitor Glastopf's logging process and look for any unwanted behavior.

```
debian@debian:~/glastopf$ sudo glastopf-runner
2015-04-19 11:21:16,796 (glastopf.glastopf) Initializing Glastopf 3.1.3-dev using
"/home/debian/glastopf" as work directory.
2015-04-19 11:21:16,841 (glastopf.glastopf) Connecting to main database with:
sqlite:///db/glastopf.db
2015-04-19 11:21:16,915
(glastopf.modules.handlers.emulators.dork_list.dork_page_generator) Bootstrapping
dork database.
```

```
2015-04-19 11:21:19,303
(glastopf.modules.handlers.emulators.dork_list.database_sqla) Done with insert of
5096 dorks into the database.
2015-04-19 11:21:19,304 (glastopf.glastopf) Generating initial dork pages - this can
take a while.
2015-04-19 11:21:19,362 (glastopf.glastopf) Glastopf started and privileges dropped.
2015-04-19 11:21:34,180 (glastopf.glastopf) 127.0.0.1 requested GET / on debian:80
2015-04-19 11:21:34,311 (glastopf.glastopf) 127.0.0.1 requested GET / on debian:80
2015-04-19 11:21:34,333 (glastopf.glastopf) 127.0.0.1 requested GET /robots.txt on
debian:80
...
2015-04-19 11:21:34,579 (glastopf.glastopf) 127.0.0.1 requested OPTIONS / on
debian:80
2015-04-19 11:21:34,598
(glastopf.modules.handlers.emulators.dork_list.database_sqla) Done with insert of 1
dorks into the database.
```

Analyzing the received results shows no issues with Glastopf's logging capabilities, which means that the implementation to fix the honeypot's HTTP response handling can be considered as a success. In the next chapter we will show some areas of use for the newly created or modified modules of the honeypot software and summarize the project overall.

5 Conclusion

The goal of this thesis was to implement certain enhancements to specified honeypots. These extensions included fixing certain issues, improving existing modules or adding new functionality to the software. The first requirement was to research and understand the principles behind honeypot software, as the knowledge is essential for further development. Chapter 2 provided an explanation about fundamentals of honeypots along with a brief introduction to the software, which the development was concentrated around. Chapter 3 dealt with the resolution of issues and improvements of Conpot along with the testing of each solution. Chapter 4 focused on the implementation of solutions for Glastopf.

The benefits this work provides are the contributions of the implemented extensions to the White Hat community, which are integrated into the final versions of the honeypots. By upgrading these honeypot systems the users gain access to an improved version of the software providing more functionality.

Honeypots have gained a significant place in the overall intrusion detection and protection strategy. While these systems do not replace existing IDS technologies, the advantages that they bring as a complementary technology are hard to ignore. From the current point of development, honeypot software should advance to process a wider variety of protocols, thus extending the current range of honeypots. In the future this would allow the deployment of different honeypot systems to more enterprises sustaining better security for their production networks, gaining more information about the attacks of the Black Hat community and indirectly providing higher safety for everyone online.

References

[1] SZABÓ, A. *Preventív hálózatvédelmi rendszerek alkalmazási lehetőségei a támadások detektálására, valamint a módszerek elemzésére (Applicability of preventative network security systems for attack detection and method analysis) Part 2. Hadmérnök. VII. Évfolyam 2. szám.* Jun, 2012. [online], [cit. 2015-1-19].
URL: <http://hadmernok.hu/2012_2_szaboa.pdf>

[2] SPITZNER, L. *Honeytokens: The Other Honeypot*. Security Focus, 2003. [online], [cit. 2015-1-19].
URL: <http://www.securityfocus.com/infocus/1713>

[3] BAUMANN, R.; PLATTNER, C. *White Paper: Honeypots*, Swiss Federal Institute of Technology, Zurich, 2002.
URL: <http://www.rbaumann.net/download/whitepaper.pdf>

[4] *Know Your Enemy: Honeynets*. Last Modified: May 31, 2006. [online], [cit. 2015-1-19].
URL: <http://old.honeynet.org/papers/honeynet>

[5] *Modbus messaging on tcp/ip implementation guide V1.0b*. October 24, 2006 [online], [cit. 2015-1-19]
URL:<http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf>

[6] *Modbus over serial line specification & implementation guide V1.0*. Last Modified: November, 2002. [online], [cit. 2015-1-23]
URL: <http://www.modbus.org/docs/Modbus_over_serial_line_V1.pdf>

