Pascal Gamper

Towards automated exploit signature generation using honeypots

Master Thesis MA-2007-09
March 2007 to September 2007

Tutor: Bernhard Tellenbach
Co-Tutor: Daniela Brauckhoff
Supervisor: Prof. Bernhard Plattner

Acknowledgement

This master thesis would not have been possible without the support of many people. The author wishes to express his gratitude to his tutor, Bernhard Tellenbach who was abundantly helpful and offered invaluable assistance, support and guidance. Furthermore, special thanks go to the co-tutor Daniela Brauckhoff and the other members of the Communication System Group (CSG), ETH Zurich, especially to Prof. Dr. Bernhard Plattner. The trip to Amsterdam was awesome!
Special thanks also to my classmates; Daniel Koller and Stefan Keller for sharing premises and numerous hours playing foosball.

Abstract

Intrusion detection systems normally use some kind of signatures to identify attacks. These signatures are manually created by experts in most cases. Nowadays a trend to more complex and fast spreading attacks can be observed and computer worms spread itself over the whole world within minutes. This is possible because vulnerabilities are known to the public just a few hours before the first exploit is available or sometimes even disclosed by the publication of exploit code. Attacks exploiting such a kind of vulnerability are called zero-day exploits. Zero-day exploits such as the Slammer worm render the manual signature generation useless for those attacks. The automated signature generation addresses this problem. It attempts to automatically identify malicious network traffic or software code in order to generate a specific signature for it.

This thesis is part of the EU project NoAH that aims at developing an automated signature generation mechanism that is based on attack alert information originating from a network of honeypots. A framework for tracking the connection state and decoding protocol-field information of application-level protocols provides additional knowledge for the signature generation.

In this thesis, we extended the framework with a plugin for the FTP protocol and did a redesign of its structure in favour of ease of use and extensibility. A signature generator for the Snort signature format has been implemented that uses both information from the honeypot network and application state information. It uses an external library for full protocol-field knowledge. The resulting signatures are able to protect server applications from buffer overflows if triggered by abusing a single protocol field. The false-positives rate for these signatures is almost zero since they capture the vulnerability characteristics of the detected attacks.

Contents

1 Introduction **15**
1.1 The trinity of trouble . . . 15
1.2 Intrusion prevention . . . 15
1.3 Situation in the Internet . . . 16
1.4 The Network of Affined Honeypots (NoAH) . . . 17
1.5 Problem statement . . . 17
1.5.1 Application plugin for the tracker framework . . . 17
1.5.2 Signature Generation Mechanism . . . 17
1.6 Structure of the Report . . . 18

2 Current state and Setup of the NoAH project **19**
2.1 NoAH architecture overview . . . 19
2.2 Components related to this work . . . 20
2.3 Containment system Argos . . . 20
2.4 The tracker framework at a glance . . . 21

3 Related Work **23**
3.1 Content-based signatures . . . 23
3.1.1 Honeycomb . . . 24
3.1.2 Polygraph . . . 24
3.1.3 Earlybird . . . 24
3.1.4 Autograph . . . 25
3.1.5 TaintCheck . . . 25
3.2 Flexible content-based signatures . . . 25
3.2.1 PADS . . . 25
3.2.2 PAYL . . . 26
3.3 Context and semantics aware signatures . . . 26
3.3.1 Nemean . . . 26
3.3.2 COVERS . . . 27
3.3.3 Polymorphic Worm Detection using structural Information of Executables 27
3.4 Other approaches . . . 27
3.4.1 DOME . . . 27
3.4.2 Paid . . . 28
3.4.3 HoneyStat . . . 28
3.4.4 Vigilante . . . 28

4 Application Protocol Plugin **29**
4.1 Requirements . . . 29
4.2 Exploit survey and Protocol Evaluation . . . 29
4.3 The File Transfer Protocol (FTP) . . . 30
4.4 The Garbage Collector Plugin . . . 31
4.4.1 Class SynchronizedMap . . . 31
4.5 The FTP Plugin . . . 33
4.5.1 Deriving a state machine for connection state observation . . . 33
4.5.2 The plugin design . . . 35
4.6 Evaluation . . . 36
4.6.1 Tracking states . . . 36

4.6.2 Measuring performance 37

5 Signature Generator 39
5.1 Requirements 39
5.2 Signature format evaluation 40
5.3 Architecture 40
5.3.1 Extracting alert information 40
5.3.2 Archiving signature information 41
5.3.3 Demonstrating signature capabilities 42
5.3.4 Attack replaying 43
5.4 A signature generation mechanism (SGM) 44
5.5 Evaluation 45

6 A more generic approach for signature generation and state tracking 47
6.1 Observations 47
6.2 Survey 48
6.3 Protocol knowledge 48
6.4 The NetBee framework 49
6.4.1 Network Virtual Machine (NetVM) 50
6.4.2 Network Protocols Description Language (NetPDL) 50
6.4.3 Packet Details Markup Language (PDML) 50
6.4.4 Usage of the NetBee library 51
6.5 Protocol state automaton 51
6.5.1 Architecture 51
6.5.2 Specifying protocol automata 53
6.5.3 Proposal for a timeout extension 53
6.5.4 Proposal for a framework for protocol state tracking and packet replaying 54
6.6 Synthesis of network packets on application-level 54

7 Refactoring the Signature Generator 55
7.1 Experiences with Signature Generation 55
7.2 Integrating NetBee 56
7.3 Flexible configuration 56
7.4 Architecture 56
7.5 Supported signatures 58
7.6 Evaluation 59
7.6.1 Measuring signature generation performance 59

8 Conclusions and Outlook 61
8.1 Results 61
8.2 Conclusions 61
8.3 Outlook 62

A NoAH core components: Extraction and Processing of attack information 63
A.1 Initial attack information 63
A.2 Current state of attack information extraction 64

B Glossary 67

C The original tracker framework 71
C.1 Component overview 71
C.2 Components and their algorithms 72

D Techniques in Signature Generation Mechanisms 77
D.1 Byte-Frequency Distribution (BFD) 77
D.2 Longest Common Substring (LCS) problem 77
D.3 Longest Common Subsequence (LCSeq) problem 77
D.4 N-Gram analysis 77
D.4.1 Z-score 78
D.4.2 Dice coefficient 78

D.5 Expectation-Maximization (EM) algorithm 78
D.6 Clustering 78
D.6.1 Hierarchical clustering 78
D.7 Rabin fingerprint 79
D.8 Bayes law 79
D.9 Content-based Payload Partitioning 79

E Signature examples **81**
E.1 Honeycomb signature 81
E.2 Earlybird signature 81
E.3 COVERS signature 82
E.4 Vigilante signature 82

F Installation manual for the application state tracker framework **83**
F.1 The tracker framework 83
F.1.1 Installing the tracker framework 83
F.1.2 Configuration of the tracker framework 84
F.2 Qemu, Argos and the Interface 84
F.2.1 Qemu 84
F.2.2 Argos 85
F.2.3 Networking 85
F.2.4 Tracker interface 86
F.3 Putting the Pieces Together 87
F.4 Ready-to-use Scripts 87
F.4.1 Qemu-ifup 87
F.4.2 Qemu-ifdown 88
F.4.3 Start-argos 88

G Signature Database **89**

H TODOs **93**

List of Figures

1.1 The spread of the Internet worm *code red* on July 19, 2001 (within 24 hours) . . 16
1.2 The spread of the Internet worm *slammer* on January 29, 2003 (within 30 minutes) 17

2.1 The core components of the NoAH architecture 19
2.2 The functioning components of the NoAH project which are related to this work . 20
2.3 The basic design of the tracker framework 21

4.1 First state machine of the FTP protocol specification 33
4.2 Second state machine of the FTP protocol specification 33
4.3 The derived state machine used by the FTP plugin 34
4.4 After each command / reply sequence one of the illustrated states is passed. They do not affect the FTP state machine itself but are meant rather for additional information. 34
4.5 The delays for processing the complete packet and processing the FTP protocol part with the FTP plugin respectively 38

5.1 The process structure after adding alert information extraction 41
5.2 The process structure containing a *MySQL* database 42
5.3 The ”one-to-many-to-many relationship” for the attack replaying mechanism . . . 43
5.4 The complete process structure of the signature generation mechanism 44

6.1 The basic class design of the finite state machine 52
6.2 The state machine embedded in the two applications state tracking and packet replaying 54

7.1 Class diagram that shows the relations between threads and information extraction 57
7.2 The class relations for the signature related data types 58
7.3 The delays for generating different signatures 60

A.1 Information provided by the containment system Argos 63
A.2 Additional packet information provided by the tracker framework 64
A.3 FTP protocol knowledge by a Tracker plugin 64
A.4 Overview of the information extraction process 65

C.1 The initialization phase of the Tracker 71
C.2 The packet processing phase of the Tracker 72

List of Tables

4.1 Tested FTP exploits for the metasploit framework 30
4.2 Tested HTTP exploits for the metasploit framework 30

6.1 Overview of projects and standards for protcol analysis and decoding 49

G.1 Database table outline for signature headers 89
G.2 Database table outline for detailed signature information 90
G.3 Database table outline for protocol-specific message type identifers 90
G.4 Database table outline for vulnerable fields of variable length 91
G.5 Database table outline for Snort signatures 91

Listings

2.1 Configuration of protocol plugins in the `trackerConfig.xml` file 22
4.1 The structures representing requests and replies respectively 35
4.2 A single packet element 37
4.3 Tracker log excerpt 37
4.4 FTP batch script 37
5.1 Extracting the entire set of available information for signature generation 40
6.1 Example of a NetPDL definition for the IP protocol 50
6.2 Example of a PDML description for the IP protocol 51
6.3 Example of a state machine specification for the FTP protocol 53
7.1 Starting an extraction thread 56
7.2 Extracting information and generating signatures 57
C.1 The most important steps in the *Main Process* 72
C.2 Conceptual code excerpt for the function `dispatcher` 73
C.3 Conceptual code excerpt for the function `getLocalThreadIdByPacket` 73
C.4 The code skeleton for the state tracking thread loop `stateTracker` 74
C.5 Program code guideline for the mandatory `entryFunction` 74
F.1 Script for bridge configuration, place it in /etc 87
F.2 Script for removing virtual interfaces, place it in /etc 88
F.3 Script that starts Argos with network capability 88

Chapter 1

Introduction

Security aspects of IT infrastructures became more and more important during the last years due to the increased number of attacks found in the Internet. Attacks can only be successful when a software component has security related *vulnerabilities*. The root causes of vulnerabilities are software bugs, misconfiguration or insiders abusing their credentials and access rights. Software bugs enable attackers to write malicious computer programs which exploit the vulnerable software. Such *software exploits* allow an intruder to gain e.g. privileged access rights on the victim host to evade authentication mechanisms or even execute malicious software on the target machine. Presumably all software available on the market has its bugs and may be vulnerable to attacks. To detect attacks and protect software from attackers different technologies have been invented. This master thesis attends to a specific aspect of such an attack detection technology: the automated signature generation for *intrusion detection systems (IDS)*. The presented work is part of the EU project *NoAH* which aims at the automated signature generation for IDS with the aid of honeypots.

1.1 The trinity of trouble

Today's operating systems for personal computers consist of millions of lines of code (LOC)[1]. Approximately five up to fifty bugs can be expected per one thousand lines of software code. It can be taken for granted that half of these bugs give raise for some security critical issues and can be regarded as vulnerabilities. Those provide the basis for writing software exploits. Even if programs are highly optimized and tested with the newest tools available with respect to security aspects, the number of bugs contained in one thousand lines of code will be in the range of a one-tenth of a percent of the number of LOC. Since software tends to grow even bigger bugs will always be existent and so the vulnerabilities.
A good example for the exploding volume of code in software is the operating system Windows. In 1993 the Windows version NT 3.1 had about six millions LOC. The last Windows NT version 4.0 consisted already of 16 MLOC. The follow-up versions of Windows NT, Windows 2000 and XP had already reached a code size of about 29 and 40 MLOC respectively. The current version Windows Vista has even more than 50 MLOC.
Interconnectivity between computers, especially through the Internet and an increasing number of software programs with an extensible architecture are the two other major contributors to an inconceivable number of present security vulnerabilities in software systems.
The three above mentioned major causes for software vulnerabilities are sometimes referred to as the *trinity of trouble*.

1.2 Intrusion prevention

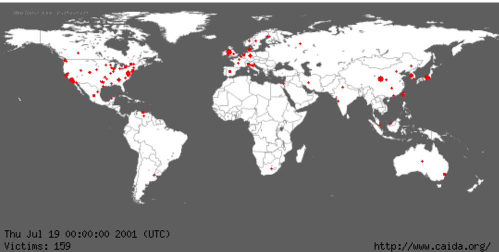
Firewalls, sometimes called security gateways, represent the typical first line of defense to protect against malicious network traffic. Unfortunately, these network components provide only

[1] The background information for this section has been taken from [10].

incomplete protection against malicious traffic by blocking or accepting traffic to certain IP addresses, ports or services. But since access to the services provided by the servers must be possible, a firewall may not be able to protect this server.
Intrusion detection and *prevention systems*, *IDS* and *IPS* respectively, try to provide a more thorough protection. Normally they use some sort of signature. The most common signatures consist of rules how a network packet is (not) allowed to look like in order to distinguish between malicious and benign network traffic. These signatures are thus called *network-based signatures*. Example rules are byte strings which are matched to the application payload of a network packet or regular expressions describing payload character patterns.
Another type of signature are *host-based signatures* where the rules describe certain code execution patterns on the host computer. Examples are system call or control-flow patterns. These patterns make host-based signatures very system-specific and their generation resource intensiv.
On the other hand signatures on the network-level are system-independent and thus easy to deploy. However, a drawback is the high rate of *false positives*. This is due to the fact that network-based signatures rather describe some properties of the network connection or some byte strings contained in the network traffic than the vulnerability itself. It is obvious that a byte string can be contained in benign traffic data too.
To narrow the number of possible matches with benign network traffic and hence to reduce the rate of false positives, signatures based on full *protocol-field knowledge* could be used. This means that a signature is based upon knowledge of the protocol-specific message fields and field-specific properties. But at least as important as the capabilities of the IDS or IPS and its signature description language is the methodology of how attacks are identified and how information from attacks is extracted. Today signatures are mostly still created by some experts. But manual creation will not be useful if exploits for software are available before experts can craft signatures for the corresponding vulnerability.

1.3 Situation in the Internet

In the case of zero-day attacks, the vulnerabilities are recognized only a few hours before or even only at the moment when a novel attack was observed. As the only defense strategy in this case is to react as quickly as possible, manual signature generation will be too slow to contain it and malicious software like e.g. a *computer worm* could have spread to thousands of hosts. Nowadays a trend to more complex and fast spreading attacks can be observed. Computer worms spread itself over the whole world within minutes. These so called *zero-day exploits* can hardly be addressed by manually generated signatures and the fast spreading worms make the manual creation of signatures meaningless. This is where automated exploit signature comes into play. Automated signature generation mechanisms attempt to automatically identify malicious network traffic or software code and generate a specific signature for it. In the best case these signatures resemble the manually crafted signatures denoted in the preceding section. In the past years several projects aimed at the automated generation of exploit signatures. The most important work is presented in chapter 3. To emphasize the ability of fast spreading computer worms, two examples for the spreading of Internet worms are given in Figure 1.1 and 1.2. Both examples show the initial situation and the world-wide distribution of infected hosts for the Code-red and the Slammer worm respectively. The spreading of each worm at a certain time after outbreak is shown on the second pictures.



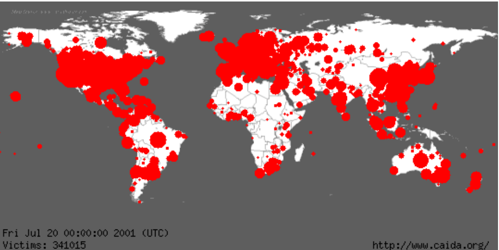
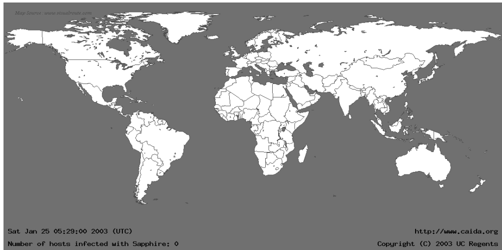


Figure 1.1: The spread of the Internet worm *code red* on July 19, 2001 (within 24 hours)



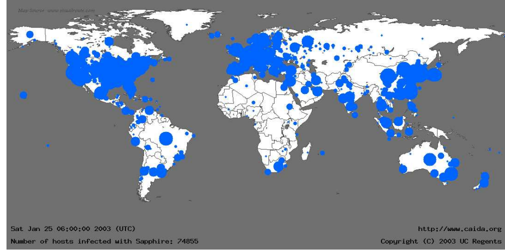


Figure 1.2: The spread of the Internet worm *slammer* on January 29, 2003 (within 30 minutes)

1.4 The Network of Affined Honeypots (NoAH)

The goal of the EU project *NoAH* [25] is to develop an infrastructure that can detect past and future remote exploits in the Internet automatically. Furthermore, upon detection of a new exploit, it should automatically generate a signature for detecting the exploit with non-NoAH systems, e.g. proprietary intrusion detection systems. To accomplish this objective it is planned to reroute traffic towards unused IP addresses. The rerouted traffic will then be handled by a farm of honeypots. These honeypots are normal PC's running emulated operating systems and services like web or FTP servers. The emulating containment system Argos will be able to detect various kinds of attacks and provide precise attack information for e.g. signature generation. The project's aim is a full-scale infrastructure across Europe.
NoAH intends to help limiting damage to national research and education networks (NREN) and to networks of internet service providers (ISP). It will further allow to better assess threats for information security organisations and provide researchers with attack-related data to improve detection techniques.

1.5 Problem statement

The context of this master thesis is the EU project NoAH. As already pointed out the goal of this project is to provide a full-scale infrastructure for the automated generation of signatures for zero-day attacks. This is useful since manually created signatures can hardly protect IT infrastructures from zero-day exploits. In the context of this project an application state tracking framework has been developed at the ETH Zurich during different past student work. This state tracking framework allows to implement application protocol plugins for tracking the state of network connections with full protocol knowledge. With this protocol knowledge it should be possible to reduce the false-positives rate of the signatures generated. As the tracker framework just had plugins for the IP, UDP and TCP protocols, the first task of this thesis was to work on the implementation of an application plugin for the Tracker and to improve the Tracker software. The state information should then be used as the basis for a signature generation mechanism which creates network-based signatures.

1.5.1 Application plugin for the tracker framework

For being able to use various protocol field and state information for signature generation an application protocol plugin was required. To accomplish this task various protocols had to be examined and compared with each other. The specifications should then serve as the basis for the implementation of a packet decoder and state machine for the selected protocol. Performance tests on the one hand and security related tests like memory profiling on the other hand should improve the plugin and make it ready for long-term usage.

1.5.2 Signature Generation Mechanism

The signature generation mechanism should extract information from the containment system and the tracker log files and create information on a meta-level. In a second step these meta

information has to be converted into a specific signature format compliant with an existing intrusion detection system. The two popular and freely available open source IDS Bro and Snort were proposed as an alternative.
Extracting attack information from both the tracker log files and the containment system poses a difficult task. Argos provides state information of the system at the time when the attack was detected. Provided information includes a memory and register dump and if possible network packet data that is related to the attack. The tracker framework additionally provides knowledge about the protocol fields and the connection state of the network packets. The challenge is to create meaningful signatures from this rather unstructured information. As not for each attack the entire set of information is available, signatures can not always be created the same way. Furthermore it is possible that the generated signatures heavily differ in accuracy. This is a major problem and complicates the design of a signature generator.

1.6 Structure of the Report

In the next chapter the architecture of the NoAH project is presented and how the project components are meant to interact. The chapter 3 delineates the different methodologies to identify attacks and extract exploit information by means of presenting related research work. Chapter 4 describes the first task of this thesis, namely the design and implementation of an application state tracking plugin for the tracker framework. The follow-up chapters deal with signature generation. In *Signature Generator* a simple first signature generation mechanism is presented whereas the chapters 6 and 7 describe a generic approach for signature creation with full protocol knowledge. The ideas documented therein evolved from experiences made during the design and implementation of a first solution as a proof of concept. The report is closed by a chapter comparing the results of this work with approaches presented in the *related work* chapter. Furthermore the findings are summarized and an outlook is given.
The appendix presents various additional information. In appendix C the tracker framework is explained in details. Appendix F provides installation advices for the software accompanying this master thesis. At the end of the report a list of abbreviations explains the meanings of the abbreviations used throughout this documentation. The following index lists technical terms and shows the page numbers where these terms have been introduced. Normally for each term first the page number of the position in the text where the term occurs is printed. The last page number given for an index entry references the corresponding entry in the glossary if existent. The glossary can be found in the appendix B and provides the reader with information for selected technical terms.

Chapter 2

Current state and Setup of the NoAH project

2.1 NoAH architecture overview

The architecture of NoAH mainly consists of two parts. A first part is responsible for redirecting network traffic towards unused IP addresses to a farm of honeypots. The second part accounts for filtering the redirected traffic and detecting possible attacks. After the detection, the detectors provide information for attack signature generation. Figure 2.1 illustrates the basic setup of the NoAH architecture. The second part is called the ”NoAH core”. The hosts from which mali-

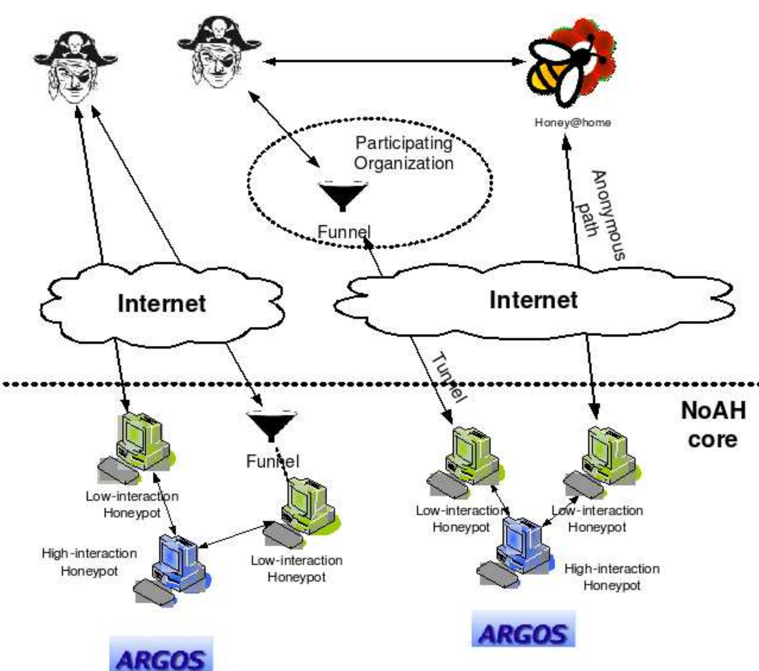


Figure 2.1: The core components of the NoAH architecture

cious network traffic originates are pictured as pirates. Traffic will be redirected by two different mechanisms:

- If an organization decides to participate in the NoAH project unused IP addresses of the organization’s address space are statically redirected to a network of honeypots.
- The above method is limited to larger enterprises reserving a range of IP addresses. *Honey@home* client’s have the honey@home client running. This small program can be used to redirect traffic directed towards unused IP addresses of home networks, also

known as *SOHO's*. This way NoAH can be brought to home users. The tool needs no configuration and is easy to install. It runs both on Windows and Linux.

Redirected network traffic passes *low-interaction honeypots* in the NoAH core first. Low-interaction honeypots emulate services using scripts. The lightweight processes are able to cover a large network space but such emulation can not provide a high level of interaction with possible attackers. One of the most popular and widely-used low-interaction honeypot is *honeyd*. It emulates thousands of IP addresses and performs network stack emulation. The honeypots are highly configurable and lightweight. They are an efficient mechanism to filter out unestablished and uninteresting connections like port scans, SSH brute-force attacks etc. This is a desired effect since the second-level honeypots, the *high-interaction honeypots*, are responsible for detecting special kind of attacks. The high-interaction honeypots are entirely emulated server systems by the honeypot containment system Argos. This latter type of honeypot is closely related to this work as Argos is the system that detects attacks and provides us with information for signature generation. It is described in more detail in the next sections.

2.2 Components related to this work

In Figure 2.2 the part of the NoAH architecture on which this work is based is depicted. The upper part in the illustration, the honeypot containment system, is developed at the Vrije Universiteit (VU) Amsterdam. The application state tracking framework has been developed in past semester theses at the ETH Zurich. The basic functionality of the interface which is to connect both the information from the containment system and the Tracker with each other has been implemented too. A possible signature generation mechanism would receive its information from

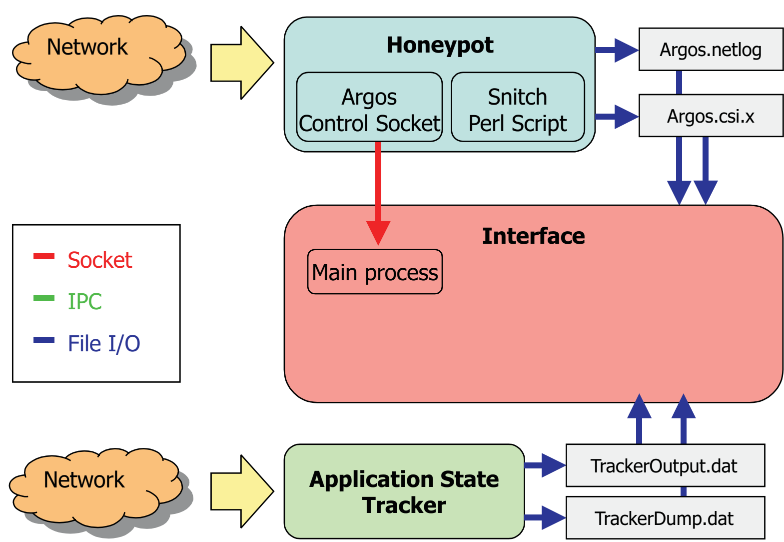


Figure 2.2: The functioning components of the NoAH project which are related to this work

the interface component. The subsequent sections will describe the two components serving as an information source for the signature generation component.

