Fighting Botnets in an Internet Service Provider Environment

Morten Knutsen

If you don’t control your mind, someone else will.

— John Allston

Abstract

Botnets are compromised hosts under a common command and control infrastructure. These nets have become very popular because of their potential for various malicious activity. They are frequently used for distributed denial-of-service attacks, spamming, spreading malware and privacy invasion. Manually uncovering and responding to such hosts is difficult and costly.

In this thesis a technique for uncovering and reporting botnet activity in an internet service provider environment is presented and tested. Using a list of known botnet controllers, an ISP can proactivly warn customers of likely compromised hosts while at the same time mitigate future ill-effects by severing communications between the compromised host and the controller.

A prototype system is developed to route traffic destined for controllers to a sinkhole host, then analyse and drop the traffic. After using the system in a live environment at the norwegian reasearch and education network the technique has proven to be a feasable one, and is used in a incident response test-case, warning two big customers of likely compromised hosts. However, there are challenges in tracking down and following up such hosts, especially "roaming" hosts such as laptops.

The scope of the problem is found to be serious, with the expected number of new hosts found to be about 75 per day. Considering that the list used represents only part of the actual controllers active on the internet, the need for an automated incident response seems clear.

Keywords: bots, botnets, network security management, incident response.

Preface

This thesis represents the conclusion of my masters degree at the Norwegian University of Science and Technology, where I have studied for 5 years at the Department of Computer and Information Science.

I would like to thank UNINETT AS who have given me a great opportunity to work with such an exiting subject matter, and everyone there involved in the project for their support and useful insights. Special thanks go to my supervisor there, Vegard Vesterheim and the UNINETT CERT team for coordinating the project.

I would also like to thank my supervisor at the university, Anders Christiansen for taking on this project and providing, as always, lots of interesting views and feedback on my work.

Finally, I would like to thank the people closest to me, my friends, family and loved ones, whose support and kind words kept me going during these 20 weeks.

Trondheim, 16.06.2005

Morten Knutsen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents the problem of botnets in general and botnet agents in an ISP-environment in particular. It presents and realises an approach suitable for use by an ISP for identifying agents and warning customers in a quick and automated fashion.

This chapter presents the motivation for this work, what we hope to accomplish and the approach and limitations that specify our work. The rest of the thesis is divided into 5 chapters in the following manner:

- Chapter 2 presents an overview of the situation, introduces terms and concepts and discusses some related work.
- Chapter 3 expands on the problem, and presents the methodology and approach.
- Chapter 4 details the prototype system, discussing design decisions and implementation.
- Chapter 5 presents the observations and results of testing the prototype system on live traffic data.
- Chapter 6 summerizes the work and draws some conclusions. In addition areas for further work is presented.

1.1 Motivation

In todays service provider environment there is an ever increasing number of threats to be faced [5], and the nature of these threats are getting worse by the day. Where there used to be script-kiddies seeking recognition of their skills, there are now organised criminals seeking economic gain through spamming, identity theft, credit card fraud and extortion to name a few.

A lot of this activity is related to the command and control of huge armies of infected hosts behaving like autonomous (ro)bots, thus the name botnets. In fact, some claim that 70% of all spam originate from such botnets [15, 25]. And while the community have gradually become aware of the threat posed by worms, viruses and denial-of-service attacks, the botnets as such have traditionally not received the same attention, though arguably they are as severe a threat.[1] This threat stems from the sheer potential inherent in a network of thousands of hosts. The packet potential is what make these nets highly attractive to people with malicious intent.

For service providers, the main concerns are possible lawsuits, breach of service level agreements and the manual work of combating denial-of-service attacks.

It is reasonable to assume that at any given moment there are a number of such infected hosts in the network of any ISP acting as bots, and that most customers are unaware of their compromised state. These hosts are, in effect, ticking bombs waiting to go off.

1.2 Purpose

This thesis aims to show that service providers can utilize the knowledge of known botnet controllers to proactivly warn customers of controlled hosts, thus increasing awareness and possibly mitigating future ill-effects. By automating the detection and incident response, the operational security staff can get more work done in less time while making informed choices, resulting in an improved network security management process, and hopefully more satisfied customers.

1.3 Goal

Our goal is twofold. First, we want to learn more about the prevalence and behaviour of botnet agents in a typical large service provider environment. To this end we want to quantify and plot collected data. Secondly, we want to develop a working prototype system able to identify controlled hosts, collate information and take operational action supporting the incident response workflow. This system should:

- Gather, store and analyze packet data relating to the controlled hosts.
- Lookup administrative information related to the host.
- Automatically take operational action, at least by issuing warnings to the appropriate personnel with the customer in question.

[1]We realize of course that the various threats are interconnected; hosts infected by malware may become bots as a result. Similarly, an attacker may use a bot to start spreading new malware.

- Generate reports to support the operational staff.
- Utilize escalation and aggregation techniques to improve the incident response process.

1.4 Approach

Our approach utilizes a list of known controllers compiled through forensic analysis. We route traffic destined for the controllers to a sinkhole host, where we analyze and store the packet data. This provides some immediate mitigation, as the data is dropped meaning the botnet agents can no longer connect to their controller hosts.

The packet data is then used to automatically classify the compromised hosts according to a measure of risk. Aggregated data is used to generate reports and take operational action.

1.5 Limitations

Our approach assumes that the service provider is able to get or compile a list of known controllers. It is reasonable to assume that the properties of such a list has a big influence on the value of a system such as ours. The more controllers on the list, the greater the number of controlled hosts that can be identified. However, the list also needs to be kept current with respect to recent controller activity. Depending on the rate with which controllers change their adressess keeping the list up to date could prove vital. Even though such a list might be hard for some service providers to acquire, the general approach as such is equally applicable to any setting where traffic destined for a number of rouge hosts can be utilized.

The operational action taken is governed in large part by the policies of the service provider. However, even with a rapid response blackholing (disconnecting) the affected hosts, a system such as ours can never hope to contain worms or other rapid malware, for that other approaches are more suitable [49, 22].