[7] TIERSCH. F. *BACnet Application Layer Protocol*. [online], [cit. 2015-3-9]
URL: < http://www.ipsta.de/download/AUTnet-030.pdf >

[8] ROHIT. G.; PUNYA. P. *White Paper: Data communication protocol for control networks enabling automated buildings*. 2014 [online], [cit. 2015-3-9].
URL: < http://www.ti.com/lit/wp/spry266/spry266.pdf >

[9] *Intelligent Platform Management Specification Second Generation v2.0.*. Last Modified: June, 2009. [online], [cit. 2015-3-21].
URL:<http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/second-gen-interface-spec-v2.pdf>

[10] LUKAS. R. *Know Your Tools: Glastopf*. Last Modified: November, 2010. [online], [cit. 2015-3-21].
URL:< http://www.honeynet.org/sites/default/files/files/KYT-Glastopf-Final_v1.pdf>

Appendix A

Setting up the honeypots

In this subsection we learn about the installation procedure of the software and we present some examples to the different setups of the honeypot.

As stated in the documentations of honeypots the software were primarily developed and tested for Ubuntu 12.04 LTS and Debian versions 7.2.0 (64bit) and 6.0.7 (64bit). Using newer versions of these distributions or other distributions based on different Linux kernel versions may cause the programs not to function properly, hence the user might encounter some errors causing the software to be unusable.

The development versions of both honeypots can be easily acquired from GitHub, due to the whole projects are open source and they are under the license of GPLv2. The required packages for each software are listed in a text file, which can be obtained from the corresponding repositories via package manager of a given distribution of the operating system. Once the requirements are met and the GitHub repository is cloned, the installation of the development version is handled by several python scripts. Finishing the installation process the honeypot is ready-to-use.

For further usage the configuration of Conpot is advised, though not necessary. Basic configuration options are listed in the default configuration file, located in conpot/conpot.cfg. This configuration file gives us the option to enable or disable some functionalities of the honeypot for example using syslog for security auditing, enabling automated cyber threat information exchange, sharing high-volume real-time data from different pieces of honeypot software between members of the Honeynet project information, and so on, with added customization abilities these features in detail. There are several templates featured already in Conpot, which define the list of emulated services. The current version (release 0.3.1) is shipped with three templates: a template emulating a proxy server, a template emulating an electricity meter named kamstrup 382, and a default template which provides basic emulation of a Siemens S7-200 micro PLC with an attack surface for the MODBUS, HTTP, SNMP and s7comm protocols. Advanced customization options require to edit the XML profiles in the protocol's directory.

Basic configuration of Glastopf can be done, by editing the glastopf/glastopf.cfg.dist file. The configuration file is divided into sections, where the user can modify the behavior of the honeypot. Glastopf contains a web application interface out-of-the-box, which can be found at

glastopf/modules/handlers/emulators/data/templates/. The style of interface can be changed in the relevant CSS file located in glastopf/modules/handlers/emulators/data/style/ directory.

Configuring behavior of Conpot

In spite of Conpot is a low involvement server-side honeypot, it features several communication protocol emulations. Each service is highly customizable in the protocol's template files providing a wider field of use, more specified environment and better overall user experience.

The upcoming chapters will enumerate the individual protocol customization options, showing examples on Conpot's template files and describing each protocol's opportunities in detail.

Emulation of HTTP protocol

The HTTP section defines the characteristics of the web server Conpot emulates. Detailed option configuration is available in the XML file located at conpot/templates/default/http/http.xml. The "global" section determines which methods should be enabled, along with the host and port of the emulated HTTP server:

```
    <global>
        <config>
            <!-- what protocol shall we use by default? -->
            <entity name="protocol_version">HTTP/1.1</entity>
            <!-- if we find any date header to be delivered, should we update it to a
real value? -->
            <entity name="update_header_date">true</entity>
            <!-- should we disable the HTTP HEAD method? -->
            <entity name="disable_method_head">false</entity>
            <!-- should we disable the HTTP TRACE method? -->
            <entity name="disable_method_trace">false</entity>
            <!-- should we disable the HTTP OPTIONS method? -->
            <entity name="disable_method_options">false</entity>
            <!-- TARPIT: how much latency should we introduce to any response by
default? -->
            <entity name="tarpit">0</entity>
        </config>

        <!-- these headers will be sent with each response -->
        <headers>
            <!-- this date header will be updated, if enabled above -->
            <entity name="Date">Sat, 28 Apr 1984 07:30:00 GMT</entity>
        </headers>
    </global>
```

The documentation[11] describes each part of the XML files we will refer to it for further explanation: The "headers" section is added to every page delivered by Conpot, allowing detailed modification of responses. The "tarpit" section slows down the delivery of the web page to simulate slower devices that would not deliver websites in a fraction of a second.