2.3 Containment system Argos

The containment system *Argos* [8] aims at the detection of remote code execution exploits. This means that some malicious code is injected remotely via network traffic and will then be executed. The executed machine code should not be confused with sequences of interpreted instructions like e.g. Java bytecode. Argos is a "secure" system emulator and meant to be used in the context of honeypots. It is based on *Qemu* [26], an open source emulator that uses dynamic translation to achieve good performance.
Argos extends Qemu to enable it to detect remote attempts to compromise the guest operating

system. Using *dynamic taint analysis (DTA)* it tracks network data in the (emulated) memory of the guest operating system and detects any attempts to execute it as part of a program. When an attack is detected the memory footprint of the attack is logged and written to a log file.

2.4 The tracker framework at a glance

In this section, the tracker framework is presented shortly. For a more detailed view on the Tracker and e.g. its algorithms, refer to appendix C. The installation and configuration of the Tracker is explained in the manual in appendix F.
The tracker framework captures network packets and processes them with the appropriate network protocol plugin if available. Each plugin can add some reporting information to a buffer which is forwarded to the reporting thread when the entire plugin stack is processed. An arbitrary number of threads can be configured which are responsible for packet processing. The basic concept is depicted in Figure 2.3. The network capturing is done by the pcap library [18].

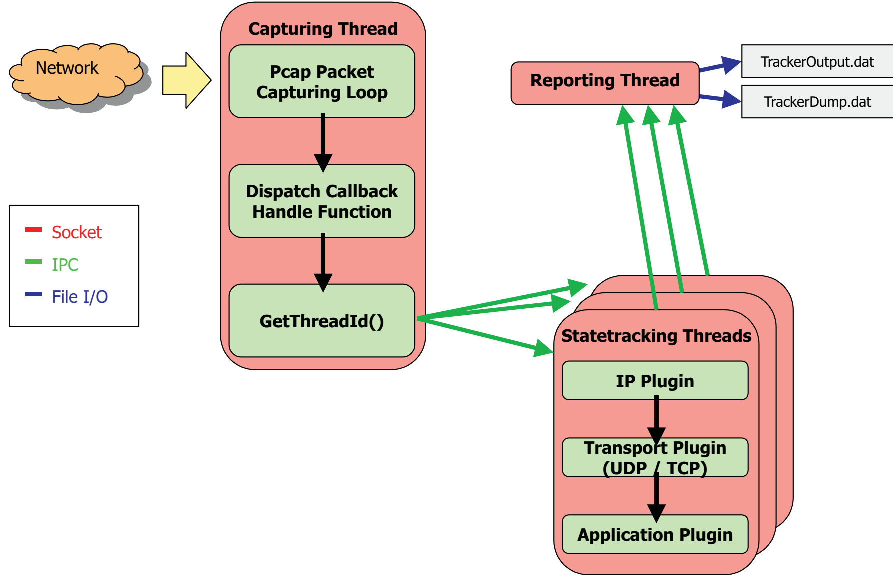


Figure 2.3: The basic design of the tracker framework

When the capturing loop receives a packet, it calls a callback function named `dispatch()`. This function has to determine if the packet belongs to a connection which is already registered in the Tracker and thus belongs to a certain thread or if it has to create a new connection entry for the packet and assign a new state-tracking thread. This mechanism is necessary since each state-tracking thread has its own plugin instances and thus only the appropriate plugin can access stored state information whereas the other plugins would not have information about this certain connection.
A plugin is written by implementing the following functions:

- `extern "C" int initFunction(stateTrackerThread*, u_int32_t);`

In this function dynamic allocated structures etc. should be initialized. The function will be called at the startup of the Tracker when the plugins are loaded.

- `extern "C" int shutdownFunction(void* );`

This function should free dynamically allocated memory. It is called when the Tracker gets terminated.

- `extern "C" int entryFunction(u_char*, u_int32_t, void*, protocol*, stateTrackerThread*, reportbufferEntry*, packetInfo*);`

Each time a captured packet is delivered to a certain thread and belongs to the corresponding protocol, this function is called. In this function the implementation of the corresponding protocol

state machine should take place amongst other things.

A typical sequence of actions to be taken in the function `entryFunction` could be:

1. Extract information of the preceding protocol which invoked this function. For this purpose the `void*` pointer can be used to submit arbitrary data structures.
2. Process the payload according to the specification of the current protocol.
3. Create a report of extracted information, especially for state changes.
4. If there is an appropriate next protocol, call its entry function.

As mentioned in the appendix F.1 an XML file is used to configure properties of the Tracker. The plugins are registered and intertwined by declaring the plugins as `<protocol>` elements and the dependencies in form of subelements called `<subProtocol>`. The listing 2.1 shows an excerpt of the `trackerConfig.xml` file and how the protocols are connected with each other by referencing them via `<subProtocol>` elements.

Listing 2.1: Configuration of protocol plugins in the `trackerConfig.xml` file

```
<protocol id="ip">
    <library>plug-in/IPTracker/libIPTracker.so</library>
    <subProtocol protocolRef="tcp" linkage="6" />
    <subProtocol protocolRef="udp" linkage="17" />
    <description>Internet Protocol</description>
</protocol>

<protocol id="udp">
    <library>plug-in/UDPTracker/libUDPTracker.so</library>
    <description>User Datagram Protocol</description>
</protocol>
```

Chapter 3

Related Work

In [30] twelve approaches for automated signature generation are presented and compared against each other. However a classification of the signature generation methods has not been done. In [2] more or less the same signature generation methods as in the already mentioned work above are described. A classification of the approaches was done by sorting them according to the attack detection method they use, e.g. approaches without attack detection or using network-level attack detection. Because the type of attack detection itself is not relevant anymore due to Argos this work groups the signature generation methods with respect to the information the signature generation mechanisms use.
The quality of a signature generation method depends not only on the methods used for identifying the relevant content and the algorithms for creating a signature out of this information. An at least equal important aspect is what kind of traffic is used for analyzing payloads and extracting signatures. If a system observes the entire network traffic and can not be sure that a certain network packet really belongs to malign traffic it is more difficult to obtain a low false-positives rate than if a system uses attack detection. In the latter case mainly malign traffic would be used for signature generation and this tends to result in a decreased false-positives rate. This is why in [2] the signature generation approaches are sorted according to the type of attack detection that is used prior to signature generation.
The following sections present existing approaches for automated signature generation, sorted by the type of information used for the signature. For a more thorough description of these approaches refer to the literature. In the appendix D different techniques for typical problems and suitable algorithms are presented, which are or may be used for signature generation. The appendix E provides some signature examples to give an idea of how signatures can look like.

3.1 Content-based signatures

Content-based signatures are used widely by automated signature generation systems. The signature is a tuple

(IP protocol number[1], destination port, byte sequence)

where the byte sequence is a variable-length, fixed sequence of bytes. The main problem is to identify a characteristic string within the payload of network packets. Afterwards string matching rules can be generated. An IDS would just compare each incoming network packet against the string and if the string can be found in the payload the packet would be treated as malign. The reason why this approach is often applied may be that it is rather easy to implement and it is fast. The main problem with this string-based approach is that when any words inside the relevant string of a payload are changed the IDS won't recognize the exploit anymore by just matching the complete string against the payload. There have been various extensions probosed of how to find relevant substrings for a signature describing a certain exploit and so to diminish the probability of the above-mentioned problem with altered strings.

[1] http://www.iana.org/assignments/protocol-numbers

3.1.1 Honeycomb

Overview Honeycomb [5] is built as an extension to *honeyd* which is a low-interaction honeypot. Network traffic seen at a honeypot is used as input for the signature generation algorithm. Thus no distinction between benign and malicious traffic is made. The signature output can be converted into Bro or Snort signature format.

Signature Generation A signature is generated on two levels. On the first level stream assembling is applied to the packets. For TCP, streams are saved as messages collected according to their direction, as TCP connections are bidirectional. In case of UDP messages are saved as unidirectional. Then an *analysis signature* is created which is based on anomalies recorded by protocol analysis at network and transport layer. There is no knowledge of application layer protocols. In a second stage similar recorded connections are compared against each other by a two-dimensional *LCS algorithm* and payload byte patterns found are added to the signature. Horizontal detection means that a message is compared with each message of all the other connections corresponding to the same state. Vertical detection is done by concatenating messages of one connection and comparing it to concatenated messages of similar other connections. This allows to detect patterns in interactive sessions in contrast to the horizontal detection. For an example of a honeycomb signature look at section E.1.

Signature Refinement Honeycomb maintains a signature pool. When a new signature is a superset of an existing one, the old signature is dropped. Dropped signatures are not lost because the pool is backed up regularly. If the new signature does not replace an existing one it is just added to the pool. Signatures equal to existing ones are discarded.

3.1.2 Polygraph

Overview Polygraph [14] is a system specifically targeted at generating signatures for *polymorphic worms*. A flow classifier reassembles *flows* per port and puts them in a suspicious flow pool. The classifier is not specified.

Signature Generation There are three different signature types specifically designed to detect polymorphic worms. All three signatures are built from substrings called tokens which are byte sequences longer than a certain minimum length. Frequent tokens are collected and each flow is represented by a number of tokens. The grouping of tokens can be achieved by hierarchical clustering.
The *conjunction signature* is a set of unordered tokens. If a signature and an incoming flow have the same tokens they are equal. For a set of tokenized flows a conjunction signature can be generated by simply extracting the tokens present in all flows.
The *token-subsequence signature* is a set of ordered tokens. A flow matches a signature if it contains the same set of tokens in the same order as the signature. A common token-subsequence signature for a set of flows is found by applying the *longest common subsequence (LCSeq)* problem to all signatures for the flows contained in this set.
The *Bayes signature* is a set of tokens with a score assigned to each of them. A flow is matched against this signature by adding up the scores of tokens, which are present in the flow. If the sum is greater than a threshold it is a worm. The score is derived from the *Bayes law*. It is mainly based on the probability that a certain token is present in a worm.

Signature Refinement Signatures are a set of tokens describing a certain exploit. They are refined by *hierarchical clustering*.

3.1.3 Earlybird

Overview Earlybird [31] proposes an automated approach for quickly detecting previously unknown worms and *viruses*. The main idea of the approach is to compute a *Rabin fingerprint* for all possible substrings of an incoming network packet. Each fingerprint is hashed together with the destination port and protocol. The hashes serve as indexes in a so-called content prevalence table which counts the number of occurences for a certain hash value. A second table called the address dispersion table counts the number of similar IP source

and destination addresses for each hash value. Sorting the prevalence table with respect to the substring counters and taking into account the size of the corresponding entries in the dispersion table one gets a set of likely worm traffic. This approach is called content sifting.

Signature Generation The system generates pattern-matching signatures formatted for the Snort intrusion detection system including transport protocol and port information. As the content-sifting algorithm does not keep any per-flow state the generated signatures describe only content information contained within a single packet. An example of an early-bird signature can be found in section E.2.

3.1.4 Autograph

Overview Autograph [9] is a system for automated generation of worm signatures. The system maintains a suspicious flow pool for which TCP flows are reassembled. If the number of flows for a specific destination port exceeds a threshold the signature generation process is initiated. Autograph measures the frequency with which non-overlapping payload substrings occur across all suspicous flow payloads and proposes the most frequently occurring substrings as signature candidates. This is done by the *content-based payload partitioning (COPP)* algorithm.

Signature Generation The above signature candidates are filtered for possible benign content. In a repetitive process the most prevalent content block is selected as signature. This process repeats with the remaining flows, until some fraction of all flows in the pool has been covered. Finally a set of selected signatures can be formatted as Bro signatures.

3.1.5 TaintCheck

Overview TaintCheck [13] uses dynamic taint analysis to protect designated applications. Whenever tainted data is used in a way that is disallowed by the installed policy, TaintCheck generates an alert and launches the signature generation process. The output is a three byte long string signature.

Signature Generation The three bytes of the signature are determined by matching the most significant bytes which were used to overwrite a return address or a function pointer with the originial content, i.e. network data is compared against memory data. If the original content and the three bytes do not match, some decoding or other data transformation operations have been applied between data input and attack detection. In this case the original content is used as a signature. The signature length of three bytes is too short because assuming an uniform byte distribution would lead on average to one false positive per 16 MB of traffic.

3.2 Flexible content-based signatures

These approaches on the one hand work on a byte level as the methods in the previous section. But on the other hand they are more flexible in that they do not just try to match strings or substrings with incoming packets. Their signatures describe patterns of how malicious bytes are organized. Example techniques are the use of byte-frequency distributions or regular expressions.

3.2.1 PADS

Overview The position-aware distribution signatures (PADS) system [36] uses a double-honeypot system to track malicious activities in local networks. A high-interaction honeypot redirects connections to low-interaction honeypots. When the high-interaction honeypot gets compromised, the low-interaction honeypots will be able to capture several worm variants. Worm signatures are computed off-line.

Signature Generation The PADS signature consists of a signature for normal and anomalous traffic. Both signatures contain byte frequency distributions (instead of fixed values) for each position in the signature string[2]. Signature generation and worm identification in a payload are very related. To understand the signature generation process the worm identification process needs to be understood.
To examine a byte sequence for the existence of a worm, a window of the length of a PADS signature slides over the byte sequence and computes a matching score Δ for every window W. This matching score is a formula that involves computing a matching score M for both the byte sequence with the anomalous and the normal signature. If the matching score Δ for a window position is higher than a threshold (which is normally zero), the sequence is assumed to carry a worm.
The position of the window with the highest matching score is called significant region. For finding a worm signature we need this region because the anomalous signature is the *byte-frequency distribution (BFD)* of the significiant regions of all worm variants at hand during the signature generation process. If the significant regions of all worm variants were known the BFD could be easily computed and so the signature. But to know the significant regions we would have to know the signatures which we don't. Fortunately we can approximate the significant regions for all the worms by applying the *expectation-maximization (EM) algorithm*.

3.2.2 PAYL

Overview PAYL [16] is an anomaly detection sensor that detects inbound anomalous loads, and correlates them with outgoing traffic on the same ports.

Signature Generation The PAYL anomaly detection sensor computes during a training phase the "normal profile" of a site using *n-grams*. For a packet payload, an n-gram consists of any sequence of n consequent bytes in the payload. When a new packet arrives, all possible n-grams are computed for it and also the frequencies of these n-grams are registered. Then a formula is used to compute the distance between arriving packets and the n-gram distribution, which was seen during the training phase. If this distance is larger than a threshold and the incoming traffic was intended for port i, then such packets are put into a buffer list of "suspects" for port i. Any outbound traffic to port i, which is also detected as anomalous using the anomaly detection sensor, is compared with this buffer. For the compared strings, a similarity score is computed based on a formula, which requires the generation of the longest common substring (LCS) and the longest common subsequence (LCSeq) of the two strings. If the similarity score is greater than a threshold, the outgoing traffic is blocked. As a by-product of the correlation between inbound and outbound traffic, a signature for the worm is generated in the form of a LCS and a LCSeq.

3.3 Context and semantics aware signatures

These approaches go beyond the previous approaches and the byte level analyzation respectively. They understand application-level protocols and thus can determine in which states an application has to be for unveiling an exploit.

3.3.1 Nemean

Overview Nemean [35] provides automatic generation of intrusion signatures from honeypot packet traces. The system consists of the Data Abstraction Component and the Signature Generation Component. The Data Abstraction Component normalizes packets and performs flow aggregation by ordering the packets into connections (multiple packets between two hosts) and sessions (multiple connections between two hosts). At last the aggregated sessions are normalized via pre-defined service specifications, e.g. for HTTP. The output of this component is a semi-structured session tree.

[2]The byte frequency distribution of normal traffic is equal for all bytes.

Signature Generation The Signature Generation Component groups sessions and connections according to a similarity metric. Automata learning is used to construct an attack signature from a cluster of sessions or connections and these Finite-State-Automata signatures are transformed into a signature format of an IDS.

Signature Refinement As clustering is used to group session and connection information to signatures, refinement could be done by successively applying the clustering algorithm.

3.3.2 COVERS

Overview The COntext-based, VulnERability-oriented Signature (COVERS) [37] system allows to automatically generate attack signatures for control flow hijacking attacks. It consists of an attack detection part and a signature generation part. The attack detection part employs the *address space randomization (ASR)* technique in contrast to e.g. Argos which uses dynamic taint analysis.

Signature Generation Signature generation consists of three steps. The correlation step identifies the specific network packet (or flow) involved in an attack, and the bytes within this packet that were responsible for triggering the alert. To identify the relevant bytes a forensic analysis of the victim process memory around the corrupted pointer is done by using the LCS method on recent input data and the data held in memory. However there are some cases where this approach will not produce meaningful signatures.
In a second step application protocol fields are analyzed whereas a language for simple message format specifications has been developed. After the field is identified by the specification, abnormal field characteristics are identified by comparing the field against reference values which are continuously updated using benign input traffic.
The signature consists of the message format specifier, the message field carrying the exploit and thresholds for the characteristics. An example can be found in the section E.3.

3.3.3 Polymorphic Worm Detection using structural Information of Executables

Overview In [6] an approach is presented to generate signatures for detecting polymorphic worms. It is based on the control flow graph (CFG) of executable code. First a linear disassembler extracts a sequence of valid instructions. Then a CFG is created for which a spanning tree is calculated. From this all possible k-node subtrees with a selected basic block as root node are generated. These trees also include non-spanning-tree links. The adjacency matrix of each tree is combined with node colors (14-bit vectors) which provide an indication of the instructions in a basic block. Out of the matrix a fingerprint is computed.
The detection part is very similar to the Earlybird approach (see subsection 3.1.3). The main difference is the mechanism used to index the prevalence table. While Earlybird uses simple substrings, this approach uses fingerprints extracted from CFGs. Thus worms are identified by checking for frequently occurring executable regions that have the same structure.

3.4 Other approaches

In this section approaches are presented which can not be classified as signature generation mechanisms (SGMs). Nevertheless these methods describe interesting ideas which could be used in a future SGM.

3.4.1 DOME

Overview The Detection Of Malicious Executables (DOME) [12] approach allows to detect code injection attacks and attacks originating from executables with modified code. The detection mechanism is based on the fact that malicious code often makes use of system calls. DOME makes a static analysis of the executable of an application to identify the location

of system calls in it and supervises if the locations at runtime differ. Although DOME does not generate any signature describing malicious activities, it can recognize such activities by verifying any activity against a signature describing normal/approved behavior.

3.4.2 Paid

Overview The Program semantics-Aware Intrusion Detection system (PAID) [17] defends applications against control flow hijacking attacks that make use of system calls. It analyses the location and ordering of system calls and parts of the control flow of the application with the aid of the application's source code.

Signature Generation The recompilation step analyses the system call usage of an application's source code and constructs a System Call Site Flow Graph (SCSFG), which is included in the resulting library or executable. This graph is a deterministic finite-state automaton (DFA) representing the system call sequences and their location (site) in the program.

3.4.3 HoneyStat

Overview HoneyStat [7] is a system which combines network and host level attack detection methods. A HoneyStat node emulates multiple operating systems and detects three different types of events: memory events, network events, and disk events. The system does not generate any signatures from the generated events but information during an event is recorded. The gathered information is forwarded to a central analysis node which correlates all received HoneyStat events.

3.4.4 Vigilante

Overview Vigilante [20] is an end-to-end approach to contain fast spreading worms using collaborative worm detection at end hosts. The system introduces the concept of self-certifying alerts (SCAs). A SCA contains a description of an attack that is detailed enough, to allow other hosts to verify, if they are vulnerable to it. This is done by replaying the message(s) of a SCA in a sandboxed version of the targeted service. This SCA verificator replaces the section that is marked as critical with a nonce. If the nonce is activated and thus the host is vulnerable, it could generate a protection filter for the corresponding attack. SCAs have been developed for three common vulnerabilities. Arbitrary Execution Control (AEC) SCAs identify vulnerabilities that allow worms to redirect execution to arbitrary pieces of code. Arbitrary Code Execution (ACE) SCAs describe code-injection vulnerabilities. Arbitrary Function Argument (AFA) SCAs identify data-injection vulnerabilities that allow worms to change the value of arguments to critical functions such as the `exec` system call.

Signature Generation The three types of SCAs have a common format: an identification of the vulnerable service, an identification of the alert type, verification information to aid alert verification, and a sequence of messages with the network endpoints that they must be sent to during verification. The verification information allows the verifier to craft an exploit whose success it can verify unequivocally. It is different for the different types of alerts, e.g. for AEC an SCA specifies where to put the address of the nonce code to execute in the sequence of messages. In the section E.4 an example of an AEC SCA is given.

Chapter 4

Application Protocol Plugin

4.1 Requirements

As NoAH is primarily meant to detect exploits for server software the application protocol to be implemented in the tracker should be a client/server application protocol, e.g. the HTTP protocol. Because in the second task state information of this application protocol plugin will be used for exploit signature generation, exploits for this protocol have to be available. So prior to the design and the implementation of the plugin a survey of available exploits in relation to the corresponding network protocols had to be done. During the design of the plugin another important requirement emerged. Several network protocols use multiple connections in parallel for the hosts communicating with each other, e.g. one connection for exchanging control commands and an arbitrary number of connections for exchanging data as it is the case for the FTP protocol. In this case it could be possible that a network packet belonging to a data connection arrives at the Tracker for processing before the data connection initiating control packet arrives. In this case of possibly asynchronously arriving network packets the Tracker has to be able to collect packets although they do not actually belong to one of the implemented protocol plugins. The section about the non-protocol specific *gargabe collector* plugin devotes to this aspect.

4.2 Exploit survey and Protocol Evaluation

Software vulnerabilities do neither depend on the purpose for which the software was written nor on which target platform or operating system it will run. Vulnerabilities rather depend on the development process and the technologies used throughout. Consequentially the number of occurring vulnerabilities in network software and thus possible exploits varies for software products and not for protocols. Testing the signature generation mechanism implied to have several successfully executable exploits for the same protocol. It has shown that the number of available exploits for the same application protocol differs but remains very low. Even if an executable exploit is available, it is not sure that the vulnerable version of the target server software can be found. And even in this case, the success of an exploit depends on the vulnerability of the overall target system. For instance the success of an exploit for server software on Linux may vary for each distribution and version. This is the reason why Windows 2000 was selected as the primary target operating system to attack. Furthermore Windows 2000 is more susceptible to remote exploits than its widely used successor Windows XP.
Because exploiting software is a very complex and cumbersome craft it has been decided to use the Metasploit framework [28] for attacking the containment system. Logically even less exploits are available for this attack framework. For the evaluation of the most suitable application network protocol which will be implemented in form of a plugin, different exploits in the Metasploit framework and the corresponding exploitable software had been searched and tested. The tables 4.1 and 4.2 list the most important and usable exploits available for FTP and HTTP protocol respectively. The tests have been done with target machines running Windows 2000 (Win2k), both english and german version, and Windows XP. A test was either successful ($\surd$) or it failed (X). The fields marked with an asterisk represent exploits which triggered the vulnerability, e.g. the buffer overflow, but could not execute the shell code successfully.

Exploit name	Description	OS		
		Win2kDE	Win2kEN	WinXP
warftpd_165_user	Uses overflow in the USER command.	$\surd$	*	$\surd$
warftpd_165_pass	Uses overflow in the PASS command.	*	*	*
3com_3cdaemon_ftp_overflow	Overflow via the USER command.	*	*	$\surd$
slimftpd_list_concat	LIST command with overly-long argument triggers overflow.	X	X	*
cesarftp_mkd	Improper input validation allows code execution via overflow.	*	X	*

Table 4.1: Tested FTP exploits for the metasploit framework

Exploit name	Description	OS		
		Win2kDE	Win2kEN	WinXP
trackercam_phparg_overflow	Stack overflow via PHP arguments.	X	*	*
minishare_get_overflow	Buffer overflow via missing Link length validation.	*	*	$\surd$
apache_chunked_win32	Due to improper interpretation of an unsigned value, buffer sizes for requests with chunked encoding are computed wrong.	X	X	X
icecast_header	Sequence of 32 http headers will overrun a buffer.	$\surd$	$\surd$	$\surd$
ypops_smtp	Too long smtp message overflows a buffer.	*	*	*

Table 4.2: Tested HTTP exploits for the metasploit framework

Server applications for the file transfer protocol (FTP) exhibit a large number of similar buffer overflow vulnerabilities. Mostly the buffer overflow can be triggered in an argument field of a command request message. The kind of protocol-specific and typical buffer overflows for the FTP protocol, and the amount of available and successfully executable exploits for different FTP server applications led to a more thorough examination of the protocol specification of FTP. It is important to emphasize that from all examined protocols FTP is the one with the highest number of similar exploits for distinct server applications. Further there was a high percentage of vulnerable versions of server applications available with respect to the available exploits.