Similarly, our approach cannot hope to do more than mitigate distributed denial-of-service attacks, again more specialized approaches exist e.g. [16, 28].

Chapter 2

Overview

This chapter presents the concepts necessary for the proper understanding of the thesis, including definitions of commonly used terms. In addition related work on botnets and routing techniques is explored.

2.1 TCP/IP

The Transmission Control Protocol (TCP) [36] and the Internet Protocol (IP) [35] are two of the most important protocols in the layer based Internet protocol suite. IP is a data-oriented protocol for communication between source and destination hosts. It uses blocks of data referred to as *packets* in an unreliable fashion. These packets may arrive damaged, out of order, duplicated or dropped in transit. TCP is built on top of IP and provides programs on computers on the network to *connect* to each other, and send data over this connection. TCP guarantees that the data will be ordered and intact when it arrives. It also allows distinctions between services through the concept of *port numbers*. The TCP and IP headers are shown in figure 2.1.

2.2 IRC

Internet Relay Chat (IRC) [34, 19] is an Internet protocol developed in 1988 for realtime text-based discussion in virtual chat-rooms called *channels*. It was loosely based on ideas from a similar system called RELAY on Bitnet/EARN. It uses a client/server model with the server being the host of the channels and providing the message delivery, pushing messages to clients. Servers can also interconnect to form large IRC-networks.

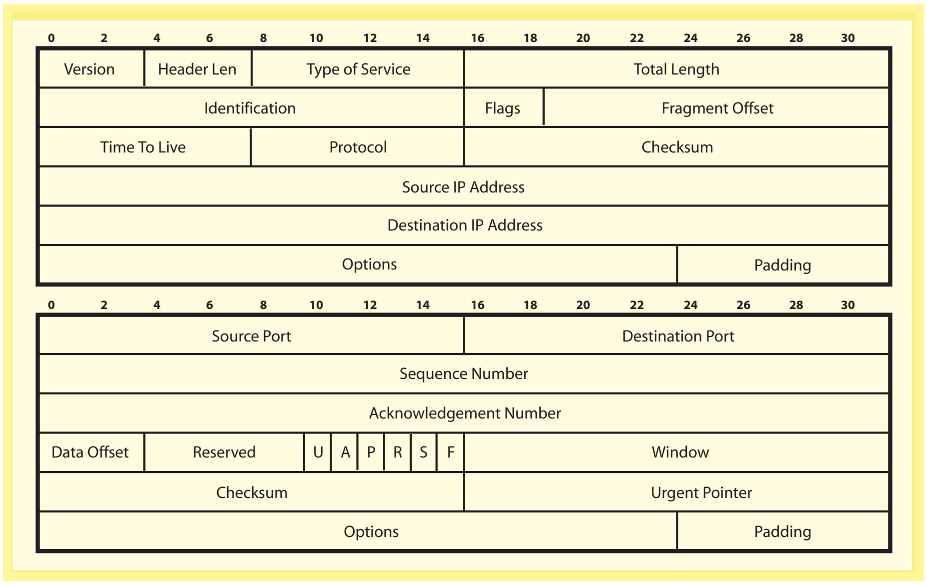


Figure 2.1: TCP/IP headers. **Top:** The IP header (v4). **Bottom:** The TCP header.

The clients are usually interactive software programs like the popular mIRC [20] for Windows. However clients can also exist in the form of automated scripts or sets of scripts, called bots in IRC-terms, from the word robot. Their use is often of an administrative nature, such as access control and logging.

2.3 Border Gateway Protocol

The Border Gateway Protocol (BGP) [38] is a routing protocol for inter-domain routing in the Internet. It maintains a table of IP prefixes that describe reachability between different autonomous systems (AS). An AS is a collection of IP networks under a common control and routing policy, typically an ISP or a very large organisation. Each such AS has a unique AS number (ASN) that identifies the network on the Internet. For every prefix in the BGP routing table there is an associated next-hop address, that can be used to direct traffic to a specific node.

2.4 Dynamic DNS and DHCP

The Domain Name Service (DNS) [30] provides name-to-address lookups on the Internet and is a crucial part of Internet infrastructure at the service level. To map a

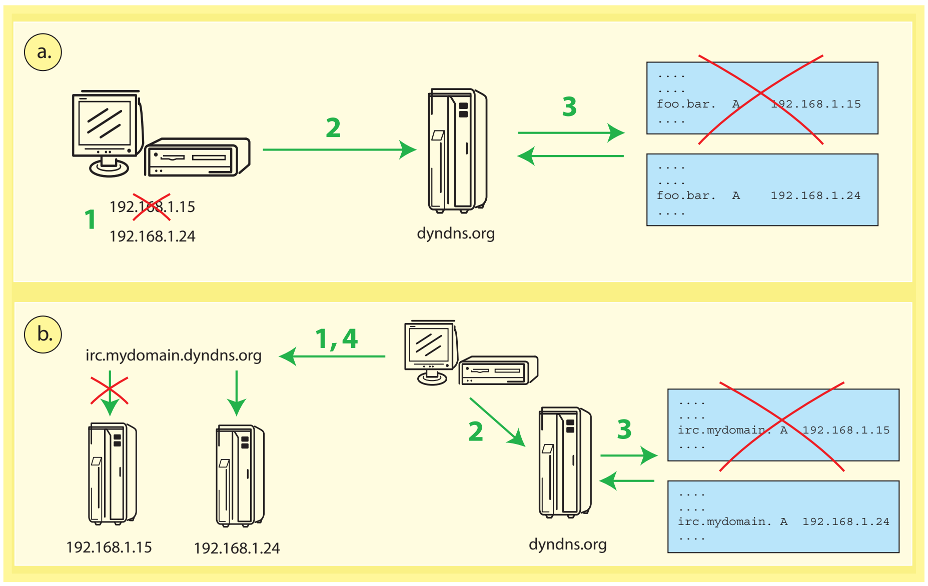


Figure 2.2: **a.** A host with a dynamic IP address changes it's address and notifies the dyndns-service who updates the relevant records. **b.** The hostname irc.mydomain.dyndns.org is to point to 192.168.1.24 instead of 192.168.1.15. A simple notification and the records are updated.

hostname to an IP address the DNS-server(s) for your domain must contain at least one A-record (address record) describing the mapping such as:

```
www.yourdomain.com.        A    192.168.1.10
```

However, for the big number of host on the Internet with dynamic IP addresses, managing hostnames can be somewhat daunting, if you are even allowed to do so by your service provider.

