Department of Computer Science

Markus Fors
Christian Grahn

An implementation of a DNS-based malware detection system

Computer Science
C-level thesis (15hp)

Date/Term:	2010-06-11
Supervisor:	Stefan Alfredsson
Examiner:	Martin Blom
Serial Number:	C2010:11

An implementation of a DNS-based malware detection system

Markus Fors
Christian Grahn

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Christian Grahn

Markus Fors

Approved, June 11, 2010

Advisor: Stefan Alfredsson

Examiner: Martin Blom

Abstract

Today's wide usage of the Internet makes malicious software (malware) and botnets a big problem. While anti-virus software is commonplace today, malware is constantly evolving to remain undetected. Passively monitoring DNS traffic on a network can present a platform for detecting malware on multiple computers at a low cost and low complexity. To explore this avenue for detecting malware we decided it was necessary to design an extensible system where the framework was separate from the actual detection methods. We wanted to divide the system into three parts, one for logging, one for handling modules for detection and one for taking action against suspect traffic. The system we implemented in C collects DNS traffic and processes it with modules that are compiled separately and can be plugged in or out during runtime. Two proof of concept modules have been implemented. One based on a blacklist and one based on geolocation of requested servers. The system is complete to the point of being ready for field testing and implementation of more advanced detection modules

We would like to thank our supervisor Stefan Alfredsson for his insights and suggestions. He has been a tremendous help throughout the project. We would also like to thank the Department of Computer Science at Karlstad University for providing development equipment and a work environment.

Contents

List of Figures

List of Tables

1 Introduction

A number of computers today are infected with malicious software acting without the knowledge of their users. This type of software is termed malware. Current malwares employ a variety of techniques to avoid detection by anti-virus systems which is why they are such a widespread problem. One idea to detect malware, besides the common anti-virus programs, is to externally observe communication patterns. The Domain Name System (DNS) is used to translate hostnames like “www.example.com” to an address usable by other protocols. Monitoring DNS traffic is an easy way to get a general idea of what is happening on a computer without requiring access to the computer itself. A DNS-based malware detection system (DMDS), such as the one created in this project, will have the possibility of detecting malware on a network it is connected to. It can not protect computers from being infected but malware can be detected without requiring all connected computers to install any software. It can not detect malware that does not generate DNS traffic or malware who's DNS traffic is not intercepted by the system. Suitable for large corporate networks or even ISPs, a DMDS can potentially reduce network load by removing unwanted traffic as well as reduce malicious traffic (DDoS/spam/other) on the Internet as a whole.

To this end, we have developed a DMDS platform written in C and in this report we will document its design and implementation. The DMDS works like a network sniffer, picking up DNS packets and logging them without disturbing the data flow (see figure 1.1). Our DMDS is built like a framework to make it easier to extend the program in the future. It is split into three distinct parts. A logging part that does the sniffing and logging. A detection part that reads the log and manages detection modules that detect specific patterns. And an action part that acts when a pattern has been found. Two examples of detection modules have been created. One blacklist based, that with help from a list of known malware domains is able to detect DNS queries made for any domains in the list. The list supports regular expressions. The other example module is a GeoIP lookup module. It can detect queries made for servers in specific countries. If a detection has been made a warning will be sent to the action part. The action part can then do what is necessary depending on how severe the warning is. Examples on actions that have been implemented are.

- Mailing a notification to a user or an admin.
- Block and redirect traffic in a firewall.

The remainder of the report is structured as follows: In chapter 2 we will start with a background section, covering the DNS protocol, the basic idea, how the most common queries looks like and how the domain name system is distributed. It will also cover the types of malware the DMDS might be able to detect. Some related work will also be discussed. Chapter 3 contains a detailed description of the DMDS implementation. A testing of the DMDS performance has also been done and a detailed description of the testing setup and testing procedures as well as the results and conclusions will be covered in chapter 4. In the conclusion chapter of the project, we will discuss some problems that we ran across, what we have accomplished and future work.

2 Background

In this chapter we will provide an in-depth look at domain names and the Domain Name System to shed light on what kind of information can be collected and analysed by a DMDS. We will also talk about different types of malware to illustrate what a DMDS is looking for. Finally we have a look at some related work to see what kind of research there is to build an implementation on.

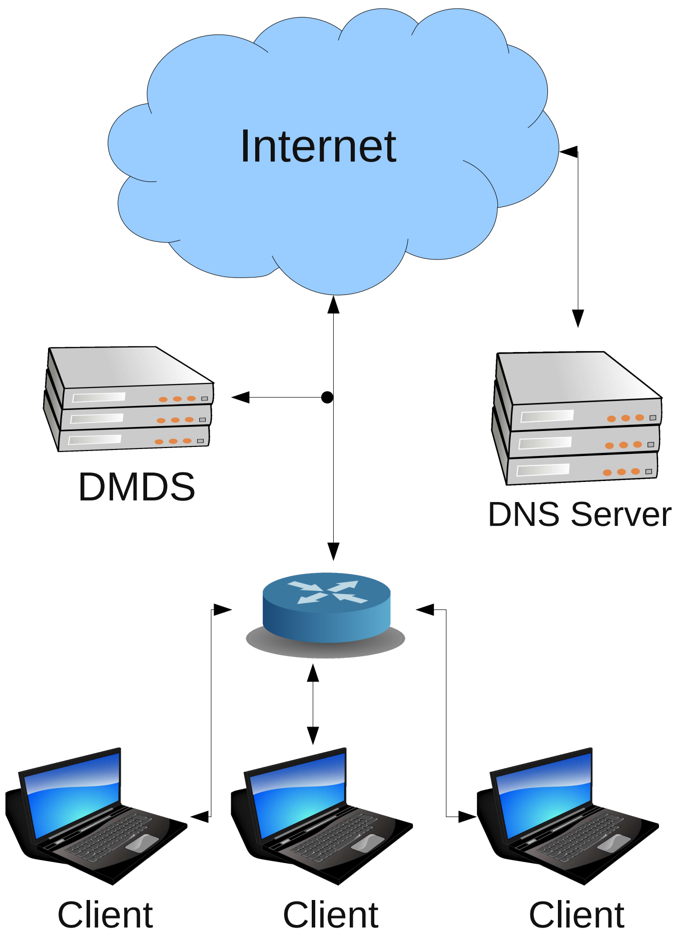


Figure 1.1: Project overview

2.1 DNS

On the Internet the Internet Protocol (IP)[5] is in use which means that connected computers are identified by IP addresses. With IPv4 being the current standard the IP address consists of four bytes of data, normally written with the value of each byte separated by a period, e.g. "192.168.0.1". This is effective and appropriate for computers but is hard to remember for humans. This is why the Domain Name System (DNS) was created. More accurately, it was created to replace older solutions to the problem of hard-to-remember IP addresses. Solutions that did not work with the rapid expansion of the Internet[3].