4.3 The File Transfer Protocol (FTP)

The FTP protocol sits on top of the transport layer in the *OSI model* and uses the transport control protocol (TCP). It is commonly used for exchanging files over a TCP/IP network. The main specification part can be found in the request for comment (RFC) document 959.
FTP consists of two connections. The control connection normally is connected to the server port 21 and initiates an arbitrarily number of data connections. The connection ports for the data connections depend on the mode which was used to negotiate the data connection. In active mode the client creates a listening socket and tells the server via the PORT command on which port it is available. As incoming connections are often refused when the client is behind a firewall and/or a *network address translation (NAT)* box, the passive mode is mostly used in the Internet. In the passive mode the client sends a PASV command to tell the server to open a listening socket on a certain port. This port is normally the port number of the control connection on the server side minus one.
FTP is based on the *Telnet protocol*. Commands are sent in cleartext and normally for each command a reply is awaited. It consists of a three digit reply code and some human readable

reply message. An example of a command specification taken from RFC 959 is given below. All commands are defined in Backus-Naur-Form (BNF) notation.
Example Definition of the USER command in FTP:

```
USER <SP> <username> <CRLF>
<username> ::= <string>
```

A typical FTP command / reply sequence would be:

```
> 220 Service ready for new user.
> USER anonymous
> 331 User name okay, need password.
> PASS anonymous@mozilla.org
> 230 User logged in, proceed.
```

The relation of data and control connections became of interest during the plugin implementation, since the tracker has to temporarily store possible data connections for the case where the data connection negotiation command would be processed delayed. This led to a second plugin called the garbage collector.

4.4 The Garbage Collector Plugin

As shown in [21], all packets belonging to a certain network connection have to be processed by the same thread, since state information for this connection is only available in the thread which received the connection initiating packet. If for some reason the FTP command for negotiating a data connection would be buffered and the data connection initiating packet would be processed first, it may be possible that this packet will be delivered to the wrong thread. Actually it will not be even delivered since we can not assign a port number for the data connections statically because the port numbers are negotiated in most cases for the FTP protocol, as we have seen before. Then the tracker would discard the packets and our plugin never notices that the data connection was indeed opened. It may be even the case for more complicated protocols such as peer-to-peer protocols that not even a single connection could be statically preconfigured since all ports are negotiated. These problems made a plugin necessary which buffers all packets belonging to a connection not assigned to a plugin. Note that the last scenario when we are not even able to preconfigure a single connection with respect to the port assignments is not solved by this plugin. However it would be easily possible to augment functionality of this plugin, e.g. by a polling mechanism, in order to come by this most complicated case.
The most important structure in this plugin is the class `SynchronizedMap` which holds all packets sorted according to the connections they belong to. In order not to overflow heap memory, packets are deleted in different situations. So what the garbage collector plugin actually does, is just to insert arriving packets into the synchronized map which is described in the following subsection.

4.4.1 Class SynchronizedMap

The class `SynchronizedMap` consists of a map called `contentMap` which holds connection information such as state changes sorted according to the connection keys for each registered network packet. The number of maximum opened connections is limited by `MAX_OPEN_CONNECTIONS`. Connections which get closed are registered in a sub-map called `closedMap` which sorts connections according to the time when they were closed. If `CARE_CLOSED_INTERVAL` number of elements have been inserted into `closedMap`, closed connections are removed from all internal maps (`contentMap` and `closedMap`) if they have been closed for longer than `CARE_CLOSED_TIME`. The connection holding map has an upper bound number of contained elements of `MAX_HASH_ENTRIES` and each opened connection must not have more than `MAX_ENTRIES_PERKEY`. Note that according to this condition it follows that

```
MAX_OPEN_CONNECTIONS * MAX_ENTRIES_PERKEY <= MAX_HASH_ENTRIES
```

But what happens if we had the maximum number of opened connections and each opened connection has the maximum number of entries? The map would be blocked and a new connection could not be added. A workaround is the following. We allow to set the constants in a way that the above inequality is hurt. But if we reach a number of connection entries bigger than `MAX_OPEN_CONNECTIONS * MAX_ENTRIES_PERKEY` we switch to congested mode. In this mode we will force first the deletion of closed connections until we reach a limit number of `NUM_DEL_FORCED`. If this is not possible it would be thinkable that we even remove entries for opened connections. As it becomes clear from the above explanations such a map is a complicated thing and needs a lot of fine tuning before it is used in a real-world scenario with a high amount of network traffic. Following the above mentioned maps for opened and closed connections are shown as they are declared in the private section of the class `SynchronizedMap`.

- **`multimap<unknownConnectionKey, unknownConnectionInformation*>* contentMap;`**

The content map just maps information on a certain connection. Each arriving packet will put its information into this map in form of a map entry. This method allows us to have information for connections with the same end points but which chronologically differ in the same map. This is important because it is theoretically possible that two consecutive connections between two hosts could have the same port assignments but the tracker was not able to process information about the first connection when the second connection will be stored in the map. The map in this case would mark the first connection as closed and then it will be possible to insert a new connection with the same end points again.

- **`map<closedTimeval, unknownConnectionKey>* closedMap;`**

Each connection which will be closed gets an entry in this map. The keys are sorted according to the time when the connection was closed. This enables us to only delete the oldest connections.

- **`sem_t*mapCompleteLock;`**

This is a semaphore which controls the access to the maps above. The maps can only be accessed via certain API functions and those are made thread safe by use of this semaphore.

Class unknownConnectionKey

The class `unknownConnectionKey` describes an arbitrary connection to the server which is monitored. This means that if we would track the connection states to more than one containment system, we would have to add additional information to this class. However it was assumed that monitoring one containment system is sufficient. A connection key has the following properties:

Transport Layer Port This number corresponds to the port taken from the IP frame and e.g. stands for the TCP protocol.

Client Network Address A network address can be currently either IP version 4 (IPv4) or version 6 (IPv6). Further types of network layer addresses could be added.

Client Port The client port of the transport layer protocol, e.g. port 21 for the FTP protocol if TCP is the transport layer protocol.

Server Port The port of the containment system.

As this class will be used in containers from the Standard Template Library (STL), the less-than-operator is overloaded. In the case of FTP it is possible that we only know one application port, either the one of the server or the client. This comes from the fact that either the client opens a listening socket (active mode) or it tells the server to open a listening socket for a certain port (passive mode). Either way only one port will be known beforehand. This is why a port set to `SMAP_UNKNOWN_PORT` will be ignored when two connection keys are compared against each other.

4.5 The FTP Plugin

4.5.1 Deriving a state machine for connection state observation

The FTP protocol defines five different state machines for request / reply sequences. Each request command is assigned to one of these state machines. The most commonly applied state machines are shown in Figure 4.1 and 4.2. For each state machine the corresponding request commands are listed. Note that the reply codes shown besides the transitions denote the first digit of the actual reply codes. This is because each of the three digits in a reply code is subject to some classification. Please refer to the original protocol specification (RFC 959, page 34) for more details.

State machine one

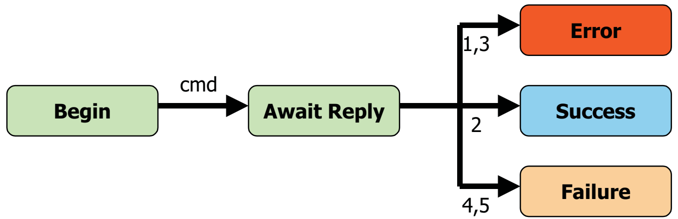


Figure 4.1: First state machine of the FTP protocol specification

Assigned request commands:
ABOR, ALLO, DELE, CWD, CDUP, SMNT, HELP, MODE, NOOP, PASV, QUIT, SITE, PORT, SYST, STAT, RMD, MKD, PWD, STRU, and TYPE.

State machine two

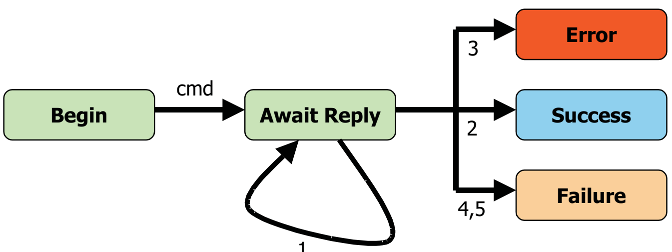


Figure 4.2: Second state machine of the FTP protocol specification

Assigned request commands:
APPE, LIST, NLST, REIN, RETR, STOR, and STOU.

Additional state machines

For some special commands more specific and complex state machines exist which can be seen as concatenations of the above two types of state machines for two or three commands. In particular:

- The sequence of the RNFR command followed by the RNTO command describes its own state machine.
- The restart command REST has its own state machine and will be followed by either APPE, STOR or RETR.

- The login sequence with the commands USER, PASS and ACCT is the most complicated state machine and describes state transitions between those three commands whereas PASS and ACCT can be seen as optional.

A derived state machine

In order to be able to track the state of a connection we have to know exactly in which overall state the connection is. This implies that we must have a single state machine which can be used throughout the entire life cycle of the connection. There will be only one single entry point and one single state when the connection gets closed. The Figure 4.3 shows the derived state machine. Note that the properties of each of the five afore described state machines are somehow contained in this new state machine. After each received reply message the state

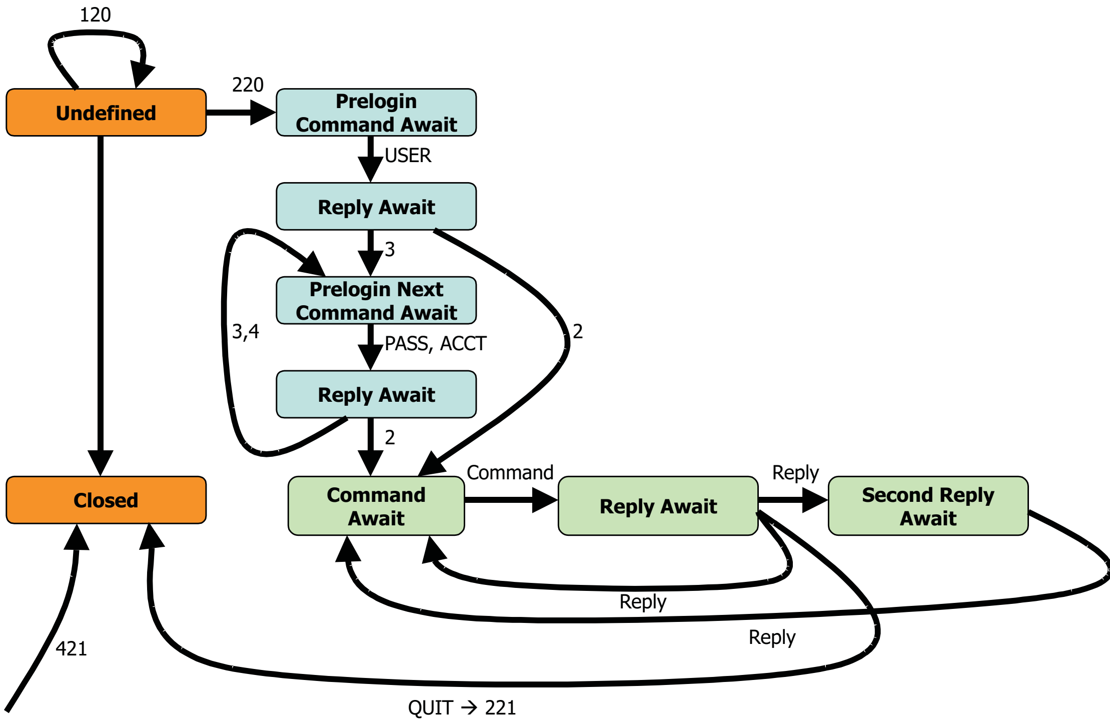


Figure 4.3: The derived state machine used by the FTP plugin

machine traverses one of the three in Figure 4.4 depicted states. These states however are just

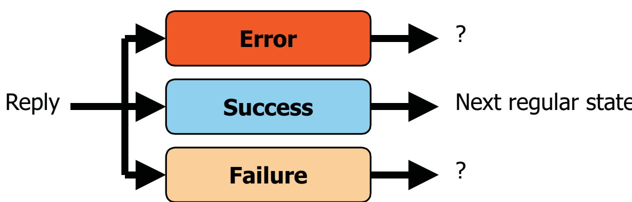


Figure 4.4: After each command / reply sequence one of the illustrated states is passed. They do not affect the FTP state machine itself but are meant rather for additional information.

for information and output purposes respectively and do not affect the internal current state. The three states correspond to the output states of the single FTP state machines. Using the above temporary states makes sense because we actually have to track the state of the server part of the communication. This is the network site which is known and it should be not possible that the server application is comprised. So the definitive state transition will be just done when the server response has been received. After reply reception we will emit information about the state of the server - either the request from the client caused an error, a failure or it was successful.
The overall state machine has been implemented rather straightforward. There is a class `FtpData` which holds all the possible requests and replies. For this purpose the following two

data structures `requestCommand` and `replyCode` are used. Listing 4.1 shows the declaration of those structures.

Listing 4.1: The structures representing requests and replies respectively

```
struct replyCode
{
    unsigned short code;          // The reply code
    unsigned short firstDigit;    // The first digit of the reply code, e.g. 5xx
    char* msg;                    // The message which should be printed if this reply
                                  // code is detected
    unsigned short argumentType;  // The type of argument to parse. Currently only
                                  // hostport is implemented for the replies

    replyCode(unsigned short, unsigned short, char*, unsigned short);
};

struct requestCommand
{
    char* cmd;                    // The command as a string
    char* msg;                    // The message which should be printed if this reply
                                  // code is detected
    char* nextCommand;            // Some requests need further commands
    unsigned short argumentType;  // The type of argument to parse.
    unsigned short cmdSize;       // The size of the command, e.g. 3 digits
                                  // (important for parsing the arguments)
    set<int>* validReplyCodes;    // Each command has a set of allowed replies and
                                  // these replies result in a new state
    unsigned short stateType;     // One of the state machines from the specification

    requestCommand(char*, char*, char*, unsigned short, unsigned short,
        set<int>*, unsigned short);
    ~requestCommand();
};
```

These data structures are hold by two containers from the *Standard Template Library (STL)* and are publicly accessible as members of the class `FtpData`. They are declared as following:

- **`typedef map<string, requestCommand*> requestMapT;`**

The request commands are hold in a map of this type. The requests are mapped to a unique string identifier which is the same as the command string in the structure `requestCommand`.

- **`typedef map<int, replyCode*> replyMapT;`**

The replies are hold in a map of this type. They are mapped to their unique reply code.

4.5.2 The plugin design

To track the state of a connection the plugin maintains a list of control connection items declared by the structure `ftpConnectionId`. Each connection is mapped to its corresponding state related structure `ftpControlConnectionInfo`. This structure holds information about the current state and a map of data connections which have been initiated by the control connection. Note that in the case of the PASV command which tells the server to create a listening socket, we just know if the control connection was successfuly negotiated when the server sends a reply with the number 227. For this reason the connection info structure also holds two data connection structures for the currently negotiated data connection and the data connection settings prior to the ongoing negotiation. This allows us to restore the previously data connection properties if a server refuses a passive data connection request. Data connections are described by the structure `ftpDataConnectionInfo`.
The functionality of the entry function which is called for each arriving packet contains the following main steps:

1. Look in the map holding active control connections if a connection for the given packet already exists. If it exists go to 3.

2. Create a new entry for the current control connection.

3. Test if the packet is a TCP closing packet. These packets are forwarded from the TCP plugin in order to have full knowledge of the connection state. If it is a closing packet, the connection state will be set to closed. Succeeding FTP operations on this connection will be ignored.
4. Decide from which site the packet originated. The packet parsing and state handling will be somewhat different for request and reply messages.
5. Parse the message.
6. Update state and do reporting.

At some positions in the code this rather conceptual functionality had to be broken and especially the order of first parsing the message and then update the state was not always the most efficient way. This is the result of the need for one single overall state machine which emerged from the five possible state machines of the specification.

4.6 Evaluation

After implementing the FTP protocol plugin for the tracker framework, evaluation had been postboned since the signature generator was intended to be demonstrated at the NoAH meeting in Amsterdam, NL. Because of this, evaluation of the plugin was planned to be done after the implementation of a first signature generator. Meanwhile we have been strucked that the implementation-oriented design of the tracker framework is not very convenient. The reasons are the following:

- Implementing a state tracking plugin for each desired protocol is cumbersome and requires too much effort. Protocols are seldomly fully specified and require a lot of experience and effort to be implemented correctly. Furthermore extensions of the original protocol specification often complicate the protocols and it has been sighted that even popular applications omit extended functionality of the original network protocol.
- Adding additional functionality that has always the same basic structure as hardcoded plugins is normally not a good design choice. Generic design and configurability are the catchwords for this point and are the techniques which constitute a valuable framework design.
- As network protocols are widely implemented and in use some already existing implementations should provide the functionality instead of implementing twenty years old network protocols by our own from scratch.

These observations amongst others are recapitulated in section 6.1. Despite the further direction of this work heading towards another approach than using the tracker framework a short evaluation of the performance of the implemented plugin will be presented.

4.6.1 Tracking states

After a couple of initial tests it showed that not a single packet was forwarded to the FTP plugin. During packet analysis the invalid TCP checksum for each network packet attracted attention. It came out that this effect is caused by a feature of newer network interface cards (NICs) called *TCP checksum offloading*. When this option is enabled, the network adapters will calculate the checksum by themselves, making the CPU and the operating system not have to do this work. When we are capturing and analyzing outgoing network packets we normally receive packets before they get to the network adapter and thus we won't see the correct checksum because it has not been calculated yet[1]. Offloading parts of network protocol processing from the host CPU is a hot topic in the research area of operating systems and network acceleration. As a consequence testing the TCP checksum in the TCP plugin had to be deactivated. Another option would have been to deactive offloading at the local host.
Listing 4.2 shows a single but complete packet element as it is logged in the XML log file of the Tracker. The listing 4.3 shows an excerpt of the Tracker log output for the FTP subelements of the packet elements only. It reveals that we are able to track commands and their replies and even are able to make note of negotiated data connections.

[1] Source: http://www.wireshark.org/docs/wsug_html_chunked/ChAdvChecksums.html

Listing 4.2: A single packet element

```
<packet packetId="1124" thread="0" captureTime="1189000430.798982">
        <IP src="192.168.3.10" dest="192.168.3.11" payloadSize="59"
                        direction="fromClient" />
        <TCP connectionId="1111" src-port="56417" dest-port="21" payloadLen="27"
                        newState="TCP_ESTABLISHED"/>
        <FTP valid="true">
                <request type="PORT" argParsed="true" argValid="true" argLength="20"
                        newState="FTP_CTRL_REPAWAIT" lastReply="215" />
        </FTP>
</packet>
```

Listing 4.3: Tracker log excerpt

```
<FTP valid="true">
        <request type="PORT" argParsed="true" argValid="true" argLength="20"
                    newState="FTP_CTRL_REPAWAIT" lastReply="215" />
</FTP>

<FTP valid="true">
        <reply type="200" argParsed="false" argValid="false" argLength="0"
                    newState="FTP_CTRL_SUCCESS" currentCmd="PORT" />
</FTP>

<FTP valid="true">
        <request type="LIST" argParsed="true" argValid="true" argLength="0"
                    newState="FTP_CTRL_REPAWAIT" lastReply="200" />
</FTP>

<FTP valid="true">
        <reply type="150" argParsed="false" argValid="false" argLength="0"
                    newState="FTP_CTRL_SECREPAWAIT" currentCmd="LIST" >
            <dataConn portSite="client" portNum="52092" />
        </reply>
</FTP>
```

4.6.2 Measuring performance

For measuring the delay added to the overall processing time of the Tracker an FTP batch job was used. Listing 4.4 shows the script.

Listing 4.4: FTP batch script

```
open 192.168.3.11 user Sysadmin peterli

dir
cd /pub
dir
cd /upload
dir
rename test.txt neu.txt
dir
rename neu.txt test.txt
dir

cd /pub
get empty_document.bin
get empty_document.txt
get short_document.bin
get short_document.txt
get long_document.bin
get long_document.txt

by
```

It opens a connection to a remote host and performs different file operations. The time needed to process a packet is measured by using the system call `gettimeofday` before and after the packet processing inside the code of the Tracker. The measurements can be enabled by using a precompiler directive to set the variable `PERFORMANCE_MEASUREMENT`. Figure 4.5 shows the delays for the processed packets exchanged with the FTP server according to the batch script.

It can be seen that the FTP plugin has only a small part of the overall delay. Furthermore the delay approximately remains constant. The overall delay was also measured without the use of the garbage collector plugin and has not shown any significant differences. The average delay induced by the FTP plugin for processing FTP network packets is about 26 microseconds. This delay corresponds to a one-digit percentage compared to the overall delay for processing a packet with the Tracker.

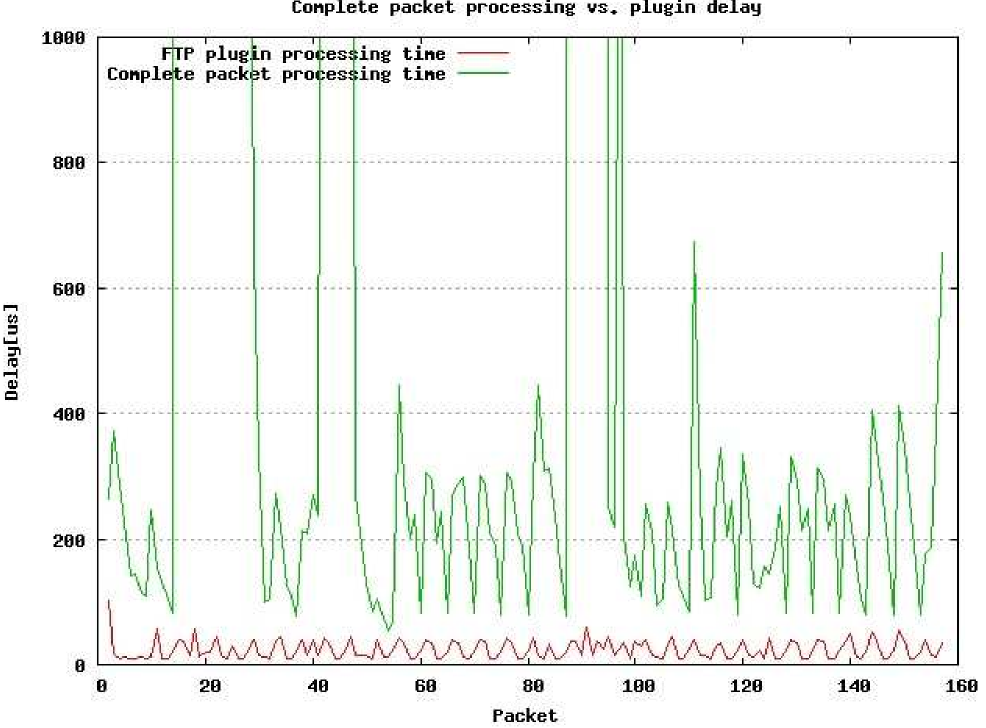


Figure 4.5: The delays for processing the complete packet and processing the FTP protocol part with the FTP plugin respectively

Chapter 5

Signature Generator

In the context of the NoAH project, the goal of this thesis was to present a simple *signature generation mechanism (SGM)* in a first step as a proof of concept (PoC). This simple SGM has been presented at the NoAH conference in Amsterdam, Netherlands, in June 2007. The goal was to demonstrate the successful collaboration of the NoAH project components, i.e. Argos, the tracker framework, the Signature Generator and an IDS supporting the signature format of the generated signatures.

5.1 Requirements

The signature generation component mainly can be divided into two parts. The first one is responsible for correlating information from Argos and the Tracker and preparing the extracted information for the second component, the SGM. The SGM has the function to synthesize a concrete signature for e.g. an existing IDS from the extracted information. It exists only one extraction component but an arbitrary number of SGM components.
Prerequisites for the information providing component are the following.

1. The information extractor has to work reliably. This means that during each process of extracting attack information it is known how much information is available if any, and of which quality the information is. The latter depends on the methods with those information was extracted as these have not always to be the same.

2. The information extraction should be fail-proof, i.e. it is not possible that an information extraction process can terminate unexpectedly.

3. The information extraction should be fast. At this point the motto ”safety before speed” applies.

Prerequisites for the SGM components and for a simple SGM for demonstration purposes in particular are

1. The target signature format should be compliant with at least one IDS available for free.

2. This IDS should be popular, widespread and easily configurable.

3. The signature rule language should be rather simple in order to create signatures efficiently. This assures that focus is laid on the information extraction.

4. In contrast to the above point the signature format should be powerful enough. This means that the accuracy of the extracted information should not be significantly reduced when transformed into a signature of this format.

5. The signature format should allow signatures with a low false-positives rate. More explicitly the false-positives rate should be constrained by the quality of the extracted information and not by the signature generated thereof.

5.2 Signature format evaluation

For a simple SGM the rule language to build signatures should be rather simple. Two of the most prominent open source IDS, Snort [11][1] and Bro [33][34], have been considered and were compared against each other. Bro is to prefer over Snort if we want to build a customized and very secure system. But because of the simplicity of the Snort rules and its architecture Snort has been chosen to serve as the basis for a first SGM. Furthermore the Bro system is compatible with Snort rules. It has shown that the Snort rule language is quite powerful despite its simplicity. In accordance to the stated prerequisite the Snort rules should not narrow the power of the extracted information and thus should not increase the theoretical false-positives rate of the extracted information.