Hosts often obtain their dynamic IP addresses from a DHCP server. DHCP stands for Dynamic Host Configuration Protocol [12] and provides a client host with information required to connect to the network. It can allocate IP addresses in different fashions, one of which is dynamic allocation where each host gets a IP address from a pool, for a given period of time (lease time). When the lease expires, the host might or might not be assigned the same IP address.

To address the needs of people with dynamic hosts who want a static hostname some providers are offering simple DNS-management. Services such as dyndns.org [1] and no-ip.com [47] provide hostname to IP-mappings that are simple to administer, free and anonymous. This enables people to quickly update their record with their new IP address when needed, as shown in figure 2.2.

2.5 Denial of Service

Denial-of-service (DoS) [39] attacks are not new, and exist in a number of forms [29]. In a DoS attack a network entity attempts to deny the legitimate use of some service by causing the target to receive unwanted traffic of some sort. In a traditional *flooding attack* for instance, the attacker floods the target with as many packets as possible with the goal of consuming enough resources to cause the service in question to stop responding or be severely hampered.

Attackers often use the technique of *spoofing* their IP address by forging the header of the IP packet to make it seem as if the packet originated from some other location. Spoofing is also used in a so called *reflective attack* where the attackers spoof the IP address of the *victim* and sends a number of packets to some intermediate hosts, which in turn reply to the target host. These intermediate hosts will assume that their replys got lost in transit and will try to resend the acknowledgement packets, normally 4 times. Thus, such an attack is capable of *packet multiplication*. For each packet sent from an attacker, a multiple is received at the target. This makes it a potent attack.

When several attacking entities combine their efforts to attack the same target entity this is known as a distributed denial-of-service (DDoS) attack. It can wreak havoc on network resources due to the amount of data involved. Figure 2.3 illustrates the concept.

For a more detailed look at the development and trends in DoS technology refer to [17, 16].

2.6 Malware

Malware is a term referring to computer software with malicious intent. This section will briefly explore some of the different types of malware commonly seen on the Internet today.

2.6.1 Worms

The term *worm* was first coined by science fiction writer John Brunner back in his 1975 novel “Shockwave Rider”, but it wasn’t until 1988 that computer worms caused anyone to raise an eyebrow, with the Internet Virus [14].

Worms are software programs using exploits in an operating system or service to rapidly and autonomously propagate across a number of hosts. After infecting one host a worm will employ various scanning techniques to find other potentially vulnerable hosts and attempt to infect them. After infection the worm could deposit

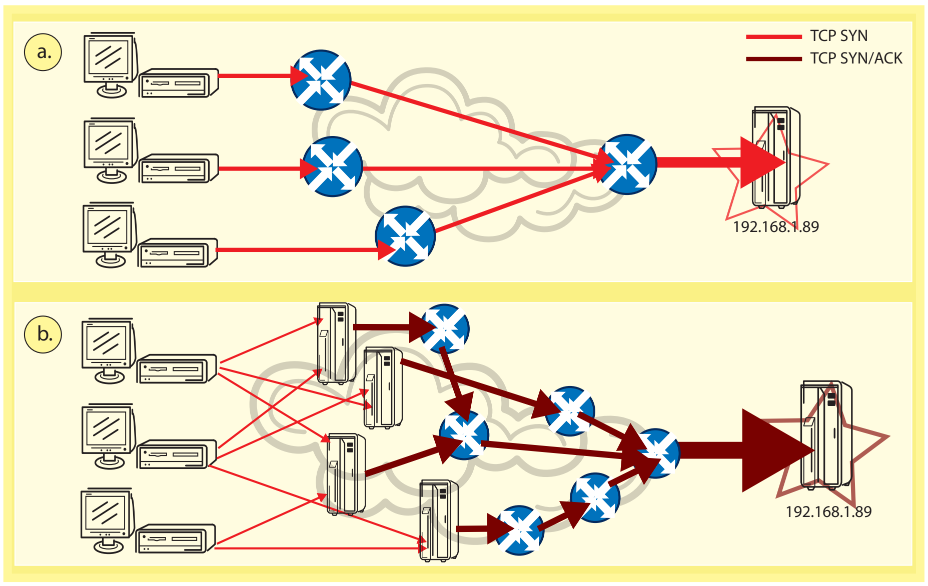


Figure 2.3: **a.** The host 192.168.1.89 is the victim of a traditional direct flooding attack from multiple sources. **b.** This time the host is the victim of a reflective attack and is bombarded by response traffic from well connected sites such as www.amazon.com. Attackers spoof the targets IP address to achieve this.

some sort of payload to suit the attacker such as code to perform DDoS attacks, as was the case with the now famous worm Code Red [13]. The MS Blaster worm of August 2003 is another example of this approach [7].

A lot of early worms suffered from bad code and/or design which hampered their propagation. This was considered by many to be just luck on the part of the community and some warned of the potential for "better", dangerous, more fast-spreading worms [43, 42]. These "flash worms" would use a precompiled list of vulnerable hosts to kick-start the propagation for example. The Saphire/Slammer worm [31] showed that fast-spreading worms were a real threat infecting more than 90% of all vulnerable hosts on the Internet within 10 minutes using a vulnerability in MS SQL Server. Though not very well written and despite having a non-existent payload it still represents a milestone in worm development for its speed.

There is a lot of material available on computer worms. A taxonomy is given in [48] and a survey of recent worms can be found in [21].

2.6.2 Viruses

The distinction between viruses and worms might not be very clear, however generally a *virus* can infect files on offline hosts and usually requires some interaction on the part of the user.

E-mail viruses have been particularly popular in the recent years due to the number of vulnerabilities in the Microsoft Outlook Express mail client. These viruses have a topological advantage when spreading as they often use the address book of the infected client as their new targets, and don't have to do any random scanning as worms traditionally have done.