DNS provides a way to translate textual domain names to IP addresses where the domain names consist of a set of labels separated by periods. In `www.example.com` the labels are `www`, `example` and `com`. A Fully Qualified Domain Name (FQDN) will also always end with an extra '.' to indicate the empty root label, e.g. "`www.example.com.`". Domain names with associated IP addresses are also called hostnames.

When a user wants to contact a server specified by a hostname the domain name is passed to a *resolver*, a piece of software normally built into the operating system, that uses the DNS to translate it to an IP address. This first resolver will send a *recursive* query (further explained in section 2.1.4) to a DNS server that will pass it to its own resolver. This resolver will do the real lookup via *iterative* queries (further explained in section 2.1.3) and get the IP address from the DNS server that is *authoritative* for that domain. The authoritative name server for a domain is the name server that has the *resource records* for that domain in its configuration files and that, through the process of iterative queries, is passed all queries regarding that domain.

2.1.1 Distributed tree structure

The Domain Name System is distributed between many hosts which means that only a relatively small amount of information is available at each node in the network. The namespace can be thought of as a tree structure that starts with the official root servers

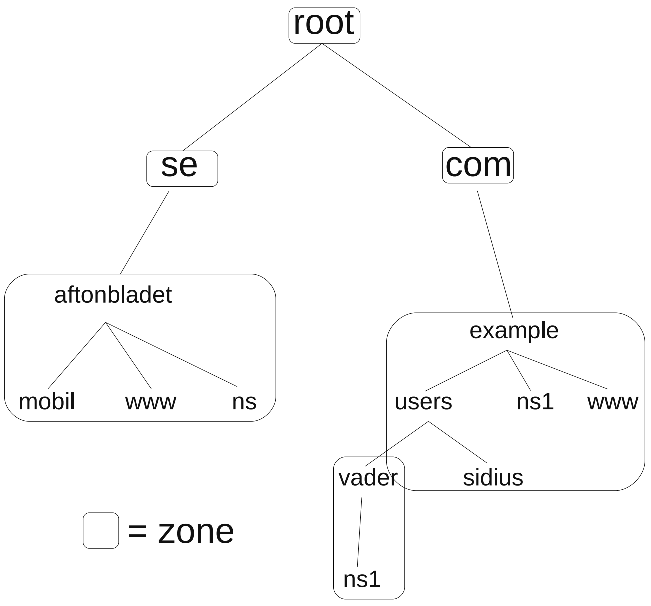


Figure 2.1: Zones

for the Internet and all branches are simply labels which may or may not have other branches. Each branch will have one or more responsible (authoritative) name servers that are registered with the parent name server in this tree structure so that a resolver can be guided step by step from root to leaf, wherever the leaf may be. All branches in this tree below the top level domains (TLDs) can be thought of as leaves because they may have resource records (covered in section 2.1.2) associated with them whether or not they have subdomains beneath them.

A set of resource records under a parent domain that are handled by the same name server is called a zone. A zone has a root label and can end at any branch below the root. In our example (illustrated in figure 2.1) the name server at `ns1.example.com` is responsible for domain names `example.com`, `www.example.com` and `users.example.com` but not for `vader.users.example.com`. It is also responsible for referring queries for `vader.users.example.com`, or its subdomains, to the authoritative nameserver for that zone which in this case is `ns1.vader.users.example.com`.

2.1.2 Resource records

A resource record describes the characteristics of a zone or domain. Each node has a set of resource information, which may be empty. Information associated with a particular domain name is composed of separate resource records. A resource record (RR) can be one of many types. Some of the common types are:

- A-records
- MX-records
- NS-records
- AAAA-records

An A-record contains a IP address associated with a domain name. An MX-record stands for mail exchanger record and contains a domain name, which must have an A-record associated with it, and a priority. The priority dictates the order. An NS-record contains the domain name of an authoritative name server for the domain name in question. An AAAA-record is the IPv6 equivalent of an A-record.

2.1.3 Iterative queries

A complete iterative query for `vader.users.example.com` would start by posing our question to a root server which will refer the resolver to an authoritative server for `com`. The `com` name server will refer the resolver to `ns1.example.com` which in turn will refer to `ns1.vader.users.example.com` that will be able to answer the query. This is the 'iterative' style of resolving a DNS query and it means 4 queries must be sent out and answered before the resolver has the final answer. The delay caused by the multiple queries may be acceptable when establishing a lasting connection, like one made to a FTP server, but when surfing the web many of these delays will have a detrimental effect on the users experience. Further more, if every lookup went through this process it would cause a lot

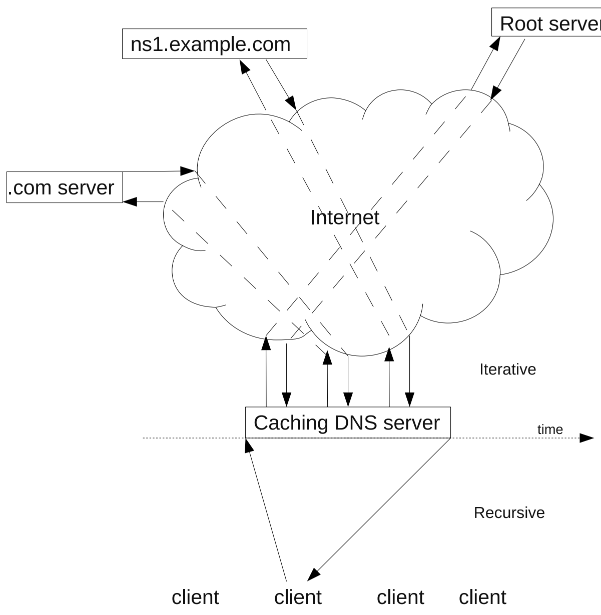


Figure 2.2: A normal DNS lookup

of unnecessary stress to the name servers. This is why every DNS response is marked with a number of seconds that the name server is willing to guarantee that the information will be valid (time to live, TTL). With this information the resolver can safely cache responses if the same DNS record is needed again before the TTL runs out.

2.1.4 Recursive queries

The key to maximising the effectiveness of DNS caching lies in recursive DNS query resolution. When a user indirectly requests a DNS record by trying to access a web-, ftp- or any other type of server via a domain name the local resolver will send a recursive query to its nearest DNS server whose IP address it may have been assigned via the DHCP protocol. This server will accept the incoming recursive query and either send a recursive query of its own to another name server or start its own iterative query to narrow down an authoritative response. When it gets the answer it will return it to the user. This simplifies the process on the client side and it also allows the intermediate name server to cache requests

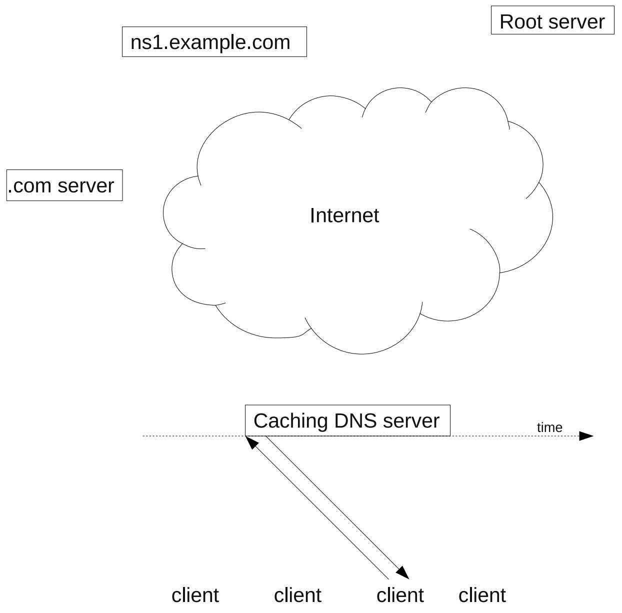


Figure 2.3: A cached response

and responses from a lot of users which makes DNS queries faster as most will not require an iterative resolution.