5.3 Architecture

This section presents the architecture for the information extracting component. The extracted information could be seen as a generic kind of signature although this is not consequently realized in the design[1].

5.3.1 Extracting alert information

Recall from the Figure 2.2 that the interface connects the log file information available from Argos and from the Tracker. From this information we should be able to extract useful attack information and generate a first simple signature.
Because the code of the interface that was already available was not that big, we decided to integrate the SGM into the interface. There are only a few classes involved in the signature generation. The class `Extractor` extracts information from the given Argos log file and tries to correlate this information with the Tracker log files. The Argos alert reports are named according to the following pattern:

```
argos.csi.<random alert identifier>
```

where the number `random alert identifier (rid)` is created by Argos and should be unique among the entire set of created alert reports. The available set of extracted information for describing an attack in the extractor class depends strongly on the state of the implementation and research. Therefore the class can be seen as a container holding all the extracted information as members. For each information there will be a corresponding function which returns the information value.
Each time an alert is generated by Argos the interface notices this by receiving a message at the control socket connected to Argos. Subsequently the information extraction and signature generation will be started.
`Snitch.pl` is an additional process running on the guest system under attack and is used to determine the process which has been attacked and some further information. The snitch process sends its information to the interface via a separate control socket. This is why a second thread was needed which waits for incoming messages from the snitch process. Because information from the thread connected to snitch is needed by the main process for signature generation, a special map was needed which can be accessed safely by multiple processes. The class called `SnitchMap` uses a semaphore to protect its information from concurrent accesses. This method is very similar to the one used by the class `SynchronizedMap` in the tracker framework. The interface polls the `SnitchMap` when it needs information for a certain alert report. Listing 5.1 shows how the interface gathers available information when an alert message from Argos was received.

Listing 5.1: Extracting the entire set of available information for signature generation

```
// int rid holds the current random alert identifier

// Extract information
if (extractor.extractInformation("ArgosAlertReport"))
{
    // Access the synchronized map
```

[1] As we will see a refactored version of the information extractor will treat the extracted information as a generic signature.

```
    SnitchEntry* currentEntry;
    int count = 0;
    while ( !snitchMap.unbagElement(rid, &currentEntry) )
    {
        sleep(5); // sleep for 5 seconds and try again
        count++;
        if (count > 5) // we won't wait any time longer
            break;
    }
    if (count > 5)
        // Information for the current rid was not available
    else
        // We gathered information successfully
}
```

First information is extracted by the extractor class. If the extractor was successful, we look up at the snitch map for the appropriate snitch entry. We poll the snitch map repeatedly at a certain polling interval until we either find the desired entry or we exceed the maximum number of lookups. Because information from Argos is almost concurrently sent with the message from the snitch process, polling is reasonable. Information from the `SnitchEntry` object and information provided by the `Extractor` now can be combined for generating a signature.
The class `SignatureFactory` is responsible for factoring a rule out of the extracted information. A class representing a concrete signature has to be derived from the class `Signature`. A function factoring a signature of this type has then to be added to `SignatureFactory`. Figure 5.1 depicts the process structure of the NoAH core after introducing the snitch thread.

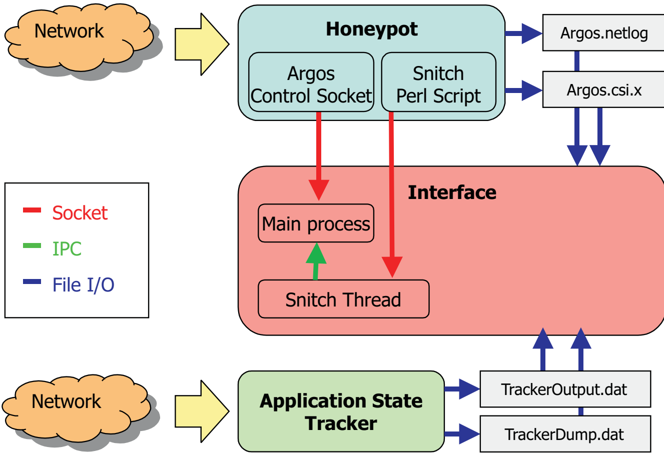


Figure 5.1: The process structure after adding alert information extraction

5.3.2 Archiving signature information

The signature generator should be able to archive common information of an alert as well as concrete signatures. This has been achieved by adding a simple *MySQL* database. In this database there are currently three important tables. One is for indexing the signatures, another holds all the common information. Concrete signatures should have their own tables. The three available tables are described below.

signatures This table holds the meta information for a signature database entry. Tuples of this table identify a signature in the database uniquely. The only master key in this scheme is provided here.

details It provides detailed information on the attack like source port or the position where a possible buffer overflow occurred. This table is used to create specific signatures, e.g. a Snort signature. If a signature will be refined information has to be changed first here.

snortSignatures Shows the concept of how a concrete type of signatures could be stored in a table. Each signature class that is available should have its own table to store the rules.

As the database itself could be on a remote machine or kept locally at the same place where the interface is running, different scenarios how to access the database and to provide API functions are thinkable:

Local signature collection This scenario is denoted local because the signature collection, refinement and deployment is done locally, in the same process. This approach was thought of as a first and simple approach, especially for demonstrating the abilities. Note that despite of the property "local" the signature database itself could be on another site than local.

Remote signature collection It is meant that we implement another signature collector class which would connect with a remote process for signature collection, refinement and deployment. In this case the signature database would be at the remote site too.

For demonstration purposes a local signature collector class `LocalSignatureCollector` has been written. The following functions provide the API for database access:

- **`bool insertSignature(SnitchEntry*, Signature*, Extractor&)`**

Inserts a signature into the database. If a signature for the corresponding attack already exists first a refinement is attempted. If a refinement is not possible a new signature for the same attack is created. If the refinement would not make sense the signature insertion is cancelled.

- **`bool updateSignature(Signature*, unsigned int)`**

This function can be used to update a signature of a concrete type.

As the collector classes represent the access point to the signature database they should provide additional functions, e.g. for printing tables and converting numerical constants into their string representations. This is provided in `LocalSignatureCollector` by the functions `print...()` and `resolve...()`. Getting back to the process structure of the entire architecture we have added a database component as depicted in Figure 5.2.

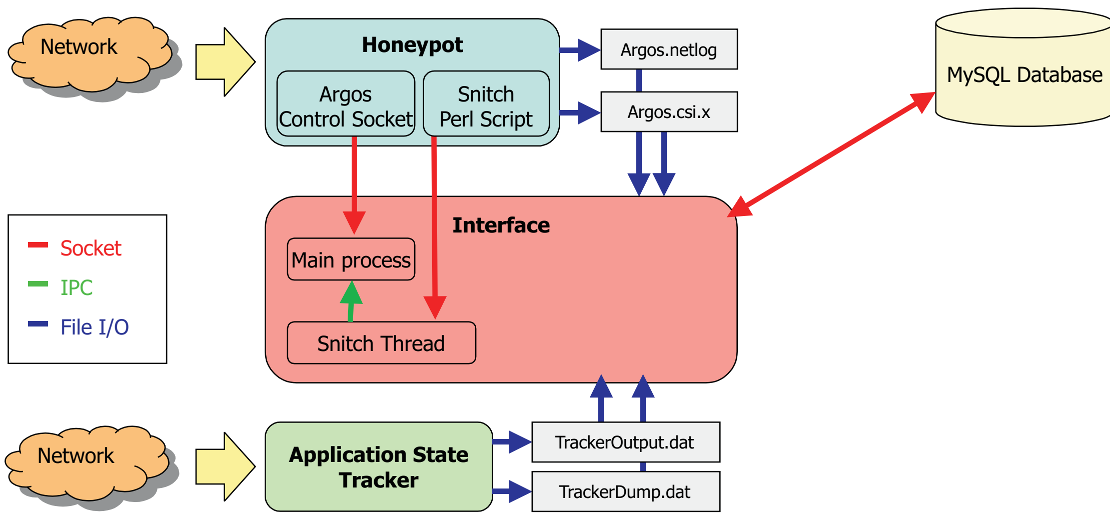


Figure 5.2: The process structure containing a *MySQL* database

5.3.3 Demonstrating signature capabilities

For demonstrating the usability of the generated signatures a small *Java* server program has been written that waits for a signature description. It then updates the Snort rules configuration file and restarts the IDS. This can be easily done in Java by using the classes `Runtime` and `Process`. The functionality to connect the signature generator with this demonstration host has been integrated into the local signature collector as this is the place where signatures are stored and refined.

5.3.4 Attack replaying

During the implementation it has shown that it could be useful to have a facility for replaying detected attacks. A possible benefit is to refine the maximum length of a protocol field triggering a buffer overflow. Furthermore it would be even possible to test the created signatures against the detected attack by using the attack replaying in conjunction with the demonstration host. This approach describes a similar methodology as proposed by Microsoft research introducing *self-certifying alerts (SCAs)*. This approach has been presented in the related work chapter in section 3.4.4.
Attack replaying comprises the following steps:

1. The available replay target systems for a certain attack are identified. Currently this is done by reading the available services and the platform properties from a XML configuration file.
2. Then it is tested if the target is responding but no service of the requested type is running.
3. The remaining targets can now be used for replaying. The server application to be attacked is started by using a *secure shell (SSH)* connection.
4. If the service does not respond after the attack we can assume that the attack was successful. When we close the SSH connection the child processes and thus the started service will be terminated properly.

Each replay target system is represented by an instance of the class `ReplayTarget`. The class `AttackReplayManager` starts the attack replaying mechanism, either as a set of synchronized processes or in the same process. This allows us to even use multiple machines to attack replay targets. This resulting ”one-to-many-to-many” relationship between the signature generation mechanism and the replay targets avoids the vulnerability to *Denial-of-Service (DoS)* attacks by minimizing the possible bottle necks. The relationship is illustrated in Figure 5.3. The replaying process itself is done by objects of the type `AttackReplayer`. This class

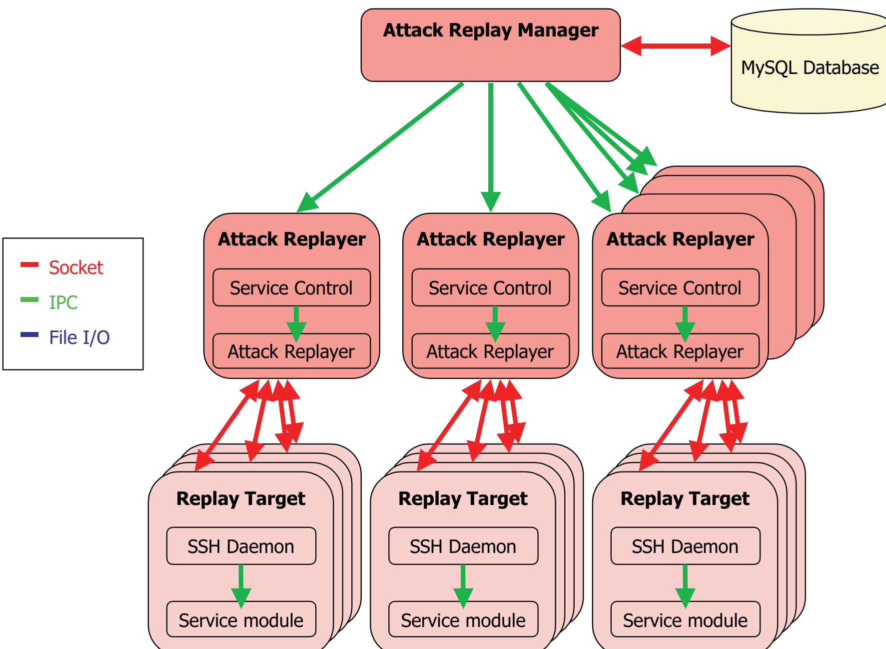


Figure 5.3: The ”one-to-many-to-many relationship” for the attack replaying mechanism

manages the connection for the attack replaying and starts a concurrent control process for the SSH connection. If we want to have the ability to replay an attack for a certain application protocol, we have to derive a class for this protocol from `AttackReplayer`. For demonstrating the concept an FTP attack replayer has been implemented. Figure 5.4 shows the overall and final architecture with all its interacting components. It has to be admitted at this position that ”self-certifying” the generated signatures was achieved only in a restricted sense. The achievement

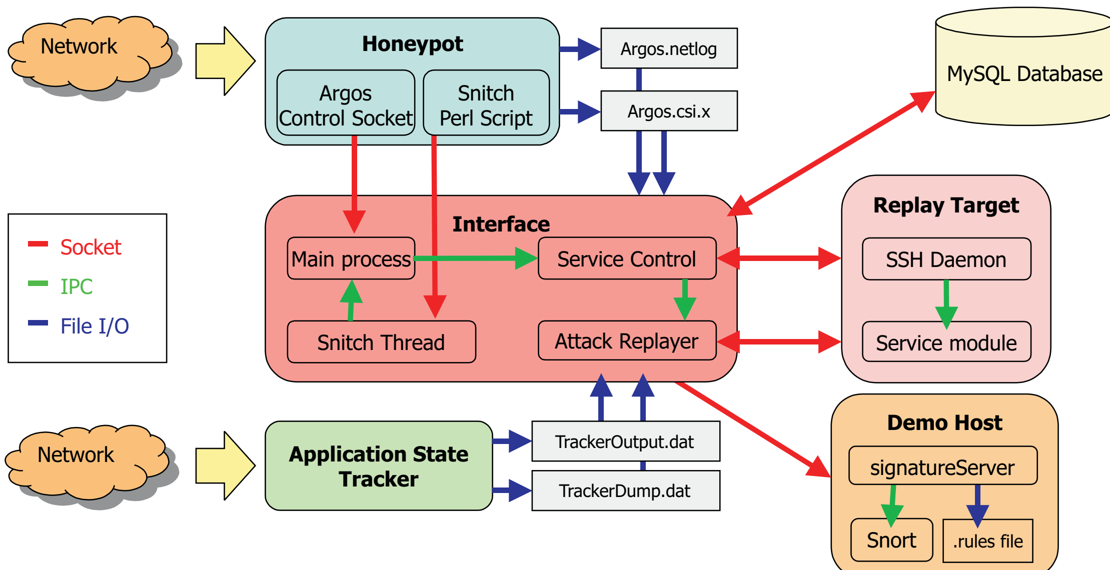


Figure 5.4: The complete process structure of the signature generation mechanism

was the functionality to replay FTP conversations up to the packet triggering the buffer overflow. The buffer overflow has been verified by generating packet fields that are longer than the buffer length of the vulnerable application. This information can be obtained from the generated signatures.
The actual information provided by the signature is not the bound of the buffer length which leads to a buffer overflow, but the critical buffer length which overwrites a target address for execution. Protecting the application from overwritten jump targets or return addresses instead from the buffer overflow itself is not a big difference since control-flow hijacking can be prevented either way. Argos aims mainly at the detection of remote code execution attacks. Control-flow hijacking is the method enabling exactly such attacks.
So at least it is possible to verify that a signature describes a vulnerability and that this vulnerability can be triggered by the packet field and field length specified.

5.4 A signature generation mechanism (SGM)

A simple approach to generate signatures from the Argos log files was proposed in [3]. The authors have applied the LCS method to extract a content string which will serve as an identifier for the detected attack. This string could be easily used in e.g. a Snort signature. More precisely Argos can detect if an overwritten return address is about to be used in the control flow. This overwritten return address is identified in the corresponding network packet. The position found in the network packet is then used to apply an LCS search around the return address in both the process memory and the network packet. The LCS approach was reimplemented with adequate effort by using the Cargos library. This library provides an API for accessing the log files generated by Argos. A Snort signature generated by this SGM could look as following:

```
alert tcp any any -> any 21 (content:"|05 D3 FF 00 09 A4 C6 12|";)
```

Because the extracted byte string can contain information that could be altered but the vulnerability could still be triggered and the exploit executed, a signature created by this method will not be able to identify altered attacks of the same exploit. Even if only the value of the return address itself is taken as the byte string, the address may be slightly changed and the rule will not trigger anymore. On the other hand the less bytes are contained in this signature the higher the possible false-positives rate will be.
This is where the log files from the Tracker come into play. In the state tracking XML log file we have a string contained which identifies the message type, e.g. a command string for FTP command messages. With this information it is possible to have less information in the content string. A possible Snort signature could now look as following:

```
alert tcp any any -> any 21 (content:"USER"; content:"|09 A4 C6
12|";)
```

However the problem remains the same. As we have a small byte string the false-positives rate would be high. So what do we actually intend to achieve our generated signatures? We try to prevent an application-specific exploit from taking place. In other words we try to protect the vulnerable service from exploits triggering its vulnerability. As the cause for a buffer overflow often is a protocol field of variable length a possible countermeasure is to filter network packets with a certain protocol field being longer than the biggest legitimate buffer size. In the case of the FTP protocol this is simple since the field of variable length in command messages is the last field of the message. Thus the Snort signature looks as following:

```
alert tcp any any -> any 21 (content:"USER"; dsize:>32;)
```

This rule assures that the user name extracted from the message will be not longer than a certain maximum length. A buffer overflow caused by too long user names can not be triggered anymore. As already mentioned in the previous section the maximum length for a protocol field as specified by our signatures is not the bound at which the buffer overflow would be triggered but at which the control flow of the application would be altered instead, e.g. by overwriting a function's return address. This is because Argos determines when and at which position tainted data will be executed. But this information may not be sufficient to estimate the real bound for triggering the buffer overflow. However protecting a vulnerable application from altering its control flow is regarded as a very strong protection unless as even powerful as protecting from the buffer overflow itself.
Unfortunately for variable-length fields that do not occur at the end of a packet the method presented above obviously does not work. To come by this shortcoming a more elaborated architecture for the signature generation is presented in chapter 7.
The above presented SGMs have been imlemented but only the first approach is not dependent on the application protocol. For the other two approaches an application protocol plugin for the tracker framework is necessary. The implementation of an application protocol plugin for the FTP protocol has been presented in chapter 4. Note that the signatures shown are simplified. A complete signature generated would look like the following:

```
alert tcp any any -> any 21 (msg: "(NoAH) Overwritten return address
via FTP USER command in warftpd.exe (win2k)" dsize:>15; flow:
established,from_client; content: "USER"; nocase; content: "|E2 31
02 75|"; sid:1000005; rev:4;)
```

5.5 Evaluation

As explained in the section 4.2 an attack usually exploits one or more security-related vulnerabilities caused by software bugs which normally have their roots in the implementation or design. This is why exploits are only exemplarily available. It follows that measuring the false-positives rate for the generated signatures does not have to be compellingly reliable.
One often made mistake regarding the generation of attack signatures is that the signatures do not take into account the vulnerable application itself that initially enabled the exploit. But this is one of the most valuable information since applications merely exhibit exactly the same vulnerability. Information about the targeted application is not supported by the Snort rules language which poses a major drawback. However, it is thinkable to select the active set of Snort signatures for the Snort IDS according to the running services by an external program. This way we could guarantee that each active signature on the host protects a running application with an existing vulnerability. The signature database stores the name of the vulnerable application and the platform for each signature.
A signature is the better the more it describes the vulnerability of the application and not just properties of the attack exploiting the vulnerability. With the presented approach of using Snort rules to protect variable-length fields the signatures protect the vulnerability pretty accurately.

The extracted attack information allows the creation of signatures with a zero false-positives rate when

1. Argos detects the execution of tainted data, e.g. data injected via the network that is about to be executed. This corresponds to a remote code execution attack. In this case the interface will receive the address value of the target instruction. Normally the target address of e.g. a return or jump instruction has been overwritten via a buffer overflow.

2. We are able to read the instruction value from the memory dump of Argos at the address given from above.

3. It is possible to identify the network packet from which the target instruction originated.

4. The buffer overflow used to overwrite the instruction's target address was triggered by a too long protocol field of variable length.

5. The Tracker log file provides information for a protocol message type identifier.

If some of the above conditions do not hold we could generate some kind of emergency signature. An example for this sidestep strategy are byte matching algorithms between the memory and the network data creating some kind of byte string identifier. The LCS algorithm is one example of such an algorithm. The different possible signatures have been described in the preceding section. With a byte string identifying the exploit the false-positives rate would be poor. This is because benign network traffic could contain the string too. We can weaken the effect of such signatures by assigning poor quality to the signature, by refining it or even by discarding the signature.
Note that the architecture and the SGM presented in this chapter have further restrictions. They have been overcome with the refactored version of the signature generator presented in chapter 7. The two most important restrictions are

1. The vulnerable field of variable length has to be the last field in the network packet.

2. The Tracker must have plugins for the appropriate protocols and the log file must provide information on the appropriate network packet.

Performance aspects are not considered in this chapter but are presented for the refactored signature generator in chapter 7. However, the presented delays for the refactored signature generator are in the same range as the ones for the original signature generator.

Chapter 6

A more generic approach for signature generation and state tracking

6.1 Observations

Protocol knowledge and decoding

As we have already seen at the end of the previous chapter the so far implemented signature generation mechanism has the disadvantage that the signatures are either not very precise or the mechanisms require implemented plugins which fully support the protocol decoding, denoted by *protocol knowledge*. This protocol knowledge is used by the SGM as well as by the Tracker and the attack replaying mechanism. Implementing plugins for each desired protocol is cumbersome and error-prone. Furthermore adding additional functioniality, here the protocol plugins, in form of hardcoded functions is not state-of-the-art. Searching protocol fields for their bounds which are not at the end of the messages unlike in the previously presented case of the FTP protocol will increase complexity of the parsing algorithms further. Recapitulatory it can be stated that our components such as the Tracker or the SGM has the need for full protocol knowledge. This can only be achieved with an adequate effort if the components can revert to an existing library providing at least functionality for protocol field parsing.

A generic state machine

For the Tracker and the attack replaying mechanism we need not only protocol knowledge but a state machine for each desired protocol. Note that for these two components the state machines will differ in some points. So as we can see we have the demand for a generic protocol state machine in addition to the protocol decoding. The state machines should be configurable for each protocol according to some specifications. The section 6.5 presents different approaches how a state machine can be implemented and how configurability can be obtained.

Knowledge of protocol field meaning

In the case of the replaying mechanism even knowledge about the meaning of protocol fields is needed. This implies that protocol decoding itself is not sufficient. For field decoding we have to know for instance the length of the field and its data type. But for replaying a network conversation we additionally have to know what the fields mean or in other words what purpose a field is for. In the section 6.6 some ideas for achieving synthesis of network packets on application-level are discussed briefly.

Conclusions

Overall, the consequences of the observations presented in this section are the necessity of a redesign of the entire tracker framework, parts of the signature generator and the attack replaying mechanism. The mentioned three techniques needed to accomplish the nontrivial task of signature generation for almost arbitrary protocols will be addressed in the next sections, especially with respect to the implementation aspect.

6.2 Survey

In the previous section we concluded that we need

1. the ability to decode protocol fields denoted by protocol knowledge,
2. a generic and configurable protocol state machine,
3. knowledge of the meaning of protocol fields and how to synthesize them.

To accomplish the above three requirements in a first step available tools for complete protocol modeling have been evaluated. Although a lot of research work exists on this topic libraries are seldomly publicly available. There is some interesting research work on the field of protocol analysis but no work seems to fit our needs entirely. On the other hand some presented approaches appeared to be an overkill with respect to our needs. It has to be emphasized that often, especially in the case of research work, even if software is available for an approach the configuration and specification files for concrete network protocols were only written exemplarily. The following approaches are introduced briefly.

Specification and Description Language (SDL) SDL [32] is used for the description of reactive and distributed systems. It is a standard defined by the ITU-T. Opposed to the approach of the IETF to specify standards in arbitrary text form the ITU-T intends to use a declarative language to exactly specify and determine behaviour of new protocols. Especially in the field of wireless communication some example protocol specifications are available. However none of the IETF protocols seem to be available in SDL. A SDL compiler is avilable from Humboldt university.

Generic application-level protocol analyzer (GAPA) This approach is presented in [24] by Microsoft Research. It defines a protocol modeling language named GAPAL and uses a stream-based technique to process network packets.

It turned out quickly that no suitable software library would be available which fits all of our needs. So we decided to look for approaches addressing only one of the above requirements at a time. The section 6.3 addresses the first of the three above mentioned requirements. The section 6.5 is about a framework for a generic and configurable state machine. The concluding section for this chapter, section 6.6 deals with the automated packet synthesis for packet replaying.

6.3 Protocol knowledge

A survey of available approaches for protocol decoding is presented in this section. The often used abbreviation *PDL* stands for *protocol declaration language* and denotes a language which can be used to specify certain characteristics of a protocol such as field masks of specific packets.

From the presented approaches in table 6.1 NetPDL had been investigated further. This declaration language is part of an extensive library called NetBee which is currently written at the university of Turin, Italy. The presented framework supporting NetPDL was very promising and will suffice our needs. The next section gives an overview of the NetBee framework.