2.6.3 Trojans

The Trojan horse is well known for its role in the Trojan War, when it allowed the Greeks to capture the city of Troy. In computer terms a *trojan* is a malicious program that disguises itself as some harmless program or better yet, one that promises to rid someones computer of other malware.

Trojans do not replicate as worms and viruses do, but they can be very dangerous and often open *backdoors*, leaving infected hosts accessible and controllable for attackers. Keyloggers and other software designed to snoop on users privacy are also commonly deployed as trojans.

2.6.4 Blended Threats

The distinction between different forms of malware is growing ever thinner as different types and techniques are combined to form what is known as *blended threats*. One such combination of techniques could be having a trojan as the payload of a worm or virus for instance. Such malware represents a very serious threat to internet security as they have a multitude of attack and replication techniques and combine the "best" features of other malware.

In fact, the topic of this thesis can be seen as such a threat. The bot software often combines techniques from other malware to form an all-in-one package to scan, exploit and control hosts on the Internet.

2.7 Botnets

There seem to be different uses of the term *bot* in the current network security literature. Some consider a bot to be a specific, malicious computer program acting as an agent. This is similar to the notion of an IRC-bot as described earlier. However,

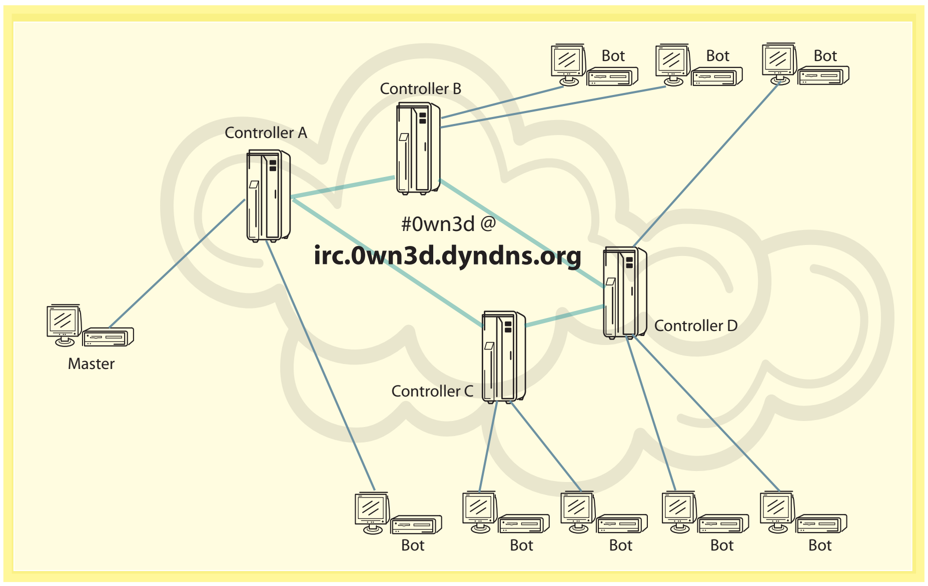


Figure 2.4: A typical botnet, consisting of a private channel on an IRC-network using a dynamic DNS-service. Four servers are linked together to form an IRC-network. The bots all join a specific channel and await further commands from the master attacker.

for the purpose of this thesis we shall use the more general definition: A *bot* is a compromised host under the command and control of an attacker. Thus, a network of such hosts under a common command and control infrastructure becomes a *botnet*. When referring to the actual piece of software making remote control possible, we shall use the term *bot software*. One commonly used infrastructure for botnets is IRC, in which case the bots manifest themselves as actual IRC-bots for the purpose of command and control, and form a botnet in a channel. However, as discussed in section 2.7.2, there are other possible control infrastructures.

Figure 2.4 shows a typical botnet structure. The compromised hosts often join a specific channel on an IRC-server, often hosted using a dynamic DNS-service. The attacker then joins the same channel and communicates with the bots through various textual commands. A more detailed look at the botnet structure is given in later sections, starting with section 2.7.1.

Botnets are not new and have often been mentioned in connection with distributed denial-of-service attacks, as in [17]. For more information on botnets an alternative overview is given in [37]. Some other resources are [23, 45, 27, 26].

2.7.1 Infection and Propagation

There are a myriad of ways bot software can infect a computer system, such as:

- Through an e-mail attachment, possibly by tricking the user into executing malicious software.
- Through a website, possibly by tricking the user into downloading and executing malicious software.
- Through file transfer after some successful exploit, for example by a worm or another bot.

We shall refer to such methods of infection and propagation as the *attack vectors* of the bot software. The first two cases often require some conscious action on the part of the user, a common technique is to trick the user into believing that he is installing some anti-viral or spyware removal software. However, it is quite possible to become infected through browsing a website or previewing a mail, given the "right" client software (e.g. through Active X-controls).

In the last case the bot software is transfered after some known vulnerability in a given program or operating system has been exploited to gain privileged access. This transfer is often accomplished by using Trivial File Transfer Protocol (TFTP), HyperText Transfer Protocol (HTTP) or File Transfer Protocol (FTP).

The malicious bot software is typically variants of software readily available on the internet, customised and configured to the attackers liking with regard to what control channels to use, what tasks the software should be able to perform and so on. Some use a plugin-based architecture to make it easy to customise and extend, for instance with a new propagation module when a new vulnerability appears. Figure 2.5 is a screenshot of a program for the Windows-platform showing the ease with which such configuration can be done.

Once active, the bot software fully installs itself, possibly disguising itself as some sort of important system service and configures the system to launch the bot software automatically on the next system startup. On Windows systems this is usually done by modifying `.ini`-files or through setting registry keys (e.g. `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`).

Bot software often contain modules or plugins to scan address ranges for vulnerable hosts and to exploit common, popular vulnerabilities. As such bots often generate large amounts of "noise" compared to an average workstation, and some botnet controlled host might be found and cleaned up on the basis of such traffic anomalies. Commonly used ports are the well-known 135/TCP and 445/TCP as shown empirically in [46]. These target Windows services with known exploits on the DCE Locator and DS Service, respectively.