In figure 2.2 we see a full DNS lookup where the caching server does not have a response cached. This illustrates the separation of the iterative and recursive side of a lookup. In figure 2.3 we see another client asking for the same information and the caching DNS server can respond immediately.

2.1.5 DNS Query/Response

A DNS packet consists of four parts. A MAC-header, IP-header, UDP-header and the DNS section which could be either a query or a response. The MAC and the IP-header is a part of every packet going on Ethernet and Internet, so we will not cover them in this report. Almost all DNS traffic uses the UDP protocol[9] to avoid the overhead of connection setup and breakdown that comes with TCP. The TCP protocol is also used but almost always for zone transfers[7].

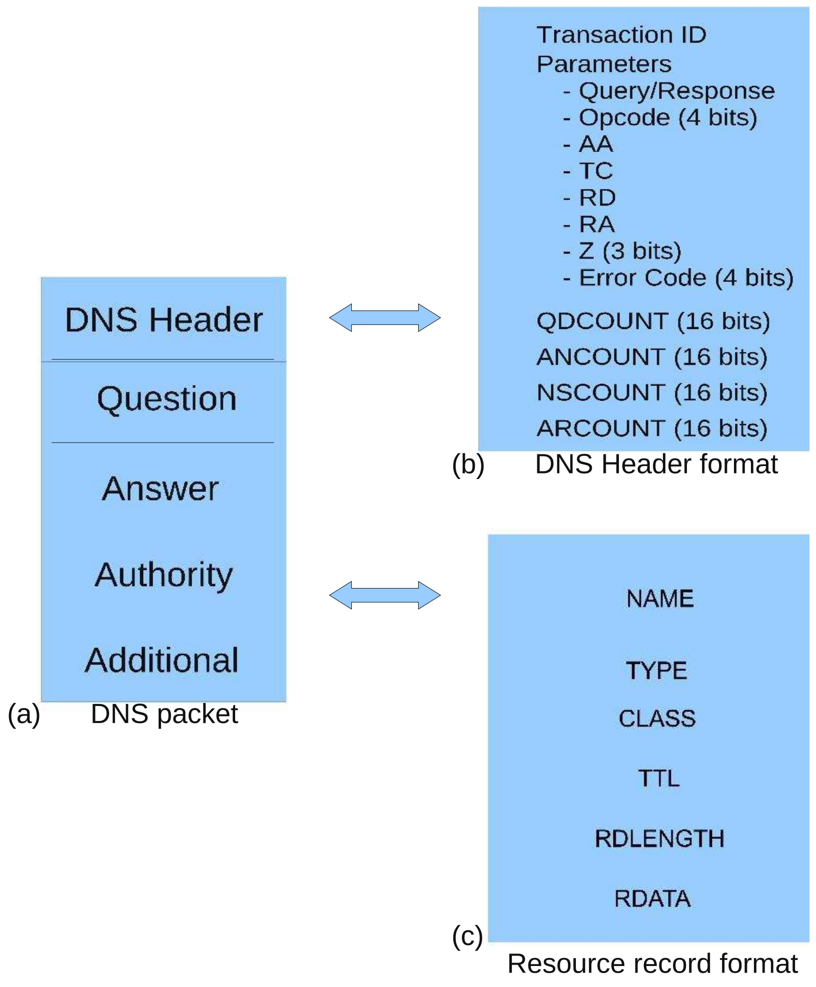


Figure 2.4: The DNS section of a packet with details about the header format and the resource record format.

The DNS section consists of five subparts (see figure 2.4 a). Where the first part, the header, is always present and the presence of the other parts depends on what is specified in the header.

DNS header format The header contains fields that specify the presence of the remaining fields (see figure 2.4 b). The transaction ID is used to match a query with a response, it is unique to the query and the corresponding response. After the transaction ID there is a 2 byte parameter/flag block. It contains eight flags. First in the parameter block is the query/response flag that declares if the packet is a query or a response. Next is a four bit opcode which indicates different type of queries. A zero value signifies a standard query and it is the only value relevant for this report. After the opcode comes the AA flag (authoritative answer), this bit is only valid in responses and declare if the responding name server is an authority for the domain name in the query. Next is the truncated flag (TC), it specifies if the message has been truncated due to the length being greater than what is permitted on the transmission channel. The recursive bit (RD) may be set in a query (if so it is copied into the response). If it is set to 1, it directs the name server to pursue the query recursively. The recursion available bit (RA) is set by the server and tells the client if recursion is available or not. The parameter marked Z consists three bits and is reserved for future use. Last in the parameter block are a four bit error code, set in the response if e.g. the server was unable to interpret the query.

After the parameter/flag block comes four variables that specify different counts. QDCOUNT specifies the number of entries in the question section, usually one. ANCOUNT specifies the number of resource records in the answer section. NSCOUNT specifies the number of name server resource records in the authority records section. ARCOUNT specifies the number of resource records in the additional records section.

Question format The second section in the DNS packet is the question section. A DNS query can contain several questions, the amount is specified in the DNS header. The question section contains the parameters that specify what is being asked for. QNAME contains the domain name that the client asked for e.g. `"foobar.com"`. QTYPE specifies the type of the query. All codes that are valid for a RR TYPE-field are valid in QTYPE. QCLASS specifies the class of the query.

Resource record format The answer, authority, and additional sections all share the same format (see figure 2.4c). The count variables specifies how many there are of each type. NAME is the domain name to which the RR pertains. TYPE contains a code that specifies the meaning of the data in RDATA. CLASS specifies the class of the data in RDATA. TTL specifies how long the RR may be cached before discarded. RDLENGTH specifies the length of the RDATA field. RDATA is a string of octets that describes the resource. The format depends on the information in TYPE and CLASS.

2.2 Malware

The term malware is an abbreviation of malicious software. It refers to any piece of software created for the purpose of infiltrating computers and perform malicious actions without the knowledge of the user or administrator of that computer[12]. The nature of these malicious actions are varying depending on the type of malware. A virus[12] is a type of malware that spreads from file to file within a computer in order to become harder to remove. Despite this relatively narrow definition the term virus is often used to describe all malware. Viruses of the stricter definition will not be covered in this report as they do not generate any DNS (or any other network-) traffic. Unless they are blended with another type of malware in what is called a ”blended threat” type malware and they can therefore not be detected by any DMDS.