Name	Characteristics
libPDL[a]	• Works on byte streams • No network protocols yet (used originally for MIDI protocols)
NetPDL[b]	• Potential for a new standard • Comes with PDML and PSML • Not yet completely ported to Linux • Performance comparable to Tethereal • Over 64 protocols available
Ethereal / Wireshark[c]	• Protocol dissection and decoding engine Epan: ○ Supports over 700 protocols ○ Protocols are hard-coded in ANSI C, more than 800 kLOC • Seems not be intended for usage as external library
Tethereal / Tshark[c]	• Uses Epan of Ethereal • Supports different outputs such as PDML, PSML or human-readable • Output seems to be only accessible from a file
tcpDump[d]	• Usage not clear, seems not to be intended for library usage • Not many protocols • Protocols are hard-coded
GASP[e]	• GASP = Generator and Analyzer System for Protocols • Compiler generates protocol-specific code • Framework is build on the script language Tcl/Tk • Most recent software release allows to encode and decode packets of different protocols • Development seems to have been stopped
JNetStream Protocol Decoder[f]	• Java library for easy manipulation of captured network packets • Based on IO streams • NPL (Network Protocol Language) is interpreted language for protocol decoding • Listed for completeness
Black-box tools	• Either commercial → no source code available • Freeware → No API available, no documentation, only a GUI instead

[a] http://nmedit.sourceforge.net/subprojects/libpdl.html
[b] http://www.nbee.org/
[c] http://www.wireshark.org/
[d] http://www.tcpdump.org/
[e] http://laurent.riesterer.free.fr/gasp
[f] http://sourceforge.net/projects/jnetstream/

Table 6.1: Overview of projects and standards for protcol analysis and decoding

6.4 The NetBee framework

The framework and the corresponding library *NetBee* are presented in [22]. The purpose of this library is to provide a modular and flexible API for protocol analysis, traffic classification and packet sniffing and filtering. The library consists of different modules which implement different standards, some of them developed explicitly for this framework. Amongst others the NetPDL language is supported by NetBee. The library consists mainly of the following components:

- A Packet Decoder
- PSML and PDML Readers for decoded protocols
- A NetPDL Protocol Database
- An Interface for the NetVM
- A Packet Processor

To achieve protocol knowledge the packet decoder, its related PDML reader and the NetPDL protocol database are particularly of interest. The packet decoder processes packets based on protocol descriptions in the NetPDL language. NetPDL specifications are currently available for

about 64 or more protocols and are stored in the protocol database. The two types of readers provide categorized information of decoded packets. The PDML language will be presented in a following subsection and is very useful for packet decoding. Of minor importance is the *PSML* specification which describes just a summary of packet information and uses Visualization Extensions of the NetPDL language.
In the following first subsection the NetVM is described briefly. The packet processor component listed above uses the NetVM.

6.4.1 Network Virtual Machine (NetVM)

Network Virtual Machine (NetVM) is a virtual machine targeted to network packet processing. It is presented in [19]. The NetVM architecture is modelled after the one of an ideal network processor, i.e. a processor that includes a set of hardware modules devoted to speed-up packet processing, e.g. lookup tables, and a set of assembly instructions that are targeted to common packet processing tasks, e.g. checksum computation. The NetVM defines

1. the architecture for a new network processor
2. how to interact with these components from an application

The NetVM aims at becoming a reference specification for network-related data processing. This interesting hardware-independent approach for network packet processing is of rather poor importance for this work and it will not be referred to in the remaining text.

6.4.2 Network Protocols Description Language (NetPDL)

The *NetPDL*[1] language is an XML-based language that aims at creating a unique database of protocols that can be used by arbitrary applications. It is presented in [23]. The language intends to describe the basic features of each protocol, e.g. the protocol fields and the protocol encapsulation. Additionally, a set of optional specifications can be defined for some specific tasks. The NetPDL Visualization Extension aims at defining how to print a decoded protocol, e.g a 32 bit number representing an IP address should be visualized in the dotted-decimal form, and more. Each NetPDL engine must be able to parse the NetPDL specification, while it can be able to understand only the optional parts that are of interest of that application.
The NetPDL language is easily extendible according to the needs of some particular application, its syntax is easy to understand, and the parsing is very simple thanks to the existing XML technologies. A first implementation of a NetPDL engine (and a NetPDL database) can be found as already mentioned in the NetBee library. Listing 6.1 shows an example definition in NetPDL for the IP protocol.

Listing 6.1: Example of a NetPDL definition for the IP protocol

```
<protocol name="IP" longname="IPv4␣(Internet␣Protocol␣version␣4)">
    <format>
      <fields>
        <field type="fixed" name="verhlen" longname="Version␣and␣header␣Length">
          <field type="bit" name="ver" longname="Version" mask="F0"/>
          <field type="bit" name="hlen" longname="Header␣length" mask="0F"/>
        </field>
      ...
```

6.4.3 Packet Details Markup Language (PDML)

The *PDML* language[2] is a simple language that keeps the information related to a decoded packet. PDML stands for *packet details markup language* and it is strongly related to NetPDL. This language is used by a NetPDL engine that understands the NetPDL Visualization Extension to create a detailed view of each packet.
A detailed view of a packet is an XML file that contains all the important information related to protocols and fields that are contained in one packet, e.g. the protocols, all the field names and their values, and more. The PDML specification is a way to organize this information. However

[1] http://www.nbee.org/Docs/NetPDL/
[2] http://www.nbee.org/Docs/NetPDL/PDML.htm

the XML representation has not to be saved as a file stringently and could just be kept in memory for further processing. A PDML document lists all the packets contained in a capture file, detailing the most important information for every protocol that has been found in the packet and for each field that has been decoded within the protocol.
The PDML language is becoming a de facto standard in packet analysis and is already supported by Wireshark and Tshark as an output method. Listing 6.2 depicts an excerpt of a PDML description of a decoded packet for the IP protocol. It corresponds to the NetPDL definition shown in listing 6.1.

Listing 6.2: Example of a PDML description for the IP protocol

```
<proto name="IP" pos="15"
    longname="IPv4␣(Internet␣Protocol␣version␣4)" size="20">
      <field name="verhlen" pos="15" showvalue="45"
            longname="Version␣and␣header␣Length" size="1" value="45">
        <field mask="F0" name="ver" pos="15" showvalue="4"
            longname="Version" size="0" value="45" />
        <field mask="0F" name="hlen" pos="15" showvalue="5"
            longname="Header␣length" size="0" value="45" />
      </field>
      ...
```

6.4.4 Usage of the NetBee library

The NetBee library and its API seemed to be exactly what we have been looking for. In order to be able to use the NetBee library and its source code for a few experiments we signed a non-disclosure agreement. Furthermore parts of the library had to be changed in order to run under Linux since it was primarily developed for Windows. Fortunately the developers wrote the code with portability in mind and most changes could be done with minor effort.
At some points the API of the NetBee library is not very comfortable. Two façade classes have been written to access the library more easily. They are described briefly at this place and are examined more thoroughly in the following chapter 7.

PacketDecoder This class can be used as an access point to the packet decoder mentioned in 6.4. It decodes a given packet and if the decoding was successful different field and protocol information can be obtained from the object.

DbFieldDecoder Because the packet decoder of NetBee is not able to return any information contained in the NetPDL definition of a protocol we have to access the database interface directly. The NetBee database API provides only basic functionality to access the NetPDL database and it had to be extended by this class, e.g. looking up a given field for a certain protocol and return the desired field properties.
This is important when we not only want to know what the value of a field is or at which position (this is what the packet decoder would do) but how the field was extracted and defined in the database. An example is when we have to know how the length of a protocol field is determined. Maybe the length is determined by another field. Exactly this kind of information is not available from the packet decoder.

6.5 Protocol state automaton

Different approaches for the implementation of a finite state automaton are presented in [27] and compared against each other. The implementation approach of J. van Gurp and J. Bosch, [15], seemed to provide the needed flexibility for implementing state machines for different tasks such as attack replaying or protocol state tracking. Due to the design of the automaton it is possible to specify automata by a simple declaration language. Furthermore additional functionality of the framework can be easily added. The basic design of the automata presented by Gurp and Bosch was adopted and slightly modificated.

6.5.1 Architecture

The Figure 6.1 shows the basic UML class diagram of the automaton. Each component of a

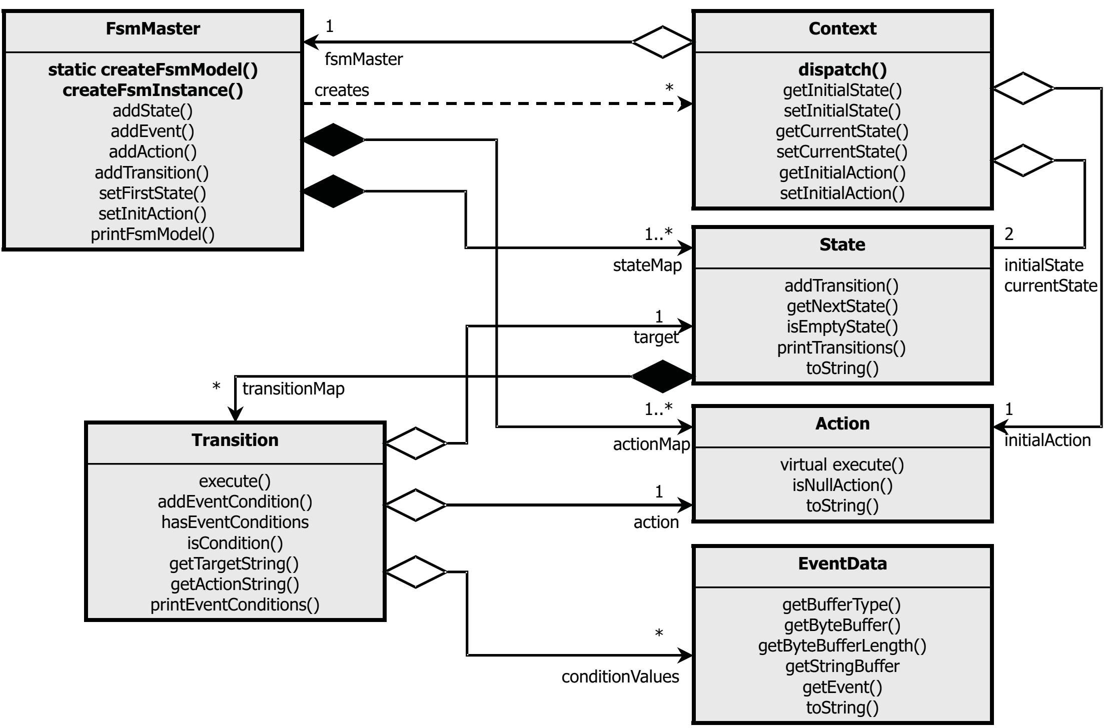


Figure 6.1: The basic class design of the finite state machine

state machine is represented by a class. The class `FsmMaster` is responsible for creating a specific type of automaton from a specification file. It will create all the necessary states and actions. They are then linked with each other by `Transition` objects. Each `Transition` is assigned to a source state and has a destination state. This linking scheme of the state machine components allows us to have multiple instances of the same automaton. The `FsmMaster` creates a `Context` object if requested which is the actual instance of the specified automaton. Each `Context` has its own current state. An event is characterized by an instance of `EventData`. If an event was received in form of such an object the appropriate `Transition` object for the current `State` object will be searched. If there is no corresponding transition the context will remain in the current state. It is the task of the specification file to specify more reasonable default transitions than just remaining in the current state. If a `Transition` object is found it will be executed and its corresponding actions will be invoked. Each executed transition will end up in a destination state which will be assigned as the new current state of the dispatching context. The following properties are implicated by this design:

- An arbitrary number of context objects can be instantiated without affecting the state machine functionality itself.
- Additional context objects will not increase memory significantly.
- The event dispatching mechanism is not thread-safe.
- The state machine is decoupled from its specific actions to be taken for the transitions. This allows to implement different functionality of the automaton by just deriving a new `Action` class.
- Additional conditions for selecting the appropriate state transition can be added by integrating the information into the `EventData` class.

Especially the last property pointed out provides some room for improvements. The `State` objects determine the transition candidate to be executed by comparing the event name with the assigned transitions and their corresponding event names. If the event matches a transition the additional properties of the event have to be examined. If some additional data type will be added to the event data condition supplementary, the functionality to find the appropriate transition

will have to be hardcoded into the class `State`. To solve this problem different solutions are thinkable. For instance we could add an interface named `EventDataProperty` which provides the functionality to compare against other objects of its own type. Then a `State` object has just to go through the property list for each transition and if a comparison fails continue with the next transition candidate.

6.5.2 Specifying protocol automata

The language that can be used to define automata is very easy and rather straightforward to use. Listing 6.3 shows a part of the automaton specification file for the FTP protocol.

Listing 6.3: Example of a state machine specification for the FTP protocol

```
<netProtoFsm name="ftp" firstState="ctrlEstablished">
    <types>
        <type name="logging" />
        <type name="replaying" />
    </types>
    <states>
        <state name="ctrlEstablished" />
        <state name="preLoginCmdAwait" />
        <state name="preLoginRepAwait" />
        <state name="cmdAwait" />
        <state name="repAwait" />
        <state name="ctrlClosed" />
    </states>
    <events>
        <event fieldName="Command" />
        <event fieldName="Code" />
    </events>
    <transitions>
        <!-- After tcp connection establishment -->
        <transition source="ctrlEstablished" target="preLoginCmdAwait" event="Code">
            <value>220</value>
        </transition>
    ...
```

The root element is called `netProtoFsm` according to the name of the state machine library which is called `libnetprotofsm`. It contains the different types of actions that are available, a list of states and events and the transitions which connect the different states. It can be seen that the event itself consists only of a string. This string normally corresponds to the name of the protocol-specific message type. Further specifications for event conditions are currently possible by declaring value elements for a transition. As already mentioned in the last section the additional event conditions are hardcoded. That means that if transitions should be able to be sensitive to more than an event name and a substring, it has not only to be added to the `EventData` and `State` class but even to the state machine declaring language itself. At this point it has to be remarked that no experience could be collected except during some test routines. The prospective usage of the library will show the best way of how to declare events. The author has the impression that two strings should be sufficient because normally an event is specified by a string denoting the message type and an additional value, e.g. of a protocol field. Nevertheless the concept of defining and passing events in the framework has to be refined.

6.5.3 Proposal for a timeout extension

For e.g. tracking the application state of a network connection or replaying network packets we need the ability to detect timeouts when a packet arrives. The state machine framework can be extended by adding a time stamp to the class `EventData`. Each state will then have either a specified or a default timeout value. If an event arrives the current state will first examine the defined timeout value against the current time minus the last internal valid timestamp. This however makes necessary to add a timestamp to the `Context` class.

6.5.4 Proposal for a framework for protocol state tracking and packet replaying

If the previous proposed extension for supporting timeouts is implemented the framework can be used for tracking the application state of network connections or replaying network packets from a log file. What has to be done is to create a derived `Action` object for both applications. In a preprocessing step the `EventData` structure has to be filled and then the event can be dispatched. Figure 6.2 shows the basic concept. The green components on the left hand side

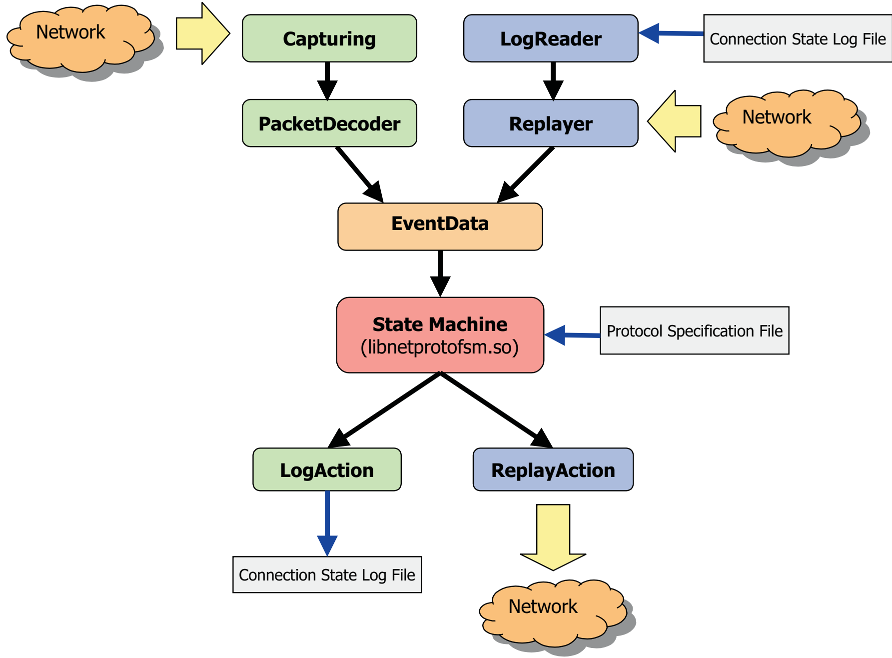


Figure 6.2: The state machine embedded in the two applications state tracking and packet replaying

correspond to the state tracking part and the blue components on the right side to the replaying part. The state tracker has to capture packets, e.g. via the library pcap. Then the captured packet has to be decoded, e.g. with the before mentioned façade classes for the NetBee library. The extracted information can be transformed to an `EventData` object which then will be used for event dispatching by the state machine. A `LogAction` class derived from `Action` will write useful information to a log file.
The replaying part would work very similar but is quite a little more complex. It first reads state and transition history from a log file. According to this information packets are prepared and generated. At the same time this information can be used to generate an `EventData` object which will be used to "drive" the automaton. The `ReplayAction` will then replay the appropriate packet. Replies to the sent packets will be received by the replayer component.

6.6 Synthesis of network packets on application-level

For the packet replayer component shown in Figure 6.2 we have to deal with the third and last requirement, namely the requirement to have full knowledge of how packet fields are created and what their meanings are. This requirement goes further than just to have the ability to decode packets - we have to synthesize packets. Fortunately the NetPDL database definitions provide information that could be used not only for decoding but also for synthesis of network packets. Information that is needed in addition to the database field definitions could be specified together with the state machine specification for each protocol.

Chapter 7

Refactoring the Signature Generator

Besides the need for full protocol knowledge as concluded in chapter 6.1 we hinted in section 5.4 that if arbitrary protocol fields rather than the field at the packet end should be protected we run into problems due to the inflexible architecture of the first SGM. The findings in the previous chapters have important implications for the design of the signature generating component. The gathered experiences have led to a redesign of the signature generator which will be presented in this chapter.

7.1 Experiences with Signature Generation

The reasons for the rather poor signature generation mechanism presented in chapter 5 can be found in the architecture of the signature generator. The design lacks of flexibility. Each possible signature type had to be hardcoded. After intensive tests it became clear that it is necessary to support different signature types. The goal is to be able to protect an arbitrary protocol field against triggering a buffer overflow in the vulnerable server application. This is the most complex type of signature that will currently be supported. But sometimes the signature generator is not able to generate the above signature type by reason of missing information etc. In this case we have to sidestep the generation of the complete signature and come up with a simpler type like identifying the packet by an LCS string as it is presented in the chapter 5.
Because protocol knowledge allows us to omit protocol-specific signature generation code we do not know for which protocol a signature will be generated. This drawback has to be taken into account in the design of the signature generator. We must have generic data types which rather describe signature properties than properties of a protocol. This rethinking originates from the fact that the signature generator changed from protocol-dependent to generic signature generation due to protocol-knowledge.
For this reason two classes have been introduced which describe parts of a possible signature:

PacketIdentifier The packet identifier is responsible to provide information about the manner of how the appropriate message of a certain protocol is identified. For instance in the case of the FTP protocol this class would tell us that the signature is only valid if the FTP message contains the string 'USER'. This way we can narrow the network packets for which a signature matches. Because identifying the message type differs between protocols the class has to be able to handle different identification types.

VariableField This class describes the field wich triggered the buffer overflow. The name already indicates that the field is of variable length since this is usually the only way to trigger a buffer overflow. Because various types of fields with variable length exist at least the most common ways of how to define a field of variable length have to be supported.

7.2 Integrating NetBee

The two façade classes `PacketDecoder` and `DbFieldDecoder` introduced in the previous chapter 6 are used to add protocol knowledge to the signature generator. Because these classes are tightly related to signature generation and the signature generator heavily relies on the NetPDL database and the NetBee library, the class `DbFieldDecoder` is able to create objects of type `VariableField` if it identifies a field of variable length for a search in the NetPDL database.

7.3 Flexible configuration

The refactored signature generator is able to work offline. That means that the log files for Argos and the Tracker can be specified manually. Online means that the signature generator receives alerts via socket communication. In the configuration file of the signature generator the Tracker version that will be used can be specified. Because the Tracker will be rewritten to support the requirements from chapter 6 there will coexist two versions of the Tracker[1]:

- The previous version written in preceding semester theses will be called *marcel* corresponding to its last major contributor.
- The new version will be called *pascal* according to the name of its inventor.

It is also possible to generate signatures without a Tracker by specifying any other version than the two above. In this case as much information as possible will be extracted solely from the Argos log files.

7.4 Architecture

The architecture will be explained in different steps. We recall from listing 5.1 how information for signatures has been extracted before refactoring the signature generator. Each time an alert from Argos was received the extraction function was called. Since this produces a bottleneck the concept has been changed to a multi-threaded approach. If an alert is received from Argos we will start a new extraction thread. This is done the following way:

Listing 7.1: Starting an extraction thread

```
// Some setup
ThreadPool* threadPool = new ThreadPool;
InterfaceBean* interfaceBean = new InterfaceBean;
...
// We receive an alert from Argos
ExtractionThread* extractionThread = new ExtractionThread(interfaceBean, logName, rid);
extractionThread->setThreadPool(threadPool);
extractionThread->start();
```

The class `InterfaceBean` holds various information such as configuration properties, e.g. where the log files are kept and how the signature database can be accessed. A thread may be registered to an instance of `ThreadPool`. This thread pool can be used to terminate threads that have not finished at a certain point in time. In addition to the `IntefaceBean` object the file name of the Argos alert file and the Argos random alert identifier are expected by the `ExtractionThread`. Figure 7.1 shows the relationship between the above introduced classes. The class diagram reveals that the threads in the signature generator are derived from the abstract class `InterfaceThread` which actually wraps the C-API functions of the `pthread` library. Subclasses of this class have to implement the abstract functions `run()` and `dispose()`. The function `setThreadPool(ThreadPool*)` can be used to assign a thread pool to the thread instance. The mechanism behind this assignment corresponds pretty much to the *observer pattern* also called *publish and subscribe pattern*.
If an extraction thread has been started the information extraction will be done by an `Extractor` object and the desired signatures will be created. Listing 7.2 shows the most important steps that are done during signature generation. Some code pieces will look familiar to some fragments in listing 5.1.

[1]The reader has to forgive the author the slight trace of geek humor...

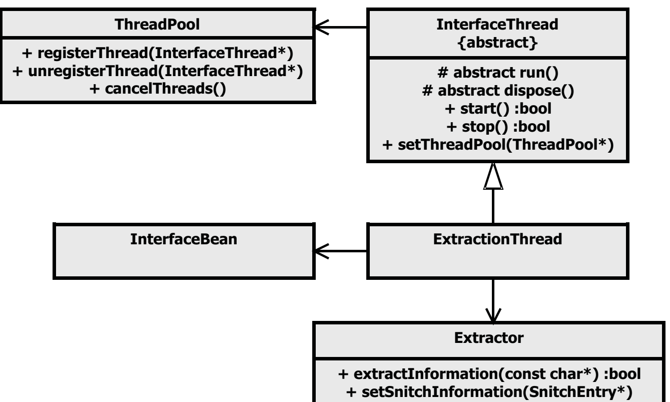


Figure 7.1: Class diagram that shows the relations between threads and information extraction

Listing 7.2: Extracting information and generating signatures

```
try {
    PacketDecoder packetDecoder;
    DbFieldDecoder fieldDecoder;
    SignatureDatabase signatureDb(interfaceBean);
    Extractor extractor(&packetDecoder, &fieldDecoder, &signatureDb, interfaceBean);

    if (extractor.extractInformation(argosAlertFile->c_str()))
    {
        // 1. Access synchronized map
        SnitchEntry* currentEntry = NULL;
        unsigned int sidTmp = 0;
        int count = 0;
        while ( !snitchMap->unbagElement(argosRid, &currentEntry) )
        {
            sleep(5); // sleep for 5 seconds and try again
            count++;
            if (count > 5) // we won't wait any time longer
                break;
        }

        SignatureHeader* signatureHeader = extractor.getCurrentSignatureHeader();
        GenericSignature* genericSignature = extractor.getCurrentSignature();
        SnortSignature* snortSignature =
            new SnortAdapter(&signatureDatabase, genericSignature, signatureHeader);

        ...
    }
    catch(ConstructorFailed& aConstructorFailed)
    {
        ...
    }
    catch(Extractor::ExtractionFailed& extractionFailed)
    {
        ...
    }
}
```

The mechanism of unbagging a `SnitchEntry` object remained the same as described in the first version of the signature generator. However the `Extractor` is no longer a container which holds the extracted information. Instead it writes the information into instances of the classes `SignatureHeader` and `GenericSignature`. These objects can be obtained by invoking the corresponding *getter functions*. In order to create a concrete signature, in our case a Snort signature, we use the class `SnortAdapter` which adapts the above by the extractor created objects to a `SnortSignature` object. This methodology corresponds to the *adapter pattern*. Note that in the refactored signature generator *exception handling* is used.
The class relations for the signature related data types are depicted in the class diagram in Figure 7.2. The two *interfaces* have to be implemented if a signature should have access functionality for the database. The class `SignatureDatabase` provides some basic functionality to access the database, execute queries, etc. The abstract class `SignatureBase` rep-

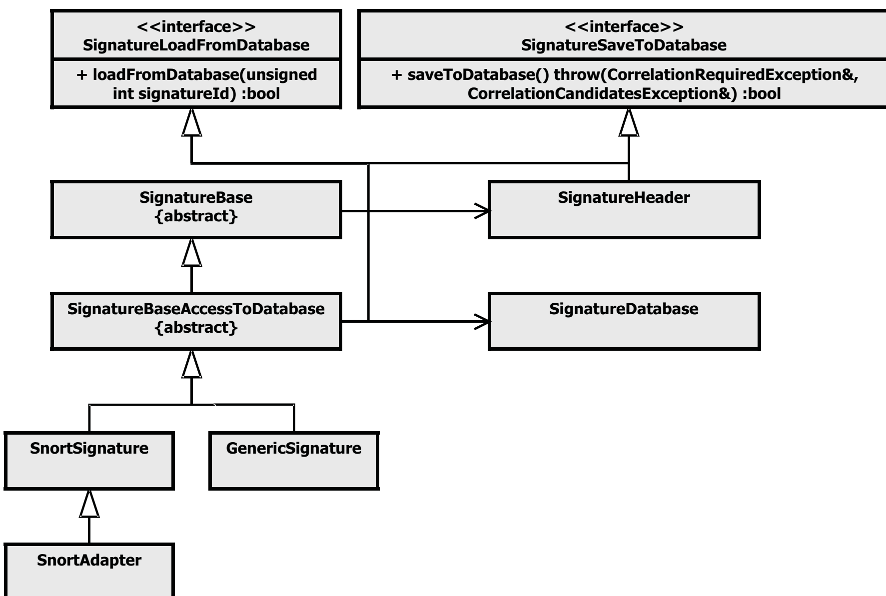


Figure 7.2: The class relations for the signature related data types

resents the abstract type of a signature. Each signature has to be related to an instance of `SignatureHeader`. This class represents the head of a signature and is used to identify a signature. The class `SignatureBase` together with the two *interfaces* build an abstract signature with database capability. Classes for a Snort and a generic signature are derived from it. The Snort signature class has a subclass which can be used to adapt a generic signature to a Snort signature. Note that the classes `SignatureHeader`, `GenericSignature` and `SnortSignature` have a database table counterpart. The database table definitions can be found in the appendix G. As the signature generator is under ongoing development the tables still may be altered.