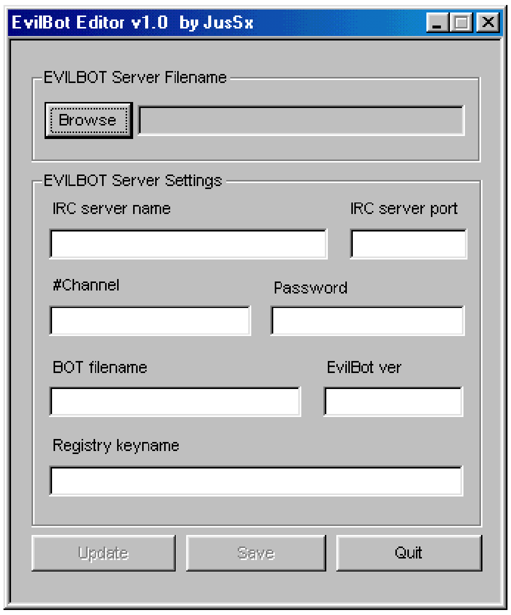


Figure 2.5: A typical, easy-to-use frontend for configuring bot software on the Windows-platform, in this case the Evilbot.a software.

2.7.2 Command and Control

When such bot software is run it will set up a connection to a predefined hostname, to allow it to be remotely controlled. The hostname will often be using some sort of dynamic DNS-service to make it convenient to change the actual IP address behind the hostname if the server needs relocation. The control channel can take different forms, the most common of which is an IRC-based connection. The bot software joins a specific channel, often using a secret key and waits for an attacker to command it. It might also require the attacker to identify himself through some shared secret before allowing such commands. Figure 2.6 shows how such an IRC-based connection might proceed.

The commands are passed to clients through setting the channel topic, or by using commands prefixed with `!` or `.`, such as in figure 2.7. Note how the attacker identifies himself through the `!login`-command, before launching an UDP packet storm.

However, there are other ways to remote control a bot. Some other techniques include:

- Peer-to-peer communication (P2P) gives an obvious benefit in that no centralised server means no single point of failure. This also means mitigation techniques using routing could be somewhat complicated as the notion of

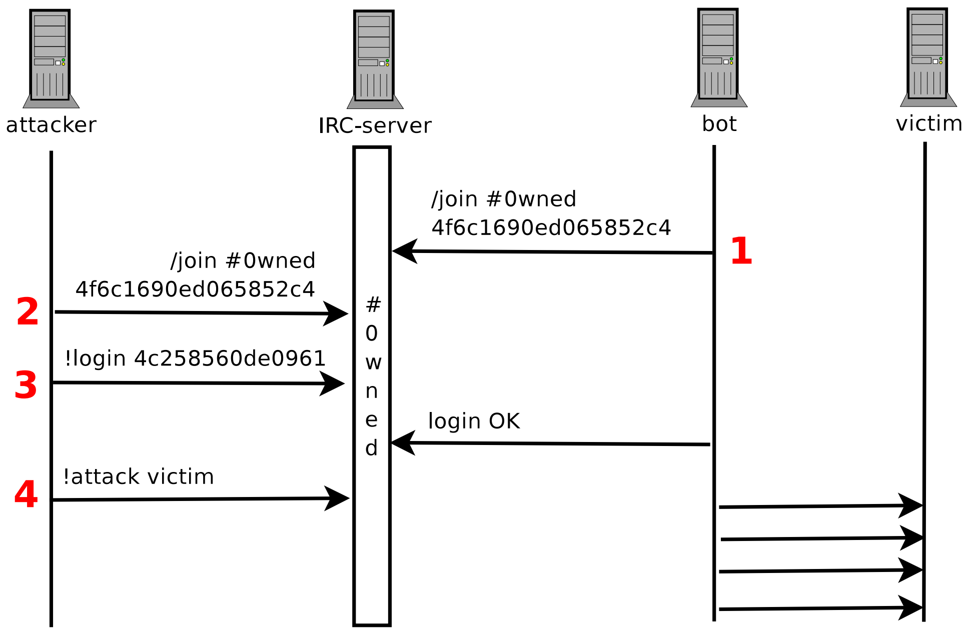


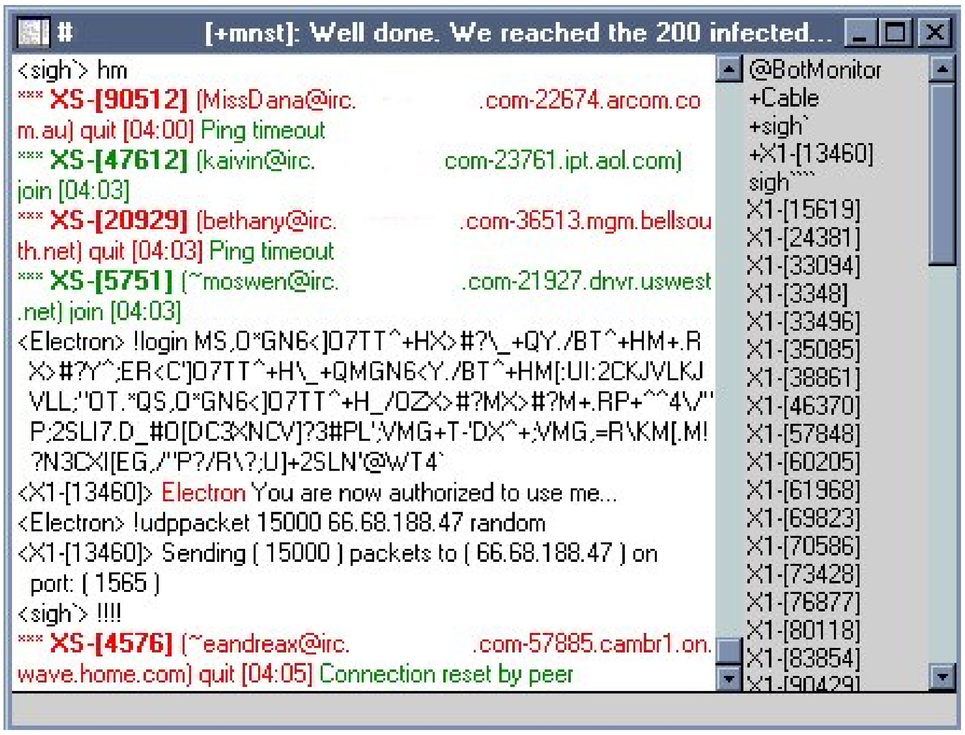
Figure 2.6: 1. Bot joins the channel using secret key. 2. Attacker joins channel using secret key. 3. Attacker optionally identifies himself to bot. 4. Attacker executes commands.

traffic towards a specific controller fails.