2.2.1 Bots and botnets

A botnet[12] is a collection of compromised computers, connected to the Internet, that interacts to do distributed harmful tasks e.g. denial of service, spam or click fraud. The compromised machines are often referred to as zombies or drones and the malicious software running on the machine, without the owners knowledge, is called a bot. The owner of the botnet is called a herder due to his task of guiding the drones.

The herders strongest motivation is usually money. The botnets impact to the Internet grows with the number of drones it has at its disposal and so does the amount of money it can bring in. Therefore the herder wants to infect and gather as many machines as possible to his net. Because number of drones has such importance most bot software contains a spreader that works almost like a worm, searching for vulnerabilities in software and spreading itself through the local network but also to contacts via email and instant messaging clients like Windows Live Messenger. What separates a bot from a worm is that the bot reports to a control system referred to as the command and control (C&C).

The C&C is the interface between the bots and the herder. The herder commands the C&C and the C&C commands the bots. Many botnets are being controlled using Internet Relay Chat (IRC)[8] because of its simplicity, flexibility and also for wider deployment over several servers[12]. An IRC server can easily be installed on a compromised machine and the herder can connect to it using an anonymization service e.g. TOR[10], for better protection. A disadvantage with IRC from a herders point of view is that all chat room traffic is being transmitted in clear text. This makes it easy to eavesdrop on the botnet commands using an Ethernet sniffer. A big number of botnets is also controlled using HTTP but this makes it impossible to control the bots in real time[12]. When using HTTP the bots are set to collect information from the C&C on specific intervals of time. The advantage from the herders point of view is that firewalls often allow traffic over HTTP, while other traffic such as IRC may be blocked.

2.2.2 Spyware/Adware

Spyware and adware is malware designed for financial gain[12]. It does not pose the same threat level as trojans, worms etc. but it makes maintaining privacy on the Internet harder.

A spyware is any software that gathers personal information about you and your actions on the computer, like websites you visit. With this information companies can make e.g. personalised advertising.

An adware is a malware that brings commercial advertising to your desktop. Often in the form of popups. Some adwares also hijack the browser and redirect web pages.

2.3 Related work

Standard botnet behaviour has been documented[17][13] and their activity is monitored[12] to a certain degree. Using DNS monitoring to observe bots and other malicious software is not new. Methods that have already been found to be useful are[11]:

- Manually looking at top 10 queried domains
- Matching known blacklisted domains
- Observing used DNS servers (trying to avoid malware detection techniques at ISP DNS server?)
- Observing what clients have sent the most queries
- Detecting fast-flux domains [16]

All of these methods can be implemented as modules to the DMDS created in this project.

A DMDS can be viewed as a Intrusion Detection System (IDS). A popular IDS is Snort [14]. Like our DMDS implementation (see section 3), Snort uses modules as a part of it's method for detecting specific patterns. We chose to implement our own system instead of implementing Snort modules because we wanted to learn more about building a system from scratch. Our implementation is a more lightweight solution.

3 Fors-Grahn DNS-based Malware Detection System

The software developed in this project is called Fors-Grahn DMDS (FoG-DMDS) and it is designed to be flexible and extensible because it is important that the software can adapt as the malware changes. FoG-DMDS is split into three executables named logging, detection and action to describe their respective purpose. In figure 3.1 you can see an overview on how the different parts in FoG-DMDS are connected. The logging part is passively listening on a connection and picking up DNS packets. The data collected is written to a logfile. The logfile is then read and analysed by detection. The detection part passes relevant data to the detection modules. If any of the detection modules detects something suspicious a warning is sent from the detection part of FoG-DMDS to the action part. Action looks at the warning and decides what action to take depending on how severe the warning is. The different actions taken are split into action modules. In our case the action modules are represented by Linux shell scripts.

This chapter will start by covering structures and code the FoG-DMDS parts have in common and then continue with a more detailed description on the different FoG-DMDS parts, in the following order:

- Logging
- Detection
- Action
- Authentication module
- Detection modules

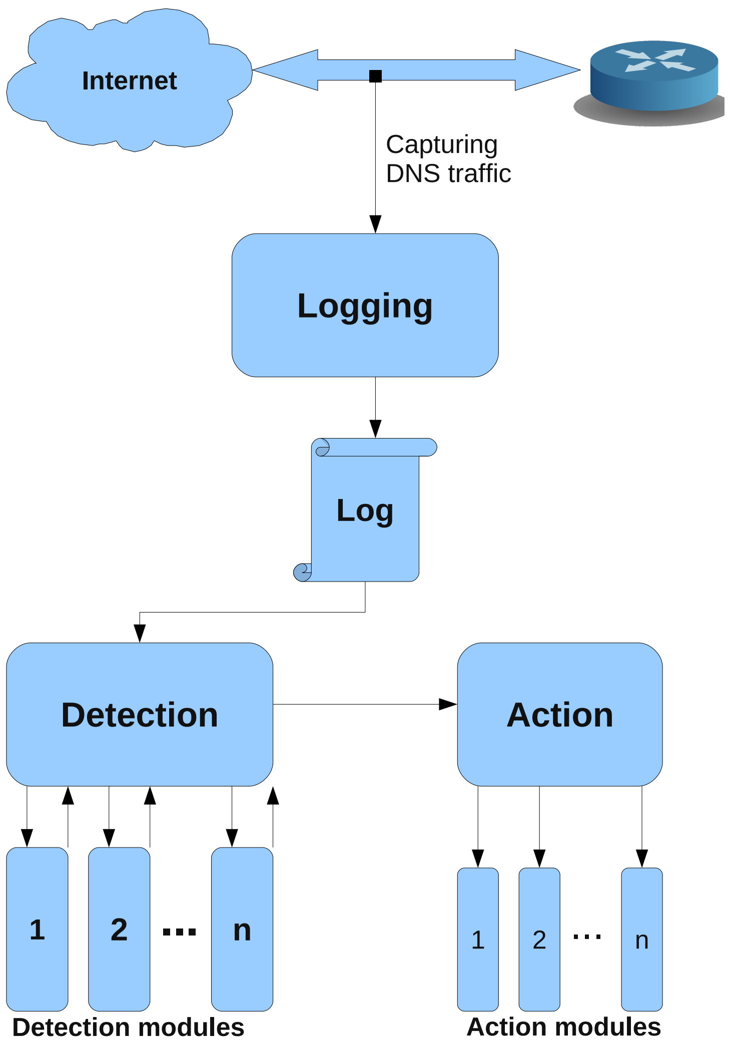


Figure 3.1: FoG-DMDS overview

3.1 Common code

Log messages are stored in a struct appropriately named `logmessage` and it contains two sub-structs to store queries and resource records.

```
typedef struct query *qref;
typedef struct query {
        uint16_t        qnameoffset;
        char            qname[MAX_NAME_LEN];
        uint16_t        qtype;
        uint16_t        qclass;
} query;

typedef struct rr *rrref;
typedef struct rr {
        uint16_t        nameoffset;
        char            name[MAX_NAME_LEN];
        uint8_t         rrtype;
        uint16_t        type;
        uint16_t        class;
        uint32_t        ttl;
        uint16_t        rdlen;
        uint16_t        rdataoffset;
        char*           rdata;
} rr;

typedef struct logmessage *lmref;
struct logmessage {
        uint32_t        sender;                 // ip
        uint32_t        recver;                 // ip
        time_t          in_timestamp;
        time_t          out_timestamp;
        uint16_t        s_flags;                // bitmap, query flags
        uint16_t        r_flags;                // bitmap, response flags
        uint16_t        n_queries;              // number of queries
        qref            queries;                // array
        uint16_t        n_rrs;                  // number of rrs
        rrref           rrs;                    // array
        uint16_t        id;                     // id
};
```