7.5 Supported signatures

If the extractor is started it tries to get as much information as possible. But if e.g. no Tracker log file is available the packet identifier may not be complete. Then a weaker signature is created. Subsequently the different possible Snort signatures are listed.

Signature with LCS byte string:

```
alert tcp any any -> any 21 (content:"|05 D3 FF 00 09 A4 C6 12|";)
```

Signature with string packet identifier and LCS string:

```
alert tcp any any -> any 21 (content:"USER"; content:"|09 A4 C6
12|";)
```

Signature with type field packet identifier and LCS string:

```
alert tcp any any -> any 21 (byte_test: 4, =, 2, 70; content:"|09 A4
C6 12|";)
```

Signature with an open-ended protocol field:

```
alert tcp any any -> any 21 (content:"USER"; dsize:>32;)
```

Signature with string packet identifier and token-ended variable-length field:

```
alert tcp any any -> any 21 (content:"USER"; content:!"|0D 0A|";
offset: 5; depth: 465;)
```

Signature with type field packet identifier and variable-length field specified by another field:

```
alert tcp any any -> any 21 (byte_test: 4, =, 2, 70; byte_test: 4,
>, 450, 66;)
```

Two possible signatures have been omitted because they follow from the last two presented signatures. The first one consists of a string identifying the message type and a variable-length field specified by another field. The second signature omitted consists of a binary field identifying the message type and a content string of variable length.

7.6 Evaluation

7.6.1 Measuring signature generation performance

For estimating the average delay for generating a signature ten different test runs have been done for four distinct attacks recognized by Argos. The measurements have been done offline. For measuring the signature generation delay the same methodolgy has been used as for the Tracker evaluation; performance measurement can be enabled by using a precompiler directive to set the variable `PERFORMANCE_MEASUREMENT`. Figure 7.3 depicts the test runs. The measured delay incorporates both the information extraction and the adaption of the generic signature to a Snort signature. From the graphic the average delay for the generation of a signature can be rated to about 850 milliseconds. Note that parsing the log file of the Tracker, correlating the Tracker log files with information from Argos and the inter-process communication with the reporting thread of the Tracker is not incorporated in these measures as they do not belong to the actual signature generation. On the other hand the time needed for initializing the NetPDL library is included in these measures. Parsing the XML file of the NetPDL protocol definitions certainly has a major part of the overall delay for generating a signature. The delays shown give an idea of the induced delay by the signature generation. However, the delay may change significantly if any of the above tasks are incorporated or omitted in the signature generation.

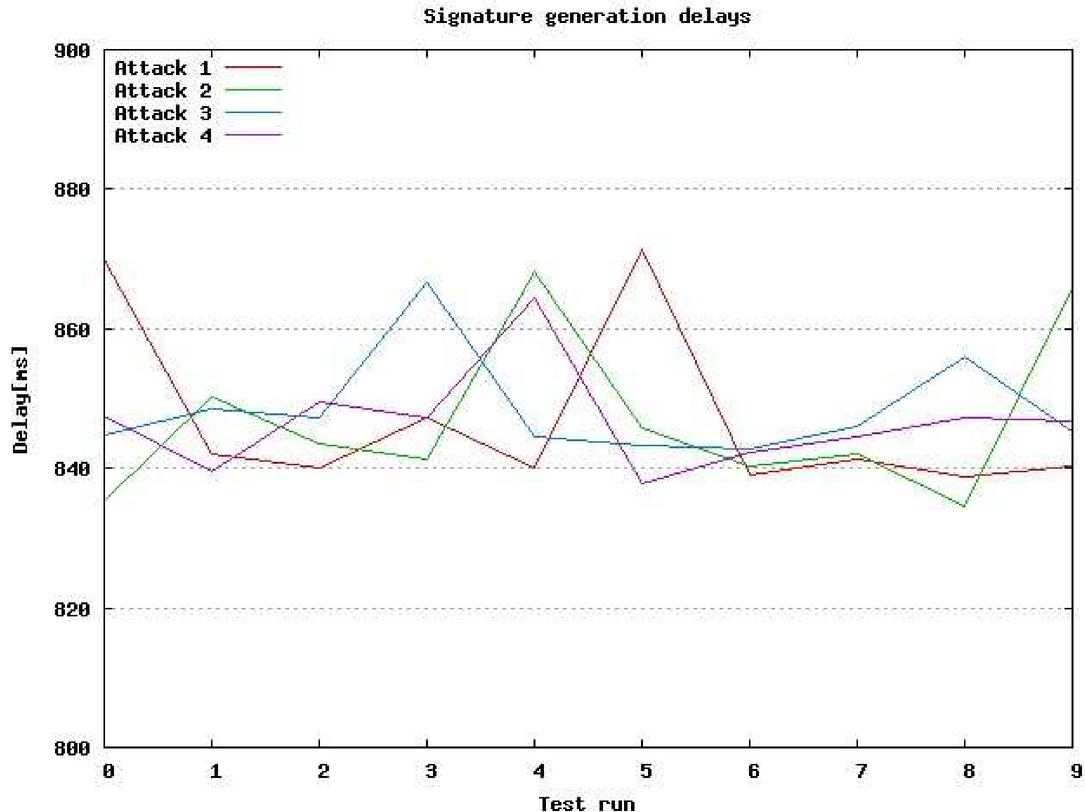


Figure 7.3: The delays for generating different signatures

Chapter 8

Conclusions and Outlook

8.1 Results

In this section the potential of our signatures is compared to some approaches presented in the chapter 3. The NoAH infrastructure filters network traffic very thoroughly before it gets to the containment system. Still more important, Argos only raises an alert when an attack actually took place. In this case it provides us with precise information about the system state. If the condition leading to the alert can be mapped to a network packet we have a richness of information that can be used to describe the attack that has occurred. Currently we use this information to describe precisely which application has a buffer overflow vulnerability and how it can be exploited. This approach for protecting server applications selectively from buffer overflow exploits is of inestimable value. Productive server systems crucial to failures can be protected from zero-day exploits if the NoAH infrastructure is able to observe the appropriate exploit first.
There are a few approaches with a containment system similar to Argos like e.g. TaintCheck [13] or Vigilante [20]. TaintCheck however creates very weak string-based signatures out of its extracted attack information. Vigilante creates its own format of self-certifying signatures. Another interesting approach is COVERS [37]. It pursues a signature generation mechanism similar to ours in that after detecting an attack the victim's process memory is compared to recent input data. A drawback of this approach seems to be that parsing input data after correlation with memory depends on a special message format language. Remember that specifying dozens of protocols by ourselves would be inefficient and error-prone. Instead we rely on the NetPDL protocol library. Furthermore with the tracker framework we have not only protocol-field knowledge but even protocol-state knowledge. The position-aware distribution signatures (PADS) system presented in [36] produces signatures based on byte-frequency distributions. These type of signature however can not be transformed into a Snort signature. Systems like Autograph [9] or PAYL [16] generate signatures based on byte strings. These signatures normally implicate high false positives and are not capable of detecting altered attack code.
It is reemphasized that there are mainly two mechanisms that contribute to a signature's quality. The first one is the type of attack detection that is used to extract relevant information for signature generation. The second aspect is the type of signature that is used to represent the extracted information. It is important not to mistake the above signature type with a concrete signature format. The latter is just a transformation of the former into a representation compliant with an existing signature format, e.g. the Snort signature format.

8.2 Conclusions

The first mandatory subtask of this thesis was to design, implement and evaluate an application protocol plugin for the application state tracking framework. The evaluation has shown that the requirements have been fulfilled and the implementation was successful. The plugin has been tested for memory leaks with different tools such as valgrind and memprof and showed no significant runtime dependent problems. Performance tests have shown an average delay induced by the FTP plugin for processing packets about 26 microseconds. This delay corresponds to a one-digit percentage compared to the overall delay for processing a packet with the Tracker.

The correct functioning of the state machine has been approved theoretically and practically by simulated FTP conversations.
A simple signature generator correlating information from Argos and the tracker framework has been designed, implemented and demonstrated at a NoAH meeting. The generated signatures and the capabilities of the generation mechanism however remained at a proof-of-concept level.
From the PoC implementation of the above two subtasks different observations have been made. The most important observations are summarized below.

- Reliability and accuracy of the generated signatures were not a question of the chosen concrete signature format, in our case the Snort format. Rather the difficulty was to extract precise and reliable information and map the information into internal representations that can be used by a SGM.
- Signature evaluation is a hard topic and is at least questionable in our case of only exemplarily available exploits.
- The initial design of the signature generator was too inflexible with respect to the internal representation of signature properties and the signatures themselves. This led to a thorough redesign of the entire signature generation component.
- The architecture of the tracker framework proved as inflexible too. The hardcoded protocol plugins were not accurate. From this and the preceding arguments a more generic approach for protocol state tracking, signature generation and further purposes evolved.

The redesigned architecture for generating signatures creates attack signatures in a flexibly manner. The complexity and accuracy of the generated signatures depend on information that was extracted from both the honeypot network and a Tracker log. If the detected attack is a buffer overflow which was triggered by abusing a single protocol field, the false-positives rate for these signatures is almost zero since the signatures describe the characteristics of the buffer overflow vulnerability. If for some reasons the extracted information is incomplete, emergency signatures are generated, e.g. based on byte strings. This assures that we always generate a valid signature. Due to different methods for extracting attack information we are capable of predicting the quality of the generated signatures during information extraction.

8.3 Outlook

In summary a system for automated signature generation imposes a large effort and results in a complex system. Flexibility and generic configurability are the catchwords for such a system. What about a design that only knows generic ”properties”? Each property is configured by a specification language and tells the information extractor how it is extracted. From this set of properties another specification would tell an adapter how to transform different properties in order to synthesize a specific and concrete signature, e.g. for the Snort IDS as in our case. Whereas this suggestion is not in the scope of this thesis more immediate but small tasks should be done first. The appendix H contains a categorized list of todo’s for the current architecture. The most important steps will be to thoroughly finish the signature generator and add a state tracking action component for the generic state machine. Different details have to be clarified such as what happens if a protocol field is stream-based and the field data could be distributed over several network packets. What has to be done to support signature generation for reassembled packets? Are there any drawbacks when relying on the NetBee library? How could the design get adapted for other possible attack types except the supported buffer overflows, e.g. format string exploits? These questions or subtasks are just an assortment and the list could be expanded arbitrarily.

Appendix A

NoAH core components: Extraction and Processing of attack information

In this chapter the different information available from both, Argos and the Tracker, is explained. Illustrations help to clarify which component provides which information. In the first section the initial state of information extraction for attacks is described. In the following section the effects of this thesis on the information extraction and processing are depicted.

A.1 Initial attack information

Argos detects when tainted data is about to be executed or used as an operand for an instruction. In this case of remote code execution the network packet from which the tainted data originated is identified. Furthermore the position of the tainted data in the appropriate packet is determined. Summarized, Argos provides information about the system's state, e.g. a memory dump, the set of processor register values as well as the network packet and data therein responsible for triggering an attack alert. Figure A.1 illustrates the information provided by Argos. The snitch perl script tells us which application was attacked on which platform. The purpose of

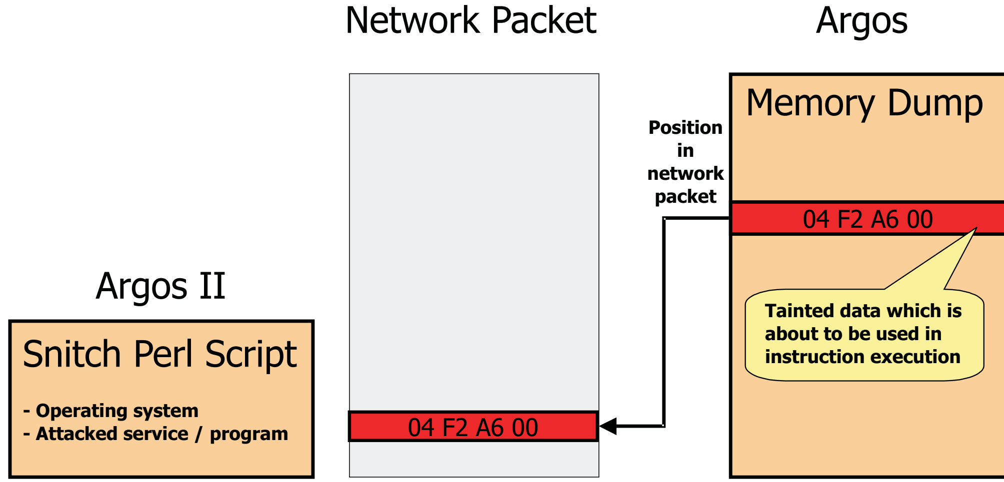


Figure A.1: Information provided by the containment system Argos

the Tracker is to provide us with information additional to the one given by Argos. It mainly has information about the packet fields and protocols contained in a network packet and in which state the corresponding protocols have been for this network packet. Note that this information could be extracted also on demand. This means that we could inspect a packet history just

when an alert is raised instead of tracking the packets before they arrive at Argos. However, this has the drawback that the time needed to generate a signature increases significantly. Figure A.2 depicts the two sources for information about the attack, Argos and the Tracker. In the initial

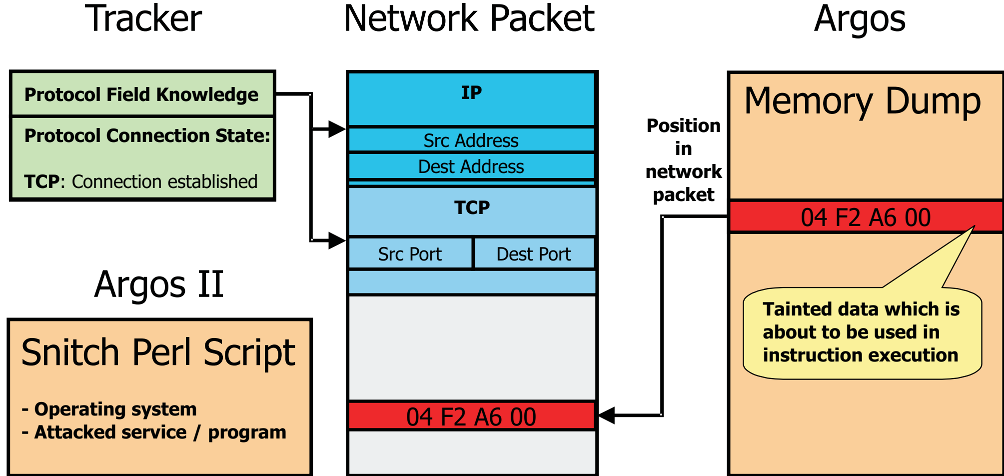


Figure A.2: Additional packet information provided by the tracker framework

state no application protocol plugin was implemented. If Argos points to a position inside the network packet which belongs to an application protocol field, no information about the protocol would be available at this time.

A.2 Current state of attack information extraction

After implementing a FTP plugin for the Tracker it was possible to fully decode packets containing an attack for the FTP protocol. Figure A.3 depicts this situation. At a later point in time

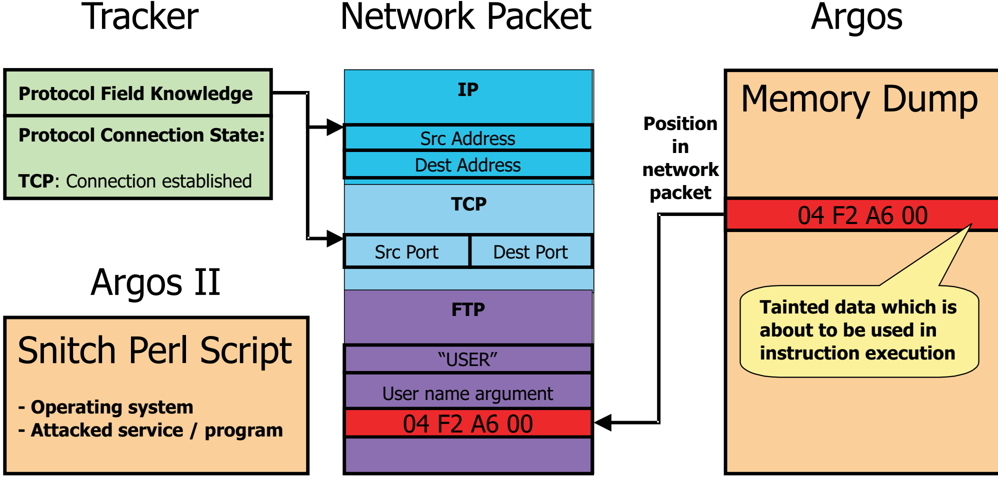


Figure A.3: FTP protocol knowledge by a Tracker plugin

the Tracker plugins have been substituted by the NetBee library and a generic state machine. However the situation depicted remains the same. In figure A.4 the current process of attack information extraction is illustrated. In a first step we use the network packet identified by Argos which was the origin for the tainted data triggering the attack. Comparing the network packet history from the Tracker and Argos we can find the same packet in the Tracker network log file. Applying an LCS algorithm on the network packet and the memory dump of Argos allows us to extract a byte string. This byte string can be used for signature generation in case more sophisticated signature generation fails. The network packet in the Tracker log file is then used

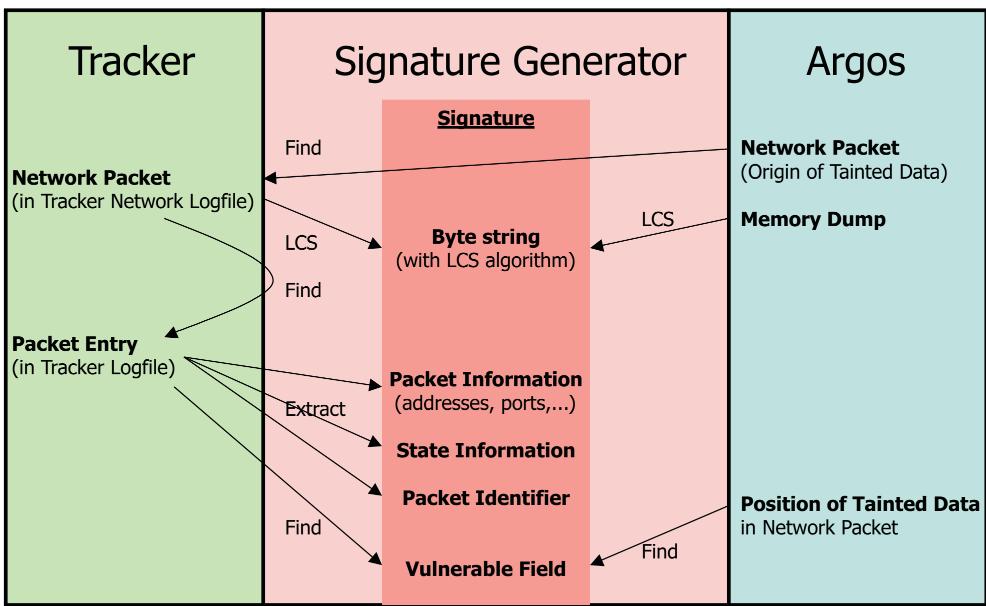


Figure A.4: Overview of the information extraction process

to find its related entry in the second Tracker log file. This second log file contains all the information extracted by the plugins. Out of the packet log entry information about the packet can be extracted. Information about the protocols, their fields and connection states is available. A packet identifier tells us how this network packet is identified, e.g. by containing a certain string. At last the position of the tainted data triggering the attack alert is used to locate the vulnerable protocol field.

Appendix B

Glossary

In this glossary the most important terms from the field of computer networking and security are explained. The term explanations do not intend to qualify for completeness. They rather should give an idea of the corresponding terms. Most of the definitions are taken from the Wikipedia online encyclopedia[1]. Some definitions may have been reworked or supplemented with additional information.