- Instant messaging is gaining popularity and might be a viable option for control of botnet agents. Similarly to IRC, an attacker would have to build a profile or group with all the compromised hosts as contacts. Scalability would be a concern.

- Socket-based communication is certainly an option and would give bot software authors the ability to customise their control and command methods. However, it is also complex and time consuming compared to using an existing protocol.

In addition attackers can utilise encrypted communication channels [11] or tunnelled communication to covert their traffic. In fact, a fork of the popular Agobot (and/or Phatbot) software [24] has been controlled through the WASTE chat network. The WASTE network uses a P2P protocol that supports encryption. [4]. The particular fork of the bot software did not implement the encryption routines however.



```
#        [+mnst]: Well done. We reached the 200 infected...
<sigh`> hm
*** XS-[90512] (MissDana@irc.          .com-22674.arcom.co
m.au) quit [04:00] Ping timeout
*** XS-[47612] (kaivin@irc.           com-23761.ipt.aol.com)
join [04:03]
*** XS-[20929] (bethany@irc.          .com-36513.mgm.bellsou
th.net) quit [04:03] Ping timeout
*** XS-[5751] (~moswen@irc.           .com-21927.dnvr.uswest
.net) join [04:03]
<Electron> !login MS,0*GN6<]07TT^+HX>#?\_+QY./BT^+HM+.R
 X>#?Y^;ER<C']07TT^+H\_+QMGN6<Y./BT^+HM[:UI:2CKJVLKJ
 VLL;'0T.*QS,0*GN6<]07TT^+H_/0ZX>#?MX>#?M+.RP+^^4\/"
 P;2SLI7.D_#0[DC3XNCV]?3#PL';VMG+T-'DX^+;VMG,=R\KM[.M!
 ?N3CXI[EG,/"P?/R\?;U]+2SLN'@WT4`
<X1-[13460]> Electron You are now authorized to use me...
<Electron> !udppacket 15000 66.68.188.47 random
<X1-[13460]> Sending ( 15000 ) packets to ( 66.68.188.47 ) on
 port: ( 1565 )
<sigh`> !!!!
*** XS-[4576] (~eandreax@irc.          .com-57885.cambr1.on.
wave.home.com) quit [04:05] Connection reset by peer
```

```
@BotMonitor
+Cable
+sigh`
+X1-[13460]
sigh```
X1-[15619]
X1-[24381]
X1-[33094]
X1-[3348]
X1-[33496]
X1-[35085]
X1-[38861]
X1-[46370]
X1-[57848]
X1-[60205]
X1-[61968]
X1-[69823]
X1-[70586]
X1-[73428]
X1-[76877]
X1-[80118]
X1-[83854]
X1-[90429]
```

Figure 2.7: An actual botnet in a IRC-channel, showing how commands are used [44].

2.7.3 Purpose and Use

Botnets have a myriad of potential uses, which combined with their relatively easy operation makes for their popularity. Some of the more common uses are [45]:

- **Distributed denial-of-service attacks.** Botnets are an ideal platform for launching massive DDoS attacks due to the packet potential inherit in such nets, and because the attacking hosts are topologically diverse making destination-end mitigation difficult without dropping all traffic to the target and thus in effect denying it service.

  The attacks are either used for economic gain through blackmailing, as has been seen with online bookmakers or for take-downs of IRC-networks or specific hosts (either for money or as part of some personal agenda).

- **Spamming.** Bot software often include the possibility to open generic proxies (e.g. SOCKS). These can then be used to send spam and mails luring people to provide sensitive information by pretending to be some trustworthy person or official, so called *phishing*.

- **Sniffing and keylogging.** Bot software is often used to sniff packets sent from the victims machine for sensitive information. Keylogging is an alternative approach that renders the effects of end-to-end encryption useless.
- **Spreading new malware.** Bots are the ideal platform to launch new malware, as a large number of hosts spreading malware simultaneously would kickstart the propagation.

All these potential uses make the botnets popular enough to make some people want to "rent" them, or their services. This is common in eastern Europe for instance, and prices have been mentioned in the range of $20 to $200 for a typical botnet in with a few thousand hosts.

2.8 Service Provider Environment

The work presented in this thesis is being done at Norwegian Internet service provider UNINETT [3]. UNINETT provides universities, university colleges and research institutions with network connectivity and services in addition to handling other national ICT tasks. It is owned by the Norwegian Ministry of Education and Research.

UNINETT has about 200 customers, and the network is used by between 200 000 and 300 000 hosts. They are liberal in their policies, and try to contact local representatives at the customer in question before taking any action to disconnect a host, except in grave cases.

2.9 Related Work

A case-study discussing the general methodology used to discover, track, and stop IRC-controlled trojans is presented in [27].

In [26] more techniques are presented, focusing on NetFlow [9] and trying to locate possible controllers through traffic patterns. It also discusses some incident response techniques and tools, though the focus is mostly on manual techniques. In addition, although identifying infected hosts by examining controller traffic, this is achieved using historical NetFlow data meaning the traffic will still reach the controller. Thus there are no immediate mitigation gains and hosts have to be explicitly quarantined in some way.

A lot of work has been done in the area of sinkholes and blackholes. A very interesting technique is Remote Triggered Blackholes [10] combining BGP with null-routing to create a rapid response tool to efficiently drop traffic based on destination address. This technique is commonly used in a reactive manner to mitigate DDoS attacks and is one of the foundations of our own technique, although arguably we use it in a more proactive way.