The integers that have names containing `offset` represent the offset of the following string from the start of the DNS message. This is so that references to earlier strings that may be present in the data section of resource records (e.g. NS type records) can be solved.

This struct in binary form along with an integer indicating its total size constitutes the format of the logfile used to pass information between the logging and detection executables.

Similarly the `warning` struct is used to pass information from detection to action, but via a network connection. A total size (in bytes) of the warning is prepended to the below struct before it is sent over the TCP connection between the two programs. The void pointer named `extra` can contain any type of information, only the module and the action program know how to handle it. As an example our blacklist module, described in section 3.6.1, sends the blacklisted address that has been queried for as a string in `extra`. `len` represents the size (in bytes) of the data in `extra` and `strlen` does the same for `printable`.

```c
typedef struct warning* warnref;
struct warning
{
        uint32_t warning;
        uint32_t src;
        uint32_t len;
        uint32_t strlen;
        void* extra;
        char* printable;
};
```

3.2 Logging

This section includes a detailed description on how the logging part of FoG-DMDS works and how it is implemented.

3.2.1 Overview

To log DNS queries the pcap library[4] is used and custom code was written to capture and parse DNS packets on UDP port 53. Firstly the program is split into two threads. One thread is dedicated to picking packets off the network interface using pcap. Pcap will wait for a packet to be captured and will then call a specified function, passing the contents of the packet and some meta information as parameters. From there the packet will be added to a thread safe First-In, First-Out (FIFO) queue. The purpose of the second thread is to check the FIFO queue for entries and handle any available packets. Queries are saved until their corresponding response is captured and then *all* the information pertaining to the transaction is saved in a data object of the type described in section 3.1. Domain names are saved in arrays of 256 characters as their length is restricted to 255 characters including periods[7]. We then write this data object in raw form to a text file and no information is lost.

This log can be read by a detection process while it is being written to or it can be saved, perhaps moved to another computer, and then examined. This log can become quite large on busy networks and should be managed with logrotate in Linux or equivalent.

Aside from storage issues the log should be rotated so that old information can be deleted or anonymised according to any policy, law or other restrictions that might be applicable. When a log is put out of rotation and is no longer written to it should not be deleted immediately however as it may still be in use by a detection program.

3.2.2 Implementation

In our struct all variable length fields (mostly strings) are preceded by an integer value indicating the length of the following field, this is to simplify reading of the log. The log entry is also preceded by an integer value representing its total size (in bytes) so memory can be allocated for the entire struct before it is read. This integer is 4 bytes, big-endian like IP integers so that standard functions like `htonl` and `ntohl`[2] can be used.

3.3 Detection

This section includes a detailed description on how the detection part of FoG-DMDS work and how it is implemented. It will also cover the concept of detection modules.

3.3.1 Overview

The job of the detection program is to read a log and to forward the information to a set of independent modules designed to detect malware in different ways. The modules, or mods, can then return either a normal status or a warning of suspicious behaviour. A warning contains a flag indicating what type of activity is suspected as well as the IP of the offending party and information specific to the warning type. This warning is forwarded from the detection program to its action counterpart via a TCP connection so the action program can be run on any system.

A module is said to be loaded when the library file is found and successfully read by the detection program by using the `dlopen` system call[1].

3.3.2 Implementation

Configuration The detection program will read a configuration file containing full paths to the modules that should be loaded. If the `init()` function for each module return successful status the mod handle is added to a list of active mods. Upon reading a log entry each active mod is called and any warnings raised are forwarded to the action program. When a SIGHUP is caught all modules are unloaded, the configuration file is re-read and the new list of mods are loaded. The configuration file may also contain one line starting with an exclamation point (!) followed by the path to an authentication module to be used when connecting to the action program. If there are several such lines the first one that is successfully loaded will be used and a warning will be printed on the screen indicating the ambiguity in the configuration file.

To illustrate, here is an example of a configuration file:

```
# Lines beginning with # are comments
# Full paths are preferred but not needed
!mods/libauth.so
/home/user/fog/mods/libblacklist.so
../fog/mods/libgeoip.so
```

Detection modules Detection modules are simply libraries with the following functions defined:

```
warnref action(lmref lm);
uint32_t init();
uint32_t unload();
```

The `init()` function is run when a module is loaded and provides a chance for the module to open files and initialise arrays or what ever might be needed. It should return 0 if the module loads correctly and anything else if it fails and the module will be ignored until the configuration is reloaded or the program restarts.

The action function receives a log entry via the `lm` parameter and may choose to return a null pointer indicating normal status or a pointer to a warning struct (`warnref`) to be forwarded to the action program.

The `unload()` function is run when a module is unloaded and gives the module a chance to save its current state to a file so that it can be reloaded in the `init()` function.

Two example modules have been constructed, see sections 3.6.1 and 3.6.2.

Connection When the detection program is started it tries to connect to the action daemon at the host and port specified on the command line. If it cannot connect it will try again with the same host and port when it catches a SIGHUP. Upon successful connection the file descriptor of the socket is passed to a function of the type `uint32_t auth(int socket);` of the authentication module if one has been loaded. If the authentication procedure is successful the auth function shall return 0, all other return values cause the connection to be closed. Because the authentication module and the action program are specific to the end user so is the authentication procedure.

3.4 Action

This section explains how the action part of FoG-DMDS work. It will also include a detailed explanation of the implementation.