Activation zone *Where the payload is executed. It produces output events (indicating that the desired outcome of an attack has occurred) and feedback events (that the attacker can determine whether an attack was successful).*

Address Space Randomization *Despite the wide publicity received by buffer overflow attacks, a vast majority of today's security vulnerabilities continue to be caused by memory errors, with a significant shift away from stack-smashing exploits to newer attacks such as heap overflows, integer overflows, and format-string attacks. Address space randomization hinders some types of security attacks by preventing an attacker being able to easily predict target addresses. For example attackers trying to execute return-to-libc attacks must locate the code to be executed; while other attackers trying to execute shellcode injected on the stack have to first find the stack. In both cases, the related memory addresses are obscured from the attackers; these values have to be guessed, and a mistaken guess is not usually recoverable due to the application crashing.*

Attack *A principal is able to gain unauthorized access to a computer system or to disrupt its operation in an unintended way. An attack is the exploitation of existing bugs.*

Attack vector *Every interface is a possible vector of attack, e.g. user interface, I/O interface, physical or network interfaces etc.*

Computer worm *A computer worm is a self-replicating computer program. It uses a network to send copies of itself to other nodes on the network and it may do so without any user intervention. Unlike a virus, it does not need to attach itself to an existing program. Worms always harm the network and if only by consuming bandwidth, whereas viruses always infect or corrupt files on a targeted computer.*

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

Design pattern *A design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application, classes or objects that are involved.*

Dynamic Taint Analysis *In computer systems with dynamic taint analysis input data from untrusted sources is normally marked as tainted. When tainted data is used in operations it may be possible that other data becomes tainted too. When monitoring the programs processing tainted data it is then possible to track how the tainted data propagates in memory. If any tainted data is used in an illegal manner program execution could be stopped. Furthermore a lot of useful information such as a memory dump etc. could be logged for analysis.*

False positives *Generally spoken a false positive occurs when the null hypothesis is rejected given that it is actually true. The null hypothesis is a hypothesis that is presumed to be true until statistical evidence in the form of a hypothesis test indicates the opposite. The null hypothesis is often the reverse of what the experimenter actually belives. A null hypothesis has to be nullified or refuted. In our case of testing signatures we have a false positive when a network packet matches a certain signature and thus the network packet is supposed to carry an exploit notwithstanding it actually belongs to malign network traffic. Our null hypothesis H_0 would amount to ”the packet contains no exploit”. A false positive rejects H_0 although the packet contains no attack and so the packet content is mistakenly suspected to be malicious by matching to the signature.*

Firewall *A firewall is a hardware or software device which is configured to permit, deny, or proxy data through a computer network which has different levels of trust.*

Flaw *In computer literature a software bug is normally regarded as an implementation-level software problem whereas a flaw is a problem originating at the design level and being instantiated in software code. A flaw may or may not be exploitable.*

Flow *In general, flows are a ”set of packets which share a common property.” The most important properties are the flow’s endpoints. For example, the simplest type of flow is a 5-tuple, with all its packets having the same source and destination IP addresses and port numbers. Furthermore, 5-tuple flows are unidirectional, i.e. all their packets travel in the same direction. Such 5-tuple flows are commonly referred to as microflows. Flows may have two endpoints, (TCP from host A to host B), or only one (all UDP flows from host C). Endpoints may also be more general, for example ”TCP from network X/20 to network Y/24”. A flow begins when its first packet is observed, but one should state how to recognise the end of a flow. The most common method is to specify a fixed timeout, alternatively one can specify a dynamic timeout algorithm. Examples for flow types are CPB, CoralReef, NetFlow v5 etc., RTFM, and NeTraMet stream. Cisco’s NetFlow format uses the common 5-tuple definition, where a flow is defined as a unidirectional sequence of packets all sharing all of the following 5 values: source IP address, destination IP address, source TCP port, destination TCP port and IP protocol.*

Honeypot *A honeypot is a trap set to detect, deflect or in some manner counteract attempts at unauthorized use of information systems. Generally it consists of a computer, data or a network site that appears to be part of a network but which is actually isolated, (un)protected and monitored, and which seems to contain information or a resource that would be of value to attackers. A honeypot is valuable as a surveillance and early-warning tool. While often a computer, a honeypot can take on other forms, such as files or data records, or even unused IP address space. Honeypots should have no production value and hence should not see any legitimate traffic or activity. Whatever they capture can then be surmised as malicious or unauthorized.*

Injection vector *Describes the format of an input-driven attack. Each target environment imposes certain restrictions on how an attack must be formatted. The goal of the injection vector is to place the attack payload into a target activation zone. They comprise truly generic rules for formatting an attack.*

Intrusion detection system *An intrusion detection system is used to detect many types of malicious network traffic and computer usage that can not be detected by a conventional firewall. This includes network attacks against vulnerable services, data driven attacks on applications, host based attacks such as privilege escalation, unauthorized logins and access to sensitive files, and malware (viruses, trojan horses, and worms). An IDS is composed of several components: Sensors which generate security events, a Console to monitor events and alerts and control the sensors, and a central Engine that records events logged by the sensors in a database and uses a system of rules to generate alerts from security events received. There are several ways to categorize an IDS depending on the type and location of the sensors and the methodology used by the engine to generate alerts. In many simple IDS implementations all three components are combined in a single device or appliance.*

Intrusion prevention system *An intrusion prevention system is a computer security device that monitors network and/or system activities for malicious or unwanted behavior and can react, in real-time, to block or prevent those activities. Network-based IPS, for example, will operate in-line to monitor all network traffic for malicious code or attacks. When an attack is detected, it can drop the offending packets while still allowing all other traffic to pass. Intrusion prevention technology is considered by some to be an extension of intrusion detection (IDS) technology.*

Network Address Translation *In computer networking, the process of network address translation (NAT, also known as network masquerading, native address translation or IP masquerading) involves re-writing the source and/or destination addresses of IP packets as they pass through a router or firewall. Most systems using NAT do so in order to enable multiple hosts on a private network to access the Internet using a single public IP address.*

Open Systems Interconnection (OSI) model *The open systems interconnection basic reference model (OSI reference model or OSI model for short) is a layered, abstract description for communications and computer network protocol design, developed as part of the open systems interconnection (OSI) initiative. It is also called the OSI seven layer model. A layer is a collection of related functions that provides services to the layer above it and receives service from the layer below it. For example, a layer that provides error-free communications across a network provides the path needed by applications above it, while it calls the next lower layer to send and receive packets that make up the contents of the path. The seven layers as defined by the OSI model ordered from the top are application, presentation, session, transport, network, data link and physical layer.*

Polymorphic worm *Polymorphic code is code that mutates while keeping the original algorithm intact. This technique is often used by computer viruses, shellcodes and computer worms to hide their presence. A polymorphic worm uses the technique of polymorphic code mutation. Polymorphic algorithms make it difficult for such software to locate the offending code as it constantly mutates. Encryption is the most commonly used method of achieving polymorphism in code. However, not all of the code can be encrypted as it would be completely unusable. A small portion of it is left unencrypted and is used to jumpstart the encrypted software. Anti-virus software targets this small unencrypted portion of code.*

Proxy server *A proxy server is a server (a computer system or an application program) which services the requests of its clients by forwarding requests to other servers. A client connects to the proxy server, requesting some service, such as a file, connection, web page, or other resource, available from a different server. The proxy server provides the resource by connecting to the specified server and requesting the service on behalf of the client. A proxy server may optionally alter the client's request or the server's response, and sometimes it may serve the request without contacting the specified server.*

Software bug *A software bug (or just "bug") is an error, flaw, mistake, failure, or fault in a computer program that prevents it from behaving as intended (e.g., producing an incorrect result). Most bugs arise from mistakes and errors made by people in either a program's source code or its design, and a few are caused by compilers producing incorrect code.*

Software exploit *An exploit is a piece of software, a chunk of data, or sequence of commands that take advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized).*

Telnet protocol *Telnet (Teletype Network) is a network protocol used on the Internet or local area network (LAN) connections. The initial protocol specification can be found in RFC 854. The introduction therein describes the purpose of the protocol in the following way: "The purpose of the Telnet Protocol is to provide a fairly general, bi-directional, eight-bit byte oriented communications facility. Its primary goal is to allow a standard method of interfacing terminal devices and terminal-oriented processes to each other."*

Virus *A computer virus is a computer program that can copy itself and infect a computer without permission or knowledge of the user. The original may modify the copies or the copies may modify themselves, as occurs in a metamorphic virus. A virus can only spread from one computer to another when its host is taken to the uninfected computer, for instance by a user sending it over a network or carrying it on a removable medium such as a floppy disk, CD, USB drive or by the Internet. Additionally, viruses can spread to other computers by infecting files on a network file system or a file system that is accessed by another computer. Viruses are sometimes confused with computer worms and Trojan horses. A worm can spread itself to other computers without needing to be transferred as part of a host, and a Trojan horse is a file that appears harmless until executed.*

Appendix C

The original tracker framework

In this chapter the components and their functions of the Tracker framework will be presented. Furthermore the skeletons of important functions will be sketched.

C.1 Component overview

There coexist multiple threads in the Tracker. The functionality is divided into components each corresponding to a single thread. The explanation of the functionality will be split into two phases, the initialization phase and the packet processing phase.

The initialization phase

Figure C.1 illustrates the components and functions participating during the initialization of the Tracker. The main function first reads the configuration file. According to the configura-

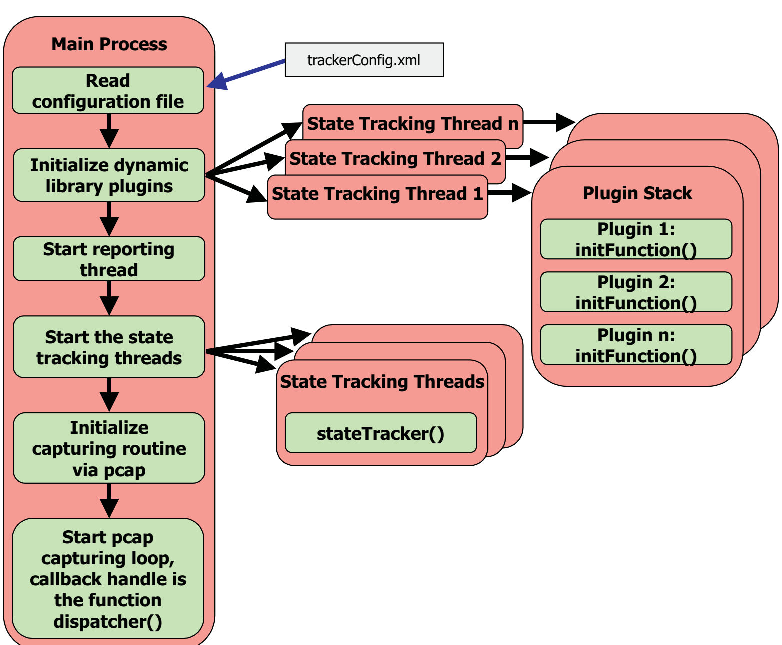


Figure C.1: The initialization phase of the Tracker

tion settings the desired number of state tracking structures is created. Each state tracking structure gets its own plugin stack. As mentioned in chapter 2 the plugins are initialized by calling `initFunction`. When the data structures have been created the reporting thread will be started. Following the state tracking threads are started. These threads remain inside the function `stateTracker` and wait for incoming messages from the `dispatcher` function in the main process.

The packet processing phase

The Packet Processing phase is depicted in Figure C.2. When a packet arrives at the state

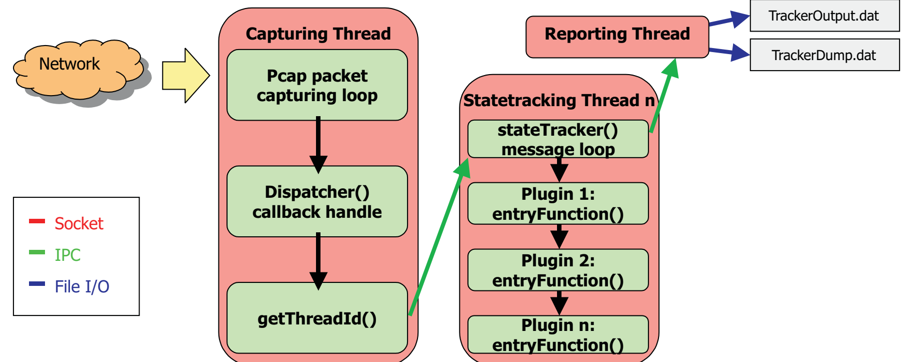


Figure C.2: The packet processing phase of the Tracker

tracking host the pcap packet capturing loop will create some data structures and invoke the callback function `dispatcher`. Inside the `dispatcher` function the packet is analyzed and if the packet belongs to an already registered network connection the network packet is sent via message passing to the appropriate state tracking thread. If no network connection can be found for the received packet a new connection for this packet will be registered and assigned to a randomly chosen thread. The packet receiving thread invokes the first plugin by calling its function `entryFunction` and passes the network packet by reference. Each plugin is responsible to forward the network packet to the next appropriate plugin when it has done its processing steps. When no suitable plugin is found anymore in the plugin stack the state tracking loop will pass a reporting buffer to the reporting thread and wait for new network packets. The reporting thread itself will write the received buffer to a log file betimes.

C.2 Components and their algorithms

In this section the basic functionality of the most important functions of the Tracker framework is presented. For an overview of the entire set of functions and a short description refer to the generated source code documentation. The source listings are not complete. They rather intend to allow the reader being able to orient easier in the Tracker source code.

Main Process

The most important steps done in the *Main Process* are illustrated in Figure C.1. The listing C.1 shows the code corresponding to the illustrated steps during initialization.

Listing C.1: The most important steps in the *Main Process*

```
// Read configuration file
configData allConfigData = scanConfigFile(file);

// Initialize dynamic library plugins
initDynamicLibrary(&allConfigData);

// Start the reporting thread
aReportingThread =
initReportingThread(&stateTrackerThreads[0].outgoingMessageQueue,
&allConfigData);

// Start the state tracking threads
stateTrackerThreads = initStateTrackerThreads(&allConfigData);

// Initialize capturing routine
captureHandle = initCapture(allConfigData.device, filter_exp);

// Global available information used by state tracking threads
```

```
dispatcherExtraArguments   someDispatcherExtraArguments;
someDispatcherExtraArguments.captureHandle = captureHandle;
someDispatcherExtraArguments.allConfigData = &allConfigData;
someDispatcherExtraArguments.connections = &connections;
someDispatcherExtraArguments.fragmentedPackets =
&fragmentedPackets;
someDispatcherExtraArguments.stateTrackerThreads =
stateTrackerThreads;

// Register necessary signal handlers...

// Start pcap capturing loop with dispatcher() as callback
pcap_loop(captureHandle,-1,dispatcher,(u_char*)
(&someDispatcherExtraArguments));
```

If a packet is captured by the pcap library the callback function `dispatcher` is invoked. Listing C.2 shows the most important code sequences for this function.

Listing C.2: Conceptual code excerpt for the function `dispatcher`

```
// Get the necessary global available information
someDispatcherExtraArguments = (struct dispatcherExtraArgument*)(args);

// Determine the id of the appropriate state tracking thread
// and return a reassembled packet if necessary
IPConnection* reassembledPacket = getLocalThreadIdByPacket(pkthdr,
    packet, someDispatcherExtraArguments, threadIdOfConnection);

if (threadIdOfConnection != -1) {
    // select the corresponding state tracker thread for this connection
    stateTrackerThread *aStateTrackerThread =
        someDispatcherExtraArguments->stateTrackerThreads + threadIdOfConnection;

    if(reassembledPacket)
    {
        // take info from reassembled packet if it exists
    }
    else
    {
        // normal, not fragmented packet
    }

    memcpy(message.msgText, &newPacket, sizeof(newPacket));
    messageQueueSend(aStateTrackerThread->incomingMessageQueue,
        &message, sizeof(newPacket));
} else {
        // The packet arrived is either not supported or it has been only a fragment.
        // In the latter case the localThreadId is returned only after the time the
        // last frame has arrived and the packet could be reassembled.
}
```

The function responsible for reassembling network packets if necessary and to determine if the packet already belongs to a registered network connection is the function `getLocalThreadIdbyPacket`. The basic steps executed in this function are listed in listing C.3.

Listing C.3: Conceptual code excerpt for the function `getLocalThreadIdByPacket`

```
// declare pointers to packet headers

// define ethernet header

// define/compute ip header offset

// delete entries of the IPkey map which are older than FRAGMENTATION_TIMEOUT

// if fragmented, that means more-fragments-flag is set and/or fragment offset!=0

// process packet */
// save the essential data (source port, destination port) in case of TCP / UDP

// create connectionKey/Identifier
connectionKey * connKey;
if (ipHeader->ip_src.s_addr ==
```

```
    someDispatcherExtraArguments->allConfigData->monitoredHostIpAddress.s_addr){
    connKey = new connectionKey(ipHeader->ip_dst.s_addr,*srcport,*destport);
} else if(ipHeader->ip_dst.s_addr ==
    someDispatcherExtraArguments->allConfigData->monitoredHostIpAddress.s_addr)){
    connKey = new connectionKey(ipHeader->ip_src.s_addr,*destport,*srcport);
}

// Try to find the entry in the connection map.
connectionMap::iterator connectionMapIter =
    someDispatcherExtraArguments->connections->find(*connKey);

if ( connectionMapIter == someDispatcherExtraArguments->connections->end())
{
    // the connection entry is not in the map, therefore we add it

    // assign a random threadId, get the timestamp and
    // add the new connection to the map
    aConnection->localThreadId =
        rand() % someDispatcherExtraArguments->allConfigData->numOfStateTrackerThreads;
    connectionEntry anEntry(*connKey, aConnection);
    someDispatcherExtraArguments->connections->insert(anEntry);

    // set the iterator to the new entry
    connectionMapIter = someDispatcherExtraArguments->connections->find(*connKey);
}
threadId = connectionMapIter->second->localThreadId;
return reassembledPacket;
```

State Tracking Threads

The functionality of the state tracking threads is resided in the function `stateTracker`. Each thread just awaits messages from the dispatching process and processes them. The code skeleton looks as shown in listing C.4.

Listing C.4: The code skeleton for the state tracking thread loop `stateTracker`

```
void* stateTracking::stateTracker(void* ptr) {
    while(!finish)
    {
        messageQueueReceive(thisThread->incomingMessageQueue, &readBuffer, ...)
        switch(readBuffer.msgType)
        {
            case MSQ_TYPE_IP_MESSAGE:
            {
                do
                {
                    dlerror();
                    plugin_t ip = ipProtocol->entryFunction;
                    ret = ip(...);
                    if (ret == EXIT_FAILURE)
                    {
                        /* Report buffer is dumped */
                    }
                    else
                    {
                        /* Plugins successful */
                        /* Append new entry in report buffer*/
                    }
                    msgsnd(thisThread->outgoingMessageQueue)
                }while()
            }
        }
    }
}
```

The state tracking threads invoke the first suitable plugin by calling its entry function. Each protocol is then responsible to call a succeeding plugin. The entry function mandatory to be implemented by each protocol plugin should follow the concept as shown in listing C.5.

Listing C.5: Program code guideline for the mandatory `entryFunction`

```
// Extract information of preceding protocol
```

```
// Process the packet for the current protocol
// and e.g. update state machine etc.

// Report

// Give packet to the next protocol

// Always return success, even if the packet had wrong format
int returnValue = EXIT_SUCCESS;

// The findSubProtocol function searches in the subProtocolMap
// for an entry with the key monitoredHostPort

protocol* nextProtocol = thisProtocol->findSubProtocol(monitoredHostPort);
if(nextProtocol && nextProtocol->entryFunction) {
    returnValue = nextProtocol->entryFunction(nextHeader,
        packetLength-SIZE_UDP_HEADER, NULL, nextProtocol,
        thisStateTracker, lastReportbuffer->next, packet);
}

return returnValue;
```

Appendix D

Techniques in Signature Generation Mechanisms

The following presented techniques have been used in one or more signature generation mechanisms presented in this documentation. If not specified explicitly the main source can assumed to be http://en.wikipedia.org.

D.1 Byte-Frequency Distribution (BFD)

The byte frequency distribution is a function $f_p(b)$, which describes the probability for b to appear at position p. b is between 0 and 255. The sum of $f_p(b)$ for all possible bytes (the values from 0 to 255) is 1. If W is the width in bytes of the signature, $(f_1, f_2, \ldots, f_W)$ represents a simple signature based on BFDs.

D.2 Longest Common Substring (LCS) problem

The longest common substring problem is to find the longest string (or strings) that is a substring (or are substrings of) two or more strings. It should not be confused with the longest common subsequence problem. You can find the lengths and starting positions of the longest common substrings of S and T in $O(n+m)$ with the help of a generalised suffix tree. Finding them by dynamic programming costs $O(nm)$. The solutions to the generalised problem take $O(n_1 + \ldots + n_K)$ and $O(n_1 \cdot \cdots \cdot \ldots \cdot \cdots \cdot n_K)$ time.

D.3 Longest Common Subsequence (LCSeq) problem

The problem of the longest common subsequence (LCSeq) is to find a longest sequence which is a subsequence of all sequences in a set of sequences (often just two). The problem is sometimes defined to be finding all longest common subsequences. The problem is NP-hard for the general case of an arbitrary number of input sequences. When the number of sequences is constant, the problem is solvable in polynomial time by dynamic programming.

D.4 N-Gram analysis

N-Grams are used to describe objects as vectors. This makes it possible to apply geometric, statistical and other mathematical techniques on objects although the techniques are only defined for vectors.
Definition: A is an alphabet and $|A|$ is the cardinal number of the alphabet. Let n be an integer. An *N-gram* is a word of length n.
If you determine the number of occurrences of an *N-gram* within a possible sequence of characters of an alphabet you get a *N-gram frequency vector* for this sequence. For comparing two

such vectors you could for instance apply the *Manhattan*, *Cosine* or *Euclidian* distance. Another metric is the *Z-score*.

D.4.1 Z-score

In statistics, a *standard score* (also called *z-score* or *normal score*) is a dimensionless quantity derived by subtracting the population mean from an individual (raw) score and then dividing the difference by the population standard deviation. The z-score reveals how many units of the standard deviation a case is above or below the mean. If the mean and deviation can not be known then may be one could assume that the subject is normally distributed.

D.4.2 Dice coefficient

A prominent algorithm to compute a similarity between words is the *Dice algorithm*. The *Dice coefficient* of two terms a and b is defined by:

$$d(a,b) = \frac{2|\Gamma(a) \cap \Gamma(b)|}{|\Gamma(a)| + |\Gamma(b)|}$$

where $\Gamma(x)$ is a N-gram decomposition of the term d. d is between 0 and 1.
An example:
Let Term a = "wirk" and Term b = "work". If we use Tri-grams the decomposition looks as follows:

$\Gamma(a) = \{w, wi, wir, irk, rk, k\}$ and $\Gamma(b) = \{w, wo, wor, ork, rk, k\}$

The Dice coefficient is $d(wirk, work) = \frac{2 \cdots 3}{6+6} = 0.5$. So you can say the similarity is 50 percent.

D.5 Expectation-Maximization (EM) algorithm

An expectation-maximization (EM) algorithm is used in statistics for finding maximum likelihood estimates of parameters in probabilistic models, where the model depends on unobserved latent variables. EM alternates between performing an expectation (E) step, which computes an expectation of the likelihood by including the latent variables as if they were observed, and a maximization (M) step, which computes the maximum likelihood estimates of the parameters by maximizing the expected likelihood found on the E step. The parameters found on the M step are then used to begin another E step, and the process is repeated.

D.6 Clustering

Cluster Analysis[1], also called data segmentation, has a variety of goals. All relate to grouping or segmenting a collection of objects (also called observations, individuals, cases, or data rows) into subsets or "clusters", such that those within each cluster are more closely related to one another than objects assigned to different clusters. Central to all of the goals of cluster analysis is the notion of degree of similarity (or dissimilarity) between the individual objects being clustered. There are two major methods of clustering, hierarchical clustering and k-means clustering.

D.6.1 Hierarchical clustering

In hierarchical clustering the data are not partitioned into a particular cluster in a single step. Instead, a series of partitions takes place, which may run from a single cluster containing all objects to n clusters each containing a single object. Hierarchical Clustering is subdivided into agglomerative methods, which proceed by series of fusions of the n objects into groups, and divisive methods, which separate n objects successively into finer groupings.

[1] from http://www.resample.com/xlminer/help/HClst/HClst_intro.htm

D.7 Rabin fingerprint

The Rabin fingerprinting scheme is a method for implementing public key fingerprints using polynomials over a finite number of elements. Given an n-bit message $m_0, ..., m_{n-1}$, we view it as a polynomial of degree $n-1$:

$$f(x) = m_0 + m_1 x + ... + m_{n-1} x^{n-1}$$

We then pick a random irreducible polynomial $p(x)$ of degree k, and we define the fingerprint of m to be

$$f(x) \text{ mod } p(x)$$

which can be viewed as a polynomial of degree $k-1$ or as a k-bit number.
For further information consider the original document about the Rabin fingerprint [29]. [4] gives an overview of applications of the above presented fingerprinting method.

D.8 Bayes law

The Bayes law is a result in probability theory, which relates the conditional and marginal probability distributions of random variables. In some interpretations of probability, the Bayes law tells how to update or revise beliefs in light of new evidence a posteriori.
The probability of an event A conditional on another event B is generally different from the probability of B conditional on A. However, there is a definite relationship between the two, and the Bayes law is the statement of that relationship, as described by the following formula:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

D.9 Content-based Payload Partitioning

The COPP algorithm is based on Rabin fingerprints that searches for repeated byte sequences by partitioning the payload into content blocks. This enables the description of payload content that is tolerant of payload variability to some degree. The algorithm has been developed for the Autograph project [9] and is performed after accumulating enough number of suspicious flows to partition the payloads into variable-length chunks of content blocks.

Appendix E

Signature examples

E.1 Honeycomb signature

The honeycomb signature records can be translated into Bro or Snort format. The strings can be distributed over several packages as the messages are reassembled per flow. Example for the Slammer Worm:

```
alert udp any any -> 192.168.169.2/32 1434 (msg:
"Honeycomb Fri Jul 18 11h46m33 2003 "; content:
"|04 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 DC C9 B0
|B|EB 0E 01 01 01 01 01 01 01|p|AE|B |01|p|AE|B|90
 90 90 90 90 90 90 90|h |DC C9 B0|B|B8 01 01 01 01
|1|C9 B1 18|P|E2 FD|5 |01 01 01 05|P|89 E5|Qh.dllh
el32hkernQhounthickChGetTf|B9|llQh32.dhws2_f |B9|e
tQhsockf|B9|toQhsend|BE 18 10 AE|B|8D|E|D4|P|FF 16
|P|8D|E|E0|P|8D|E|F0|P|FF 16|P|BE 10 10 AE|B|8B 1E
 8B 03|=U |8B EC|Qt|05 BE 1C 10 AE|B|FF 16 FF D0|1
|C9|QQP|81 F1 03 01 04 9B 81 F1 01 01 01 01|Q|8D|E
|CC|P|8B|E|C0|P|FF 16|j|11| j|02|j|02 FF D0|P|8D|E
|C4|P|8B|E|C0|P|FF 16 89 C6 09 DB 81 F3|<a|D9 FF 8
B|E|B4 8D 0C|@|8D 14 88 C1 E2 04 01 C2 C1 E2 08| )
|C2 8D 04 90 01 D8 89|E|B4|j|10 8D|E|B0|P1|C9|Qf|81
F1|x|01|Q|8D|E|03|P|8B|E|AC|P|FF D6 EB|"; )
```

E.2 Earlybird signature

Earlybird signatures are content-based signatures formatted for the Snort IDS. As the content-sifting algorithm does not keep any per-flow state the generated signatures describe only content information contained within a single packet. An example:

```
drop tcp $HOME_NET any -> $EXTERNAL_NET 5000
(msg:"2712067784 Fri May 14 03:51:00 2004";
rev:1; content:"|90 90 90 90 4d 3f e3 77 90
90 90 90 ff 63 64 90 90 90 90 90|";)
```

E.3 COVERS signature

In [37] two signature examples are given. The first example is a size based signature, which is generated for the OpenSSL heap overflow vulnerability:

$$\{type = 2, data.size > 420\}$$

The type and data fields refer to variable names defined in the input format specification for OpenSSL. It means that the system needs to drop an input if one of its field's type is 2, and the field's size is greater than 420. Another example uses character distribution based signature for FTP:

$$\{cmd ='' SITE'', params.size > 452\&\&non - ASCII(params)\}$$

This signature specifies that an attack has size greater than 452 and contains non-ASCII characters.