Combining this type of blackholing with Unicast Reverse Path Forwarding (uRPF) [8] allows rapid blackholing based on source address as well. A more detailed, hands-on look at the various options for such discard routing including discarding botnet control traffic is given in [40].

While there have been many recent contributions in the areas of DDoS mitigation and worm containment, we are not aware of any work specifically rerouting and using botnet control traffic to identify infected hosts in customer networks. However, the idea of placing sinkhole hosts on botnet command and control addresses is not unique [23], and it is likely other service providers are trying similar methods.

In addition some work on uncovering botnets is presented in a recent whitepaper [45]. Using a honeynet [41] the authors uncover botnet controllers and use special software to observe and learn more about their activity. They tracked botnets through the German Honeynet Project for four months. During this time botnets of up to 50 000 hosts were seen, and in total more than 200 000 unique IP addresses were seen. Their work presents a very useful insight into the operation of such botnets. They are more focused on uncovering new botnets and forensic analysis, and as such their approach complements our own, more incident response focused approach.

Chapter 3

Methodology and Approach

In the previous chapter the various terms and concepts were presented, now it is time to get back to the problem. This chapter restates and expands on the problem and presents the approach taken in this thesis.

3.1 Problem

As seen in chapter 2, botnets are a potent threat to Internet Service Providers and their customers on a number of levels. Gaining the packet potential of a few thousand hosts is relatively easy, and the applications of such an army are many.

For UNINETT the problem is discovering infected hosts acting as part of one or more botnets. Traditionally such hosts has been (perhaps by chance) found to act in a strange manner, provoking further investigation. Hosts might also come to attention through traditional abuse-channels or it might be blatantly obvious if the host is part of a DDoS-attack. In the latter case it might require considerable manual labour to address the problem.

This process is highly reactive in nature, it is not structured and involves a lot of manual labour. Basically it means putting out fires as they appear. As a conservative estimate UNINETT might handle 10–15 botnet-related cases each month. For big DDoS-events the man hours needed grows rapidly.

UNINETT also wants to get some kind of measure on the number of hosts under botnet control in their AS and to better understand the distribution and characteristics of these hosts.

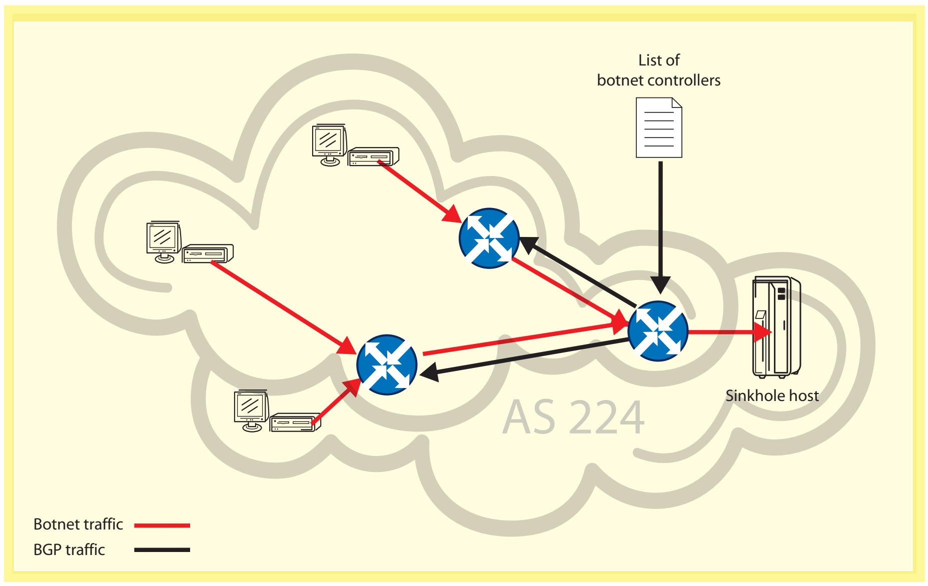


Figure 3.1: Rerouting botnet traffic using BGP.

3.2 Approach

UNINETT has access to a list of known botnet controllers, compiled on the basis of forensic analysis. This list is nothing more than a list of IP addresses of these controller hosts, suitable for dropping traffic destined to these hosts.

By combining this list with a sinkhole host it is possible to collect information on which hosts try to contact addresses on this list, and to store and process data on botnet-related traffic. This is possible by using BGP to inject routes for each of the addresses into our AS. The routes send all traffic to the sinkhole host. Figure 3.1 illustrates the approach.

This approach ensures that updates are handled in a quick, dynamic manner. Once a route changes, these changes are reflected quickly throughout the AS.

Once packets arrive at the sinkhole they are stored and dropped. This ensures some immediate mitigation benefits. Hosts under the command and control of the botnet controllers on the list will no longer be able to participate in spamming or DDoS-attacks for instance. In addition none of the hosts can be “upgraded” to serve as controllers. Storing the packet data also enables UNINETT to gain insight into infection rates and the characteristics of the botnet problem via aggregation and statistics.

The hosts attempting to communicate with the controllers on the list are ranked according to immediate risk. Using a numerical measure, a degree of risk is assigned to each host for a given period of time. Influencing this measure are factors such as activity level and destination port information. This is then used as the basis for the incident response process, warning the customer in question of hosts that are likely compromised. Of course, this approach enables integration with other systems in measuring the risk or in taking more severe action towards a given host.

The approach provides a proactive, automatic means to handle possible botnet infected hosts in a service provider environment, catering to storage, risk assessment and incident response. It is simple, adaptive and provides immediate mitigation benefits.

Chapter 4

The Prototype System

To realise the approach from the previous chapter a prototype system is built. This chapter presents the work done developing and setting up the prototype system, beginning with the techniques used for rerouting the traffic and configuration of the sinkhole. This is followed by the software development work and the methodology of monitoring, logging and incident response.