3.4.1 Overview

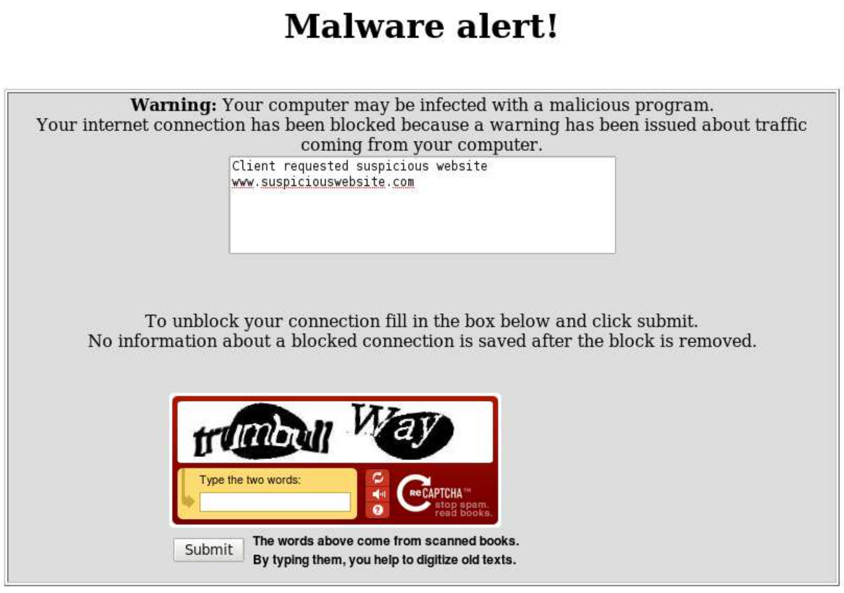
The purpose of the action program is to act on the alerts raised by the detection program. What action should be taken and how it should be accomplished will differ for each FoG-

DMDS user which is why the action program can not be fully implemented beforehand. It will also need to be recompiled to add support for new detection modules. The action program must accept an incoming TCP connection, possibly authenticate the connection and then start receiving alerts. Where it goes from there is implementation specific. The action "server" can be made to handle several clients (detection connections) but running all available log data through the same detection program may be advantageous for modules that need large amounts of data and to prevent malware from avoiding detection by spreading DNS queries over multiple log points (and thus different detection instances) to disguise its patterns. Possible actions include:

- E-mailing the administrator
- E-mailing the offending user
- Calling the offending user
- Blocking Internet access
- Redirecting all or part of the users traffic in order to alert him/her or to attempt to sink-hole the malware

3.4.2 Implementation

In our proof of concept implementation of this system we have included two possible actions; E-mailing the administrator and redirecting part of the users traffic to get his/her attention. Both of these are done with simple shell scripts. Redirection of traffic in our implementation is done with `iptables` and is dependent on all of the users traffic being routed through the computer running the action program. The script stops all traffic from the specified IP except traffic on UDP port 53 (DNS), which is redirected to a known, 'safe' DNS server so that malicious traffic does not slip by on this port, and on TCP port 80 (HTTP) which is redirected to a webserver that alerts the user to its possible infection


Malware alert!

Warning: Your computer may be infected with a malicious program.
Your internet connection has been blocked because a warning has been issued about traffic coming from your computer.

```
Client requested suspicious website
www.suspiciouswebsite.com
```

To unblock your connection fill in the box below and click submit.
No information about a blocked connection is saved after the block is removed.

The words above come from scanned books.
By typing them, you help to digitize old texts.

Figure 3.2: Example of a webpage to inform clients of problems

and could contain a link to a script that will allow the user to access the Internet without restrictions. Any such link should be protected by some form of CAPTCHA[15] so that it cannot be activated automatically by the malware. An example interface for this type of webpage is shown in figure 3.2.

Sending an e-mail to the administrator is a shell script that takes a string as a parameter. This string is the `printable` string sent with every warning (as explained in section 3.1. This string along with the offending IP address is embedded into an e-mail template and sent to the SMTP server running on the local machine addressed to root@localhost. Here is an example e-mail of a warning from the action program:

```
From dmds@localhost  Tue Apr  6 11:07:23 2010
Return-Path: <dmds@localhost>
Date: Tue, 6 Apr 2010 11:07:23 +0200
From: "DMDS-system" <dmds@localhost>
To: DMDS-admin <root@localhost>
Subject: FoG-DMDS: WARNING
Status: R
```

```
Client 192.168.9.101 has triggered a warning of type 1.
"Client tried to look up blacklisted domain www.aftonbladet.se."
```

3.5 Authentication module

An authentication module is necessary to verify that the detection client connecting to the action server is authorised to issue warnings. This is to prevent malicious or accidental connections from tricking the system into taking unwarranted action. Therefore the authentication module is meant to verify the client with the server but not necessarily the server with the client. There is reason to implement a two-way authentication however since a detection client tricked into connecting to a false action server would be ineffective. Furthermore, from a privacy standpoint the warnings transmitted could potentially contain sensitive information.

The authentication module we have created is very basic and from the detection (client) side it will read a file defined in the header file as `AUTHFILE`. The contents of this file is sent over the socket.

On the action (server) side the same module is included and it will receive the first transmission from client and compare it to the contents of the `AUTHFILE`, if the contents match the received text then the authentication is successful.

The security of this authentication method is dependent on the permissions of the file chosen. It should only be readable by users who are trusted to run the detection and action programs.

3.6 Detection modules

In this section we cover the two detection modules that we have implemented. Each subsection will start with a short overview and after that a more detailed information regarding the implementation.

3.6.1 Blacklist module

This section describes the blacklist module implementation.

Overview The blacklist module takes a domain name and tries to match it to an entry in a predefined blacklist. If the domain name was matched the module will return a pointer to a warning struct. If a match was not made the module will return a null pointer.

Implementation In the `init()` function, which is called from the detection, the module will read blacklist.txt containing a list of domain names and regular expressions, one domain per line. It will then loop through the list and compile all the expressions using the `regcomp()` function in `regex.h` and at the same time storing each compiled expression in an array. If the init went well the module will return 0 otherwise 1.

The `action()` tries to match the domain name in the query with the array with compiled expressions using the `regexec()` function in `regex.h` then return a null pointer if there was no match or a pointer to a warning struct if the domain name was in the blacklist.

3.6.2 GeoIP module

This section describes the GeoIP module implementation.

Overview The purpose of the GeoIP module is to render a warning when a client generates a suspicious amount of queries to a country known to have a lot of malware servers for example C&C servers or spam servers. To do this it uses an open source country lookup database provided by MaxMind[6] which has an accuracy of 99.5%. MaxMind also provides an ISP lookup database which is used in a similar way. To generate warnings from suspicious ISPs might also be relevant in this study, but after we had implemented the GeoIP module we had no time left for an ISP version.

Implementation The `init()` function, called from detection, will read from geowarn-list.txt. This file contains a list with country codes for which the module is meant to give a warning. The code is a two to three letter word e.g. SE for Sweden. The `init()` function will then generate a list of geowarning structs. The struct contains the country code, a pointer to a list of client structs and a pointer to the next element in the list. The client struct contains an IP address, a count variable and a pointer to the next element.

The `action()` function takes an `lmref` as a parameter. From the `lmref` it will extract the senders IP address and the domain name. The domain name will be sent as a parameter to the function `getcountry()`. This function will use the Maxmind API to get the country code from the database. The return value will either be the country code or null (if the domain was not in the database). The module will then try to match the country code with the list that was generated in `init()`. If a match is found the module will add the client IP address to the client struct and add one to the count variable. If the IP already is in the client struct list the corresponding count variable will be increased by one. If the count reaches a predefined value the module will send a `warnref` to the action program.