E.4 Vigilante signature

The following example is an arbitrary execution control SCA for the Slammer worm. The alert is 457-bytes long. The enclosed message is 376-bytes long and has been truncated.

```
Service: Microsoft SQL Server 8.00.194
Alert type: Arbitrary Execution Control
Verification Information: Address offset 97 of message 0
Number messages: 1
Message: 0 to endpoint UDP:1434
Message data: 04,41,41,41,41,42,42,42,42,43,43,43,43,44,44,44,44,
45,45,45,45,46,46,46,46,47,47,47,47,48,48,48,48,49,49,49,49,4A,4A,
4A,4A,4B,4B,4B,4B,4C,4C,4C,4C,4D,4D,4D,4D,4E,4E,4E,4E,4F,4F,4F,4F,
50,50,50,50,51,51,51,51,52,52,52,52,53,53,53,53,54,54,54,54,55,55,
55,55,56,56,56,56,57,57,57,57,58,58,58,58,0A,10,11,61,EB,0E,41,42,
43,44,45,46,01,70,AE,42,01,70,AE,42,......
```

The SCA identifies the vulnerable service as Microsoft SQL Server version 8.00.194 and the alert type as an arbitrary execution control. The verification information specifies that the address of the code to execute should be placed at offest 97 of message 0. The SCA also contains the 376 byte message used by the Slammer worm.

Appendix F

Installation manual for the application state tracker framework

The *application state tracker framework* is a standalone application which tracks the state of various network protocols up to the application layer. In this manual the installation procedure and the configuration of the *Tracker* is described. Because it was written to work in conjunction with *Argos*, also tips for installation and configuration of the latter are provided. The *tracker interface* (actually the identical component as the *Signature Generator*) links the Tracker to Argos and is briefly presented. For a more thorough description of installing the entire set of NoAH components, especially Argos, refer to the document "How to set up the NoAH infrastructure?" which is provided on the NoAH project homepage as part of the deliverable 2.2, http://www.fp6-noah.org/publications/deliverables/D2.2/index.html.

F.1 The tracker framework

F.1.1 Installing the tracker framework

The tracker framework is a standalone application tracking the state of various network protocols up to the application layer. The installation is not yet automated but rather straightforward. The following points have to be done for a proper installation:

1. Make sure you have a version of the *libpcap* [18] installed. The tracker framework has been tested with the version 0.8 of libpcap. This library is used for capturing the raw network packets and is used by various other projects in the network domain, e.g. by Wireshark (former Ethereal). To install libpcap under Debian and Ubuntu Linux Distributions follow these steps:

   (a) Download and install the library: `apt-get install libpcap0.8`

   (b) As the Tracker is looking for the library libpcap.so, we have to create a link with this name to the libpcap version we want to use,
   e.g. `ln /usr/lib/libpcap.so.0.8 /usr/lib/libpcap.so`

   The installation of the libpcap will be quite similar on other Linux distributions.

2. Copy the Tracker files into a folder.

3. If you compile the Framework the first time, type `autoreconf -i`

4. Start the configuration script: `./configure`

5. Run the Makefiles: `make`

F.1.2 Configuration of the tracker framework

The Tracker is configured by a configuration file named `trackerConfig.xml`. Later versions of the Framework will allow configuration files specified by the user. A DTD (Document Type Definition) specification of the XML file will follow. The configuration file first defines some parameters for the tracker framework itself. The number of state-tracking threads for instance allows to improve the utilization of multicore-processors or multiprocessor systems. Then the network protocols available as plugins for the framework are declared. A protocol can have other subprotocols attached. Note that the current version of the Tracker is able to monitor only one host, i.e. packets are only examined if they belong to a connection with the monitored host as source or destination port.

F.2 Qemu, Argos and the Interface

Because the tracker framework is meant to work with Argos, we describe the installation and configuration of *Qemu*, the tracker interface and Argos itself in this section.

F.2.1 Qemu

Qemu [26] is a processor emulator which allows full virtualization of computer systems. In conjunction with the NoAH project Qemu is used to create working image partitions of guest operating systems which will be then able to run with Argos. We won't use Argos for the installation of the guest operating systems because Argos slows down the operating system due to its *dynamic tainted analysis (DTA)*. To install Qemu download the latest archive at the project page and unzip it. Type `./configure`, `make` and `make install`. Qemu is now located in `/usr/local`. Building Qemu may not work with the Gnu compiler *gcc* version 4.xx. Instead use 3.xx. E.g. change the symbolic link `gcc` in `/usr/bin` from `gcc-4.1` to `gcc-3.4`.

KQemu accelerator

It is recommended to additionally use the Qemu accelerator *KQemu*. It can be downloaded at the project page of Qemu. Unzip the archive, run `./configure`, `make` and `make install`. This will install KQemu into `/usr/local`. To run the kernel module `kqemu` the following commands can be used:

1. `/bin/mknod /dev/kqemu c 250 0`
2. `/bin/chmod 666 /dev/kqemu`
3. `/sbin/modprobe kqemu`

The Qemu version has to support the accelerator module. You can check this with the output of the configure script in the directory of Qemu. In the output it should say `kqemu support yes`.

Installing a Guest Operating System

To install a guest operating system from a cd-rom or an image the following steps are necessary:

1. `qemu-img create myOS.img 5GB`
2. `qemu -hda myOS.img -cdrom /dev/cdrom or myImage.iso -boot d`

The first command creates an empty image file with a size of 5 gigabytes. The second command starts Qemu with `myOS.img` as local harddisk. Because we want to mount `myImage.iso` from where we install the guest operating system, we use the `-cdrom` option. The `-boot d` option tells Qemu not to boot from hda but from the cdrom.
The cdrom option can also be used to mount software into the guest OS. If you want to create an image of a directory you can use

```
mkisofs -D -iso-level 3 -joliet-long -l -o directory.img
    -no-iso-translate -allow-multidot -U directory
```

The content of the `directory` is now stored in the image `directory.img`. This image can be mounted with Qemu: `qemu -hda myOS.img -cdrom directory.img`. Now the operating system in `myOS.img` can access `directory.img` as a cd-rom drive.

Providing more memory for the Guest OS

If you want to use more memory for the guest OS use the following commands:

- `umount /dev/shm`
- `mount -t tmpfs -o size=528m none /dev/shm`

The last command allows us to set the memory to 512 megabytes by starting Qemu with `qemu -m 512 ...`. We have set the size to 528 mb because the tmpfs should always be slightly bigger than the memory size used for the guest OS.

F.2.2 Argos

The Tracker is part of a project called NoAH (Network of Affined Honeypots) and thus is meant to work in conjunction with other programs although it can be run independently. The core of the NoAH project is a honeypot system based on the virtual machine Qemu. This system called Argos allows the detection of remote code execution through dynamic tainted analysis (DTA). How to set up Argos and other related topics are described in the documentation section on the project page of Argos [8]. The installation steps in a nutshell: Download the latest Argos archive, unzip and compile it by running `./configure -prefix=/opt/argos`, `make` and `make install`. This will install Argos into the directory `/opt/argos`. If you want to use Argos with the state tracker framework you will have to add the option `-enable-net-tracker` to the configure script. If Argos is too slow and consumes too much memory you can add `-enable-lowmem` as option to the configure script.

F.2.3 Networking

There exist various howto's in the internet for how to set up networking capabilities within Qemu and Argos respectively. A common way is to use a virtual bridge. But with this configuration you won't be able to connect from the guest OS to the outside world except to the host OS. To achieve full NAT-alike networking functionality you will have to combine both bridging and IP forwarding / NAT. In this section we will explain the idea of bridging and virtual interfaces. Further we provide the reader with some simple shell scripts which set up a network environment for Argos and Qemu respectively. The guest OS however will not be able to contact hosts except the host OS. At `www.linuxforen.de` you can find a german howto for Qemu and networking via IP forwarding[1]. A configuration manual for bridged networking with IP forwarding will follow at a later time.

Ethernet bridging A virtual bridge allows you to interconnect network devices like it is done when using real bridges. Additional to the functionality of a pure switch the bridge kernel module gives you the opportunity to filter packets. For further information look at http://linux-net.osdl.org/index.php/Bridge.

Tap / Tun interface *Tun* is a virtual point-to-point network device. It was designed as low level kernel support for IP tunneling. For user applications there are two interfaces: /dev/tunX (character device) and tunX (virtual point-to-point interface). *Tap* is the same virtual device as Tun but instead provides support for Ethernet tunneling. The two interfaces for user applications are: /dev/tapX and tapX.
The idea is that a user application can write a packet (Ethernet or IP frame) to the virtual device /dev/tunX and the kernel will receive the frame from the tunX interface. The other way around the kernel writes to the tunX interface and the user application reads from /dev/tunX. Thus these virtual network devices (Tap / Tun) can receive packets from a user space program instead from a physical network interface. And the other way around

[1] http://www.linuxforen.de/forums/showthread.php?p=939854#post939854

they send packets to user space applications instead of to a physical network interface. The kernel itself handles the packets like they came from real physical devices. A good starting point for further information is http://vtun.sourceforge.net/tun/faq.html.

Combination of bridging and the tap interface Because we want to have access to network devices via Argos and no one should realize that there is a host OS, we first build a bridge on our host OS. This way the emulated OS will access the network via the bridge and will not see the guest OS. But this is only possible if we additionally assign a virtual tap interface to Argos through which it can communicate with the kernel of the host OS.

Preconditions

- Make sure your kernel supports bridging and virtual interfaces (tun). This can be checked by typing `grep CONFIG_TUN .config` and `grep CONFIG_BRIDGE .config` in the directory `/usr/src/<your linux kernel version>`. These two options should be included statically or as modules, e.g. `CONFIG_BRIDGE=m`.
- The following packets have to be installed: bridge-utils, SDL library and headers

Starting Emulation with Network Support

If we have met above preconditions we will be able to start Qemu or Argos with network support. The following scripts are necessary:

qemu-ifup A script which automates the procedure of setting up the bridge and the virtual interface (appendix F.4.1).

qemu-ifdown A script which removes the created bridge and interfaces (appendix F.4.2).

start-argos A script which starts Argos (appendix F.4.3). Starting Qemu works similar.

When we have started the guest OS do not forget to configure the guest OS too. The presented script in F.4.1 explains how it works.

F.2.4 Tracker interface

The tracker interface links the tracker framework to Argos. When Argos detects an attack the Interface is notified via a socket. The Interface sends a signal to the Tracker and the latter dumps its outputs. In the Tracker output the network packets which have caused Argos to produce an alarm are then identified by the Interface.

Installation

The interface needs the *cargos-lib* which can be downloaded at the project homepage of Argos [8]. For the installation run `autoreconf -i`, `./configure` and `make`. In some deliverables the Interface is part of the tracker framework in that it is contained as a subfolder of the framework. Then `autoreconf` will not only create the necessary files for the Tracker but also for the Interface. The makefile of the Tracker may then also invoke the makefile of the Interface.

Database setup

Because the interface uses a MySQL database to store information about the generated signatures, you have to first make sure that you have a version of MySQL server installed on one of your hosts. The access credentials are later added to the configuration file of the interface as mentioned in the next section. If you have a running version of MySQL server use the following commands to set up the signature database:

- `mysql -uuser -ppassword -hyourhost`
- `mysql> CREATE DATABASE signatureDB;`

- `mysql> signatureDB < signature_database.sql;`
- `mysql> exit;`

Note that you have to replace the access credentials for user, password and host according to your settings in the first of the above listed commands.

Configuration

The interface is configured by the file `configI.xml`. In this configuration file we have to set the IP at which we can connect to Argos through the control socket. With our network configuration this IP corresponds to the IP of the bridge. The .netlog file of Argos will be located from where Argos is started. So be aware when setting the path to the netlog file, that Argos will have to be started from the same directory. It is recommended to use a dedicated directory from where to start Argos because all session related files will then be held in a separate place. The output and the dump file of the Tracker will be located in the Tracker directory.

F.3 Putting the Pieces Together

After the successful installations of all software components we can use our scripts to run either Qemu or Argos. Use Qemu for the installation of guest operating systems and for the installation of additional software. When we have our desired guest OS setup, we start Argos for attack detection, preferably not as root. At last we start the Tracker and the Interface.

F.4 Ready-to-use Scripts

F.4.1 Qemu-ifup

Listing F.1: Script for bridge configuration, place it in /etc

```
#!/bin/bash
#place it in the folder /etc/

#################################
# The bridge
#################################

clear

echo ////////////////////////////////
echo // Configuring the bridge...
echo ////////////////////////////////

if ifconfig br0
then
        echo Bridge br0 exists already.
        sudo ifconfig br0 192.168.3.55 # The bridge gets this IP address
else
        echo Bridge br0 does not exist yet, I am creating it now.
        # adds a bridge br0
        sudo brctl addbr br0
        # bridge br0 with 192.168.3.55 is the gateway for the guest OS...
        sudo ifconfig br0 192.168.3.55
fi
echo Bridge-IP-Address: 192.168.3.55

#################################
# The virtual interface
#################################

echo ///////////////////////////////////////////
echo // Configuring the virtual interface $1...
echo ///////////////////////////////////////////

# $1 is the ifname-argument from argos (for us tun1)
# The bridge connects to the guest OS via this IP
# The guest OS itself doesn't know this and just sees eth0...
sudo ifconfig $1 192.168.3.50
sudo brctl addif br0 $1 # adds the interface $1 to the bridge br0
echo The interface $1 gets the IP 192.168.3.50

echo /////////////////////////////////////////////////////////
echo // Configuration at the Guest OS:
echo // Network-Controller IP: 192.168.3.51 - Host IP
echo // Default Gateway IP: 192.168.3.55
echo // Note: The virtual tap interface is not addressed here
echo /////////////////////////////////////////////////////////

exit 0
```

F.4.2 Qemu-ifdown

Listing F.2: Script for removing virtual interfaces, place it in /etc

```
#!/bin/bash
sudo /sbin/ifconfig br0 down
sudo /sbin/ifconfig $1 down
sudo /usr/sbin/brctl delif br0 $1
sudo /usr/sbin/brctl delbr br0
```

F.4.3 Start-argos

Listing F.3: Script that starts Argos with network capability

```
#!/bin/sh

# without the next line, the mouse won't work
export SDL_VIDEO_X11_DGAMOUSE=0

#without the next line, there would be a warning because of the frequency
sudo sh -c 'echo 1024 > /proc/sys/dev/rtc/max-user-freq'

#set access permissions
sudo chmod 666 /dev/net/tun

#without the next line, you have to start argos as root (only on kernel >2.6.??)
sudo tunctl -u user -t tap0

#start argos, snapshot mode
argos -m 256 -hda /home/user/DiplomaThesis/images/win2kEN.img -net nic,vlan=0
-net tap,vlan=0,ifname=tap0,script=/etc/qemu-ifup -win2k -csaddr 192.168.2.55

#shutdown interface
/etc/qemu-ifdown
```

Appendix G

Signature Database

Table signatureHeader	
Field name *Data type*	Description
sigNo *int unsigned, primary, auto-increment*	The primary key which uniquely identifies a signature and its header. Every signature must have a header.
revision *smallint unsigned*	If the signature was improved the revision number is increased.
osType *smallint unsigned*	The numerical type of operating system for this signature.
moduleName *text*	The name of the application that should be protected.
attackType *smallint unsigned*	The type of attack the signature is describing.
protocol *tinytext*	The protocol name according to the corresponding entry in the NetPDL database.
field *tinytext*	The vulnerable protocol field name according to the entry in the NetPDL database.
lastModification *datetime*	The timestamp of the last modification of this signature.

Table G.1: Database table outline for signature headers

Table signatureDetails	
Field name *Data type*	Description
sigNo *int unsigned, primary*	This foreign key points to the signatureHeader table.
transportProto *tinytext*	The textual description of the transport protocol.
srcPort *smallint unsigned*	The application source port.
destPort *smallint unsigned*	The application destination port.
flowDirection *smallint*	Numerical value of the packet direction.
flowState *smallint*	Numerical value of the connection state.
eipString *tinytext*	The overwritten EIP value if any in text form.
rawEipNetworkPacketOffset *int unsigned*	The offset of the EIP from the beginning of the network packet.
rawEipApplicationPayloadOffset *int unsigned*	The offset of the EIP from the beginning of the application payload.
rawApplicationPayloadOffset *int unsigned*	The offset of the application payload in the network packet.
lcsString *text*	A byte string created with the LCS method that can be used to identify the attack in the network packet.
lastModification *datetime*	The timestamp of the last modification of this signature.

Table G.2: Database table outline for detailed signature information

Table packetIdentifier	
Field name *Data type*	Description
sigNo *int unsigned, primary*	This foreign key points to the signatureHeader table.
name *tinytext*	The name of the PDML field declaring the message type according to the NetPDL database.
packetIdentificationType *smallint unsigned*	Numerical value of the identifier type.
string *tinytext*	A string if the message is identified by a string.
specOffset *int unsigned*	The offset of the field specifying the message type.
specWidth *int unsigned*	The width of the message specifying field.
specFieldValue *int*	The value of the field for this message type.

Table G.3: Database table outline for protocol-specific message type identifers

Table variableField	
Field name *Data type*	Description
sigNo *int unsigned, primary*	This foreign key points to the signatureHeader table.
name *tinytext*	The name of the field with variable length that seems to be vulnerable.
offset *int unsigned*	The offset of the field.
criticalLength *int unsigned*	The critical field length triggering e.g. a buffer overflow.
endTokenString *tinytext*	A hexadecimal string if the field is terminated by a token.
nbFieldType *unsigned short*	The field type according to the NetPDL database.
lengthSpecifyingField *tinytext*	The name of the field specifying the length.
lengthSpecifyingFieldOffset *int unsigned*	The field offset.
lengthSpecifyingFieldWidth *int unsigned*	The field width.
lengthMultiplicator *smallint*	A multiplicator for the field value to compute the actual field length.
lengthAddend *smallint*	An addend for the field value.

Table G.4: Database table outline for vulnerable fields of variable length

Table snortSignatures	
Field name *Data type*	Description
sigNo *int unsigned, primary*	This foreign key points to the signatureHeader table.
revision *smallint unsigned*	The revision of this Snort signature.
content *text*	The Snort signature in text form.
lastModification *datetime*	The timestamp of the last modification of this signature.

Table G.5: Database table outline for Snort signatures

Appendix H

TODOs

Old tracker

1. Support offloaded and thus wrong checksums

Signature information extraction

1. Add a signature property for byte frequency distributions (BFD)
2. Support other attack types, especially format strings
3. Improve idea of generic properties for information extraction

Signature Generation

1. Test if packets are encoded
2. Signature for reassembled packets
3. Testing the generic approach with different protocols
4. Stream-based protocols
5. NetBee library constraints and workarounds?
6. Evaluate new Snort signatures

State machine

1. Event data handling in the network protocol state machine framework
2. Adding timeouts for each state to the event processing
3. Implementation of a state tracking action class

List of Abbrevations

ASR .... Address Space Randomization
BFD .... Byte Frequency Distribution
BNF .... Backus Naur Form
CFG .... Control Flow Graph
COPP .... Content-based Payload Partitioning
DFA .... Deterministic Finite-state Automaton
DTA .... Dynamic Taint Analysis
FTP .... File Transfer Protocol
IDS .... Intrusion Detection System
IPS .... Intrusion Prevention System
ISP .... Internet Service Provider
LCSeq .... Longest Common Subsequence
LOC .... Lines Of Code
MB .... Mega bytes
NAT .... Network Address Translation
NetPDL .... Network Protocols Description Language
NIC .... Network Interface Card
NoAH .... Network of Affined Honeypots
NREN .... National Research and Education Network
OSI model .... Open Systems Interconnection (Reference) Model
PDL .... Protocol Declaration Language
PDML .... Packet Declaration Markup Language
PoC .... Proof of Concept
PSML .... Packet Summary Markup Language
RFC .... Request For Comment
SGM .... Signature Generation Mechanism
SOHO .... Small Office, Home Office
SSH .... Secure SHell
STL .... Standard Template Library
TCP .... Transport Control Protocol
UML .... Unified Modeling Language

Index

address space randomization, 27, 63

Bayes law, 24, 75
byte frequency distribution, 26, 73

computer worm, 16, 63
    polymorphic, 24, 66
content-based payload partitioning, 25, 75

dynamic taint analysis, 21, 25, 27, 64

expectation-maximization algorithm, 26, 74

false positives, 16, 64
firewall, 15, 64
flow, 24, 64

hierarchical clustering, 24, 74
honeyd, 20, 24
honeypot
    high-interaction, 20
    low-interaction, 20

intrusion detection system, 16, 65
intrusion prevention system, 16, 65

Java, 42

longest common subsequence, 24, 73
longest common substring, 24, 73

n-gram analysis, 26, 73
NetBee library, 49
NetPDL, 50
NetVM, 50
network address translation, 30, 65
network processor, 50

OSI model, 30, 65

PDML, 50
protocol declaration language, 48
protocol knowledge, 47
protocol-field knowledge, 16
PSML, 50

Rabin fingerprint, 24, 75

self-certifying alerts, 28, 43
signature
    host-based, 16
    network-based, 16
signature generation mechanism, 39
software bug, 66
software exploit, 15, 66
    zero-day, 16

TCP checksum offloading, 36
Telnet protocol, 30, 66
trinity of trouble, 15

virus, 24, 66
vulnerability, 15

Bibliography

[1] *Snort Users Manual*.

[2] *European Network of Affined Honeypots (NoAH): Attack Detection and Signature Generation*, 15 May 2006.

[3] *European Network of Affined Honeypots (NoAH): Containment Environment Design*, 2006.

[4] A. Broder. Some applications of rabin's fingerprinting method. In Alfredo De Santis Renato Capocelli and Ugo Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[5] C. Kreibich and J. Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[6] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic Worm Detection Using Structural Information of Executables. *8th Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[7] D. Dagon, X. Qin, G. Gu, W. Lee, J. B. Grizzard, J. G. Levine, and H. L. Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of RAID'04*, volume 3224, pages 39–58. Springer, 2004.

[8] Argos: An Emulator for Capturing Zero-Day Attacks. http://www.few.vu.nl/argos/.

[9] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Symposium*, pages 271–286, 2004.

[10] Greg Hoglund and gary McGraw. *Exploiting software: how to break code*, volume 3. Addison-Wesley, 2004.

[11] Snort Intrusion and Prevention System. http://www.snort.org/.

[12] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. In *WORM'03: Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 76–82, New York, NY, USA, 2004. ACM Press.

[13] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[14] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *IEEE Security and Privacy Symposium*, 2005.

[15] Jilles Van Gurp and Jan Bosch. On the implementation of Finite State Machines. *3rd Annual IASTED International Conference Software Engineering and Applications*, October 1999.

[16] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2004.

[17] L. chung Lam and T. cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of the 7th International Symposium in Recent Advances in Intrusion Detection (RAID)*, pages 1–20. Springer-Verlag GmbH, September 2004.

[18] The libpcap project. http://sourceforge.net/projects/libpcap/.

[19] Loris Degioanni, Mario Baldi, Diego Buffa, Fulvio Risso, Federico Stirano, and Gianluca Varenni. Network Virtual Machine (NetVM): A New Architecture for Efficient and Portable Packet Processing Applications. In *8th International Conference on Telecommunications, ConTEL*, June 2005.

[20] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP'05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 133–147, New York, NY, USA, 2005. ACM Press.

[21] Marcel Marghitola. Towards exploit signature generation using honeypots ii. Technical report, ETH Zurich, Computer Engineering and Networks Laboratory, 2007.

[22] Mario Baldi and Fulvio Risso. A Framework for Rapid Development and Portable Execution of Packet-Handling Applications. *5th IEEE International Symposium on Signal Processing and Information Technology*, December 2005.

[23] Mario Baldi and Fulvio Risso. Using XML for Efficient and Modular Packet Processing. In *Proceedings of IEEE Globecom*, December 2005.

[24] Nikita Borisov, David J. Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuangxiong Guo. A Generic Application-Level Protocol Analyzer and its Language. In *The 14th Annual Network & Distributed System Security Symposium (NDSS)*, February 2007.

[25] NoAH: European Network of Affined Honeypots. http://www.fp6-noah.org/.

[26] Qemu: An open source processor emulator. http://fabrice.bellard.free.fr/qemu/.

[27] Paul Adamczyk. The Anthology of the Finite State Machine Design Patterns. *Proceedings of the Pattern Languages of Programming conference, PLoP*, 2003.

[28] The Metasploit project. http://www.metasploit.com/.

[29] Michael O. Rabing. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, Harvard University, 1981.

[30] Rashid Waraich. Automated Attack Signature Generation: A Survey. Technical report, ETH Zurich, Computer Engineering and Networks Laboratory, 2005.

[31] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. *OSDI*, pages 45–60, 2004.

[32] SDL: ITU specification and description language. http://www.sdl-forum.org/sdl/index.htm.

[33] Bro Intrusion Detection System. http://www.bro-ids.org/.

[34] V. Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, 14 December 1999.

[35] V. Yegneswaran, J. T. Giffin, and S. J. Paul Barford. An architecture for generating semantic-aware signatures. In *Proceedings of the 14th USENIX Security Symposium*, pages 97–112, August 2005.

[36] Y. Tang and S. Chen. Defending against internet worms: A signature-based approach. In *Proceedings of IEEE INFOCOM'05*, March 2005.

[37] Z. Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, November 2005.