[11] https://glastopf.github.io/conpot/

Further sections of the XML file modify the responses of Conpot's HTTP server, for example the "htdocs" section configures the all individual files Conpot delivers, while the "statuscodes" section describes how the different status codes should be handled.

Emulation of MODBUS protocol

The XML file is configuring the MODBUS server is located at conpot/templates/default/modbus/ directory. According to the documentation's description the "slave" section allows to define the slaves Conpot emulates. Every slave definition is separated into "blocks". Each block further specifies the slaves' Read/Write functionalities conformed to the specifications defined by Modbus-IDA.

Emulation of SNMP protocol

The XML profile can be found at conpot/templates/default/snmp/snmp.xml. This file contains configuration of the management information bases.

These MIBs consist of a "symbol" and its "value". Several symbols feature dynamic values, which can be delivered by adding the engine definition to the template. The implemented engine types can be found in Conpot's documentation. Conpot is capable to compile the MIB files automatically, however the MIB file's location and providing Conpot with the appropriate parameters might be crucial for proper functioning.

The SNMP interface and the "tarpit" can be configured to adjust its behavior desired by the user. A tarpit is a service that purposely delays the transmission of data. It slows down the delivery of SNMP responses to simulate slower devices in the network. Tarpits are configured individually for each type of SNMP request. If not configured for a given type, the answers are generated instantly. Because SNMP uses UDP as transport protocol, it is prone to address spoofing. SNMP responds to small requests with responses containing bigger payloads. To avoid Conpot being used for traffic amplification attacks, the "evasion" feature has been implemented. This feature mitigates the risks by applying a threshold to each request type. This threshold specifies the number of request allowed per IP per minute and the number of request allowed overall per minute, what Conpot may accept. This feature of the honeypot can be adjusted in the SNMP XML profile as well.

Human Machine Interfaces

Conpot comes with a default Human Machine Interface. A human machine interface (HMI) is an apparatus which present processed data to a human operator. Through this interface based on the given information the operator interacts with the industrial process. In case of Conpot is running with the default configuration and no additional parameters, it will serve the default page on port 80 as an

additional service increasing the attack surface. This HMI can be easily modified by editing conpot/templates/default/http/htdocs/index.html, or the user can create a new using the hmi_crawler tool. The hmi_crawler is an additional utility to Conpot, which can be used to crawl an existing HMI to create a copy. Fetching a web application's HMI can be easily acquired by running the hmi_crawler against a web page. The hmi_crawler is located at bin/hmi_crawler. For more detail on the usage can be found in the documentation.

Once we obtained the necessary information about Conpot to provide a clear image about its capabilities, field of use and the user's options to customize the honeypot, we will proceed to explain the security issues of this software in more detail.

Configuring behavior of Glastopf

Configuring the web application interface

Glastopf provides a wide range of possibilities in terms of customizing the emulated web application. The related files can be found in the glastopf/modules/handlers/emulators/data/ directory. This directory holds all the HTML and CSS files necessary to set up the emulated web server, including layout, style and behavior adjustments.

Configuring logging capabilities

Glastopf's emulated web server and logging features can be adjusted in the configuration file located at glastopf/glastopf.cfg.dist. The file contains detailed customization options about the enabled logging features of Glastof, for example setting up the environment and database for logging. Furthermore it holds adjustable options for each separate logging module implemented in Glastopf to modify the honeypot's behavior.

Appendix B

Content of the CD

The attached CD contains the following directories and files:

- File **/thesis/bachelor_thesis.pdf** Technical report in PDF.
- File **/thesis/bachelor_thesis.docx** Technical report in docx format.
- File **/src/conpot_changelog.txt** List of changed files in Conpot.
- File **/src/glastopf_changelog.txt** List of changed files in Glastopf.
- Directory **/src** Source codes of the improved honeypots.