4 Performance testing

In this section we outline a test to show the maximum capacity of our system in terms of the number of DNS queries it can handle per second. The measurements evaluates the logging program performance and a summary of the test results can be found in table 4.1.

4.1 Test design

To find the maximum amount of DNS queries per second that FoG-DMDS can handle we set the system up on a gateway with a client on one side and private DNS-server on the other. The DNS-server was subjected to a constant stream of queries, some for an address blacklisted with our detection module. The flow of queries was steady for 60 seconds

Total queries sent	Queries per second	Hot queries sent	Cold queries sent	Mean packets captured (%)	Mean hot queries intercepted (%)	Mean system exec. time	Mean logging exec. time
6000	100	600	5400	100%	100%	62.00s	60.97s
30000	500	3000	27000	100%	100%	69.25s	60.98s
45000	750	4500	40500	100%	100%	85.09s	61.00s
60000	1000	6000	54000	100%	100%	101.24s	60.94s
90000	1500	9000	81000	99.99%	99.96%	135.32s	61.14s
120000	2000	12000	108000	99.98%	99.84%	169.72s	64.14s
150000	2500	15000	135000	99.94%	99.89%	206.90s	71.75s
180000	3000	18000	162000	99.74%	99.72%	246.70s	86.61s
210000	3500	21000	189000	99.11%	99.04%	290.59s	112.19s
240000	4000	24000	216000	96.39%	95.98%	357.29s	192.41s

Table 4.1: Results of the 60 second performance test

and "hot" queries (those concerning blacklisted addresses) were sent at regular intervals. Evaluation is made by comparing the number of hot queries sent to the number caught and acted upon by the action program. Another, perhaps better, metric is the number of packets intercepted by the logging program compared to the number that passed through the gateway. Because of the unreliable nature of UDP, the number of packets can not be assumed but must be measured at the gateway. For this we use `tcpdump`.

We also want to find the point where the system with all three parts falls behind and is no longer able to keep up in real time.

4.2 Execution

To send the queries a custom program named `perftest` has been created that sends DNS-queries without waiting for the response. It uses only two different packets which are read in binary form from files `white.pkt` and `black.pkt` where `black.pkt` contains a query that should raise a warning in the blacklist module. To these packets a 16 bit transaction identifier is prepended before the query is sent. `perftest` takes as command line arguments

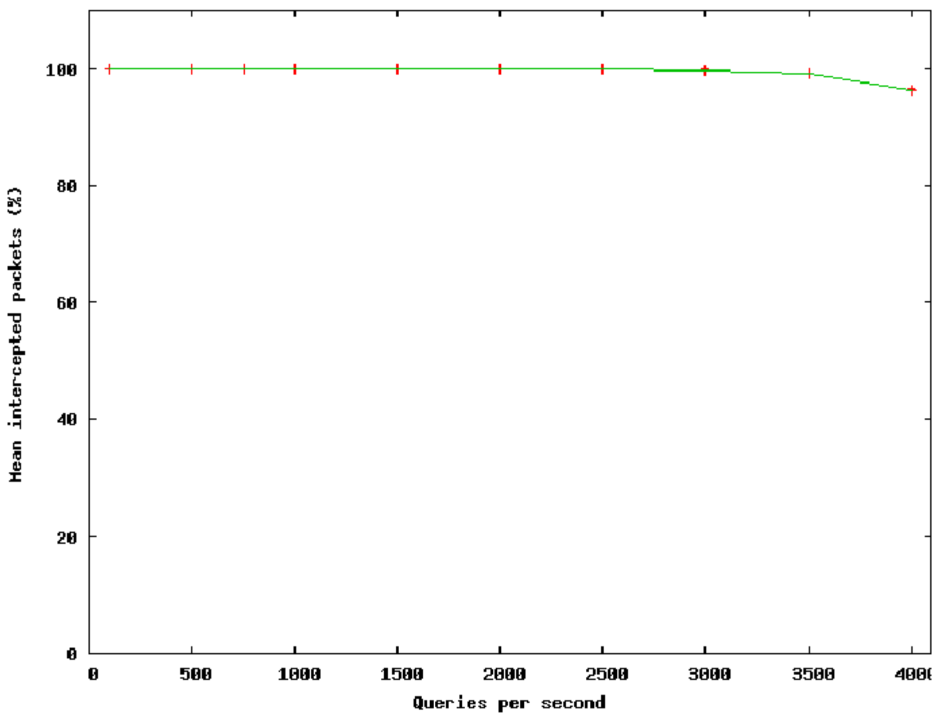


Figure 4.1: Packet loss in percent as a function of queries sent per second.

the IP address of the DNS server, the number of queries to send, how many "cold" queries should be sent between hot queries and finally the delay in microseconds between packets.

The gateway running FoG-DMDS utilizes a Pentium III 1GHz CPU and 256MB of RAM. Both implemented modules are loaded during the tests and the tests are repeated 30 times.

4.3 Results

Based on the results of our tests, shown in table 4.1 we can see that no hot queries are missed as long as all DNS packets both ways are collected. We can also see that packets are first missed by our logging program at 1500 queries per second (qps) and that we can see consistent loss at 2500 qps. The rest of the system however was hardly able to keep

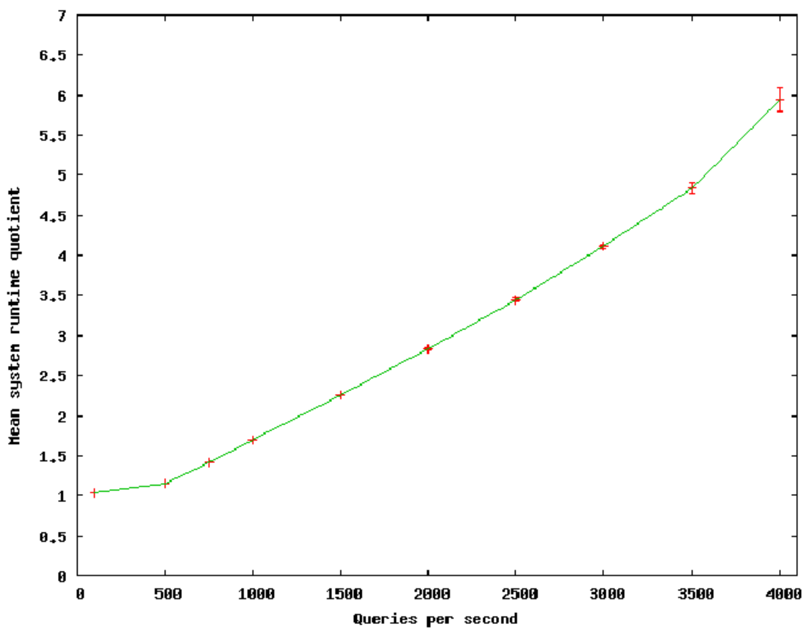


Figure 4.2: Runtime quotient as a function of queries sent per second.

up with the incoming traffic at 500 qps. At 750 qps the last warnings were acted upon around 25 seconds after the flow of queries had stopped. At 2000 qps it took approximately two additional minutes for the system to work through the accumulated queries of the 60 second test. At 2500 qps and more the second thread of the logging program also fell behind causing packets to be buffered in primary memory until they could be dealt with. Between 2500 qps and 4000 qps the additional time required by the logging program to write everything to the logfile went from ten seconds to two full minutes. In figure 4.1 you can see the increasing packet loss towards 4000 qps. Figure 4.2 shows the execution time relative to the set test duration of 60 seconds. The y axis represents the mean execution time of the entire system, from logging to action, divided by the test duration.

Considering the hardware used for this test we can conclude that even on a very modest machine the system can handle near 500 qps. Assuming active users send approximately 10 DNS queries per minute on average it would take 3000 active users to reach 500 qps.

5 Conclusion

The purpose of our work was to contribute to the concept of looking at DNS data in the big battle against malware and computer virus. The goal of our work was to implement a framework for malware detection via DNS traffic monitoring that could be used by organisations of any size. The system was to be easy to expand with new methods for detecting malware and customised actions for when a malware is detected. We have detailed the design of the resulting system in this report.

In this section we explain what we have accomplished in this project. We also discuss some thoughts on future work and ideas for other modules.

5.1 Product

We have developed a program named FoG-DMDS that is split into three parts. The first part can passively pick up DNS packets in a network and write the data to a log. The second part will read the log and can with the help of two example modules detect queries that are deemed suspicious because of the domain in question or the location of the server behind the domain name. Finally the third part that is just a proof of concept implementation can act on warnings raised by the second part. It can use one of two scripts in response to a warning. A mailing script that sends a mail to a user or admin, notifying him or her that FoG-DMDS has come across something suspicious or a redirection script that can block or redirect traffic in a firewall (in our case `iptables`).

5.2 Future work

In this section we will cover some of the things we wanted to do but did not have time for. Like implementing more detection modules and fixing some bottleneck problems to optimise FoG-DMDS performance.

5.2.1 Bottlenecks

After the testing was done we noticed some delay in the detection part of our program. One possible optimisation is switching to a less flexible blacklist module that is not based on regular expressions, but will perform faster matching.

We also noticed that the logging part inexplicably used all available CPU time, even when the network was idle. To make sure that this is not a problem due to slow hardware some further testing with better hardware must be done. Some rewriting of the code to make logging more efficient may also be of importance.

5.2.2 Ideas for other modules

In this section we will list some of the modules we think would be useful and want to integrate with FoG-DMDS in the future.

Spam detection module A module for detecting hosts infected with spam sending malware by looking at the MX-records (see section 2.1.2).

ISP detection module A similar module to the GeoIP module, using a ISP-lookup database and give a warning when a query is made for a server that is held by a suspicious ISP.

Statistic reporting module A module for statistic analysis and reporting that collects data and is able to display it in a few useful ways e.g. tables and graphs.

Statistical detection module A module that looks at the query statistics to detect anomalies in a network e.g. one client generating alot more queries than any other client.

5.2.3 Testing with malware

The system is ready for testing with real malware to see what types can be detected. Aside from testing methods to detect malware it would also be necessary to measure the rate of false positives and to test for the best solution to inhibit the malware without disturbing the user or automated system to the point that it cannot perform its duties.

References

[1] The Open Group. *POSIX:2008 - IEEE Std 1003.1-2008, dlopen function.* Retrieved 2010-06-16. `http://www.opengroup.org/onlinepubs/9699919799/functions/dlopen.html`.

[2] The Open Group. *POSIX:2008 - IEEE Std 1003.1-2008, htonl function.* Retrieved 2010-06-16. `http://www.opengroup.org/onlinepubs/9699919799/functions/htonl.html`.

[3] K. Harrenstien, M. Stahl, and E. Feinler. DoD Internet host table specification, RFC 952. 1985.

[4] Luis Martin-Garcia. Programming with libpcap - sniffing the network from our own application. *Hakin9 Magazine*, volume 2, 2008.

[5] Thomas A. Maufer. *IP Fundamentals.* Prentice Hall, 1999.

[6] MaxMind. *maxmind.com.* Retrieved 2010-05-04.

[7] P. Mockapetris. Domain names - implementation and specification, RFC 1035. 1987.

[8] J. Oikarinen and D. Reed. Internet relay chat protocol, RFC 1459. 1993.

[9] J. Postel. User datagram protocol, RFC 768. 1980.

[10] TOR The Onion Router. *http://www.torproject.org.* Retrieved 2010-05-01. `http://www.torproject.org/`.

[11] Antoine Schonewille and Dirk-Jan van Helmond. The domain name service as an IDS. Technical report, Universiteit van Amsterdam, 2006.

[12] Shadow Server. *shadowserver.org.* Retrieved 2010-05-01. `http://www.shadowserver.org/`.

[13] A. Shahrestani, S. Ramadass, and M. Feily. A survey of botnet and botnet detection. *Publisher: IEEE Computer Society*, 2009.

[14] Snort. *snort.org*. Retrieved 2010-06-16.

[15] Luis von Ahn, Ben Maurer, Colin McMillen, David Abraham, and Manuel Blum. recaptcha: Human-based character recognition via web security measures. *Science*, 321, 2008.

[16] Bojan Zdrnja, Nevil Brownlee, and Duane Wessels. Passive monitoring of DNS anomalies. *CAIDA*, 2006.

[17] Zhaosheng Zhu, Guohan Lu, Yan Chen, Z.J. Fu, P. Roberts, and Keesok Han. Botnet research survey. *Publisher: IEEE Computer Society*, 2008.

A Glossary

API Application Programming Interface. An interface implemented by a software program which enables it to interact with other software.

Bot The malicious software running on the compromised machine.

Botnet A collection of compromised machines used of harmful distributed tasks.

C&C Command and control. The C&C is the interface between the bots and the herder.

Cold query A DNS query that is not “hot”. See hot query.

Data mining To collect large amounts of data for the purpose of drawing some generalization from it.

DDoS Distributed denial of service.

DMDS DNS-based malware detection system. Any software system that attempts to detect malware based on DNS traffic

DNS Query A request for information from a client to a server.

Herder A botnet owner.

Hot query A DNS query concerning a domain name that has been blacklisted by our blacklist module.

ISP Internet service provider.

Malware An abbreviation of malicious software.

MX-record The mail exchanger record.

Namespace A set of identifiers where one identifier can not have two meanings.

Shell script Shell scripts are sets of instructions interpreted by a command line interpreter (a.k.a. ”shell”). Our shell scripts are all written for the Bourne-again shell for Unix (bash) and are in this report also called simply “scripts”.

tcpdump A packet analyser that runs under the command line. It can intercept and display packets being transmitted or received over a network.

Thread safe A piece of code that can be executed simultaneously by several threads without risking data loss or corruption. Also used in this report in reference to an abstract object and its associated functions.