4.1 Rerouting Botnet Traffic

To gather and store the traffic destined for botnet controllers all known controller addresses are routed to a sinkhole host. BGP is used to inject the routing information into the network, setting the `next-hop` address to that of the sinkhole host and ensuring the routing announcements do not escape the AS by setting the `no-export` property. A typical configuration would be something like:

```
.
.
route-map bc-feed-in permit 10
 description Filter Botnet controller routes
 match ip address prefix-list bc-prefixes
 match community 10
 set ip next-hop z.z.z.z
 set community no-export
.
.
```

The full BGP configuration is given in appendix A.

4.2 Configuring the Sinkhole

The basic setup and configuration of the sinkhole host will be discussed in this section.

4.2.1 Hardware

The host is a typical workstation, given two network interface cards for the purpose of being a sinkhole. The point of this, of course, is to separate the management traffic (interactive sessions, SNMP etc.) from the packets to be logged and discarded.

The host has the following hardware specifications:

- Intel Pentium 4 1.8GHz CPU
- 512 MB RAM
- 80 GB IDE 7200 RPM HDD
- 2 100Mbit Full-Duplex Network Interface Cards (NICs)

4.2.2 Software

The host runs the Debian Sarge GNU/Linux operating system, maintained through the cfengine software. The following other software is used for the prototype system:

- Python, pylibpcap, mx.DateTime, pyPgSQL
- Perl, HTML::Mason, DBI
- tcpdump
- iptables
- PostgreSQL 7.4

4.2.3 Setup

The host was configured with a periodic tcpdump data logging on the sinkhole interface. For this purpose the logrotate software was used to rotate the logs daily, restarting tcpdump.

In addition some basic packet filtering rules were set up using iptables, to ensure no packets would be sent or received on the sinkhole interface if not routed there:

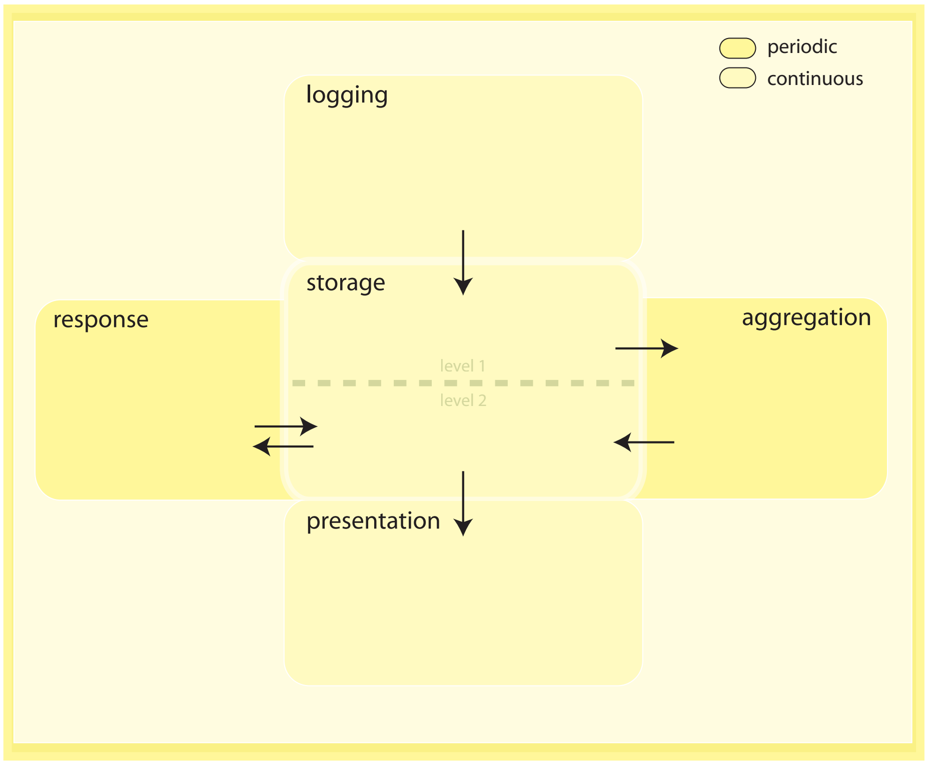


Figure 4.1: The components of the prototype system. Periodic components are meant to be run at regular intervals. The storage system is split into two conceptual levels. Level 1 represents the raw packet data, while level 2 is aggregated data.

```
# Drop and log all incoming packets on eth1
iptables -A INPUT -i eth1 -j LOG_DROP

# Dont let any packets out through this interface
iptables -A OUTPUT -o eth1 -j LOG_DROP
```

The complete logrotate and iptables configuration is given in appendix B.

4.3 System Overview

The main components of the prototype system are shown in figure 4.1. Each component is described briefly below, and discussed more thoroughly in sections 4.4 through 4.8.

- **Storage**. A relational database, PostgreSQL, is used as a two level storage system. The packet data forms the base level, while aggregated datastructures form the more high-level storage.
- **Logging**. Packets routed to the sinkhole host are captured using libpcap [32] and inserted into the database.
- **Aggregation**. This component is run periodically and generates statistics and aggregated data for other components to use.
- **Response**. The response component uses aggregated data to determine the most likely possible compromised hosts and deliver some sort of automated response.
- **Presentation**. Statistics and reports are available through the dynamic web modules in this component.

4.4 The Storage Component

The storage system is shown in figure 4.2. It can be viewed as a two-layer system, with the raw packet data as the bottom layer, and various aggregated tables as the top layer. The rationale behind such an approach is query speed. The amount of data to be stored, and (to a lesser extent) the relatively weak I/O-performance of the sinkhole makes such an approach necessary. Even with a modest 5 packets per second (pps), that still amounts to 432 000 packets every 24 hours, and thus 432 000 new rows in the packets table each day.

Such an approach dictates to a certain extent the use and design of the database:

- Never query the packet table directly unless absolutely necessary. Commonly used data should be aggregated to separate tables.
- Never use foreign keys leading to JOINs on the packet table; replicate data instead.

A more thorough discussion of the database design on a per table basis is given next.

4.4.1 Packet Layer

These are the tables containing the “raw” and unaggregated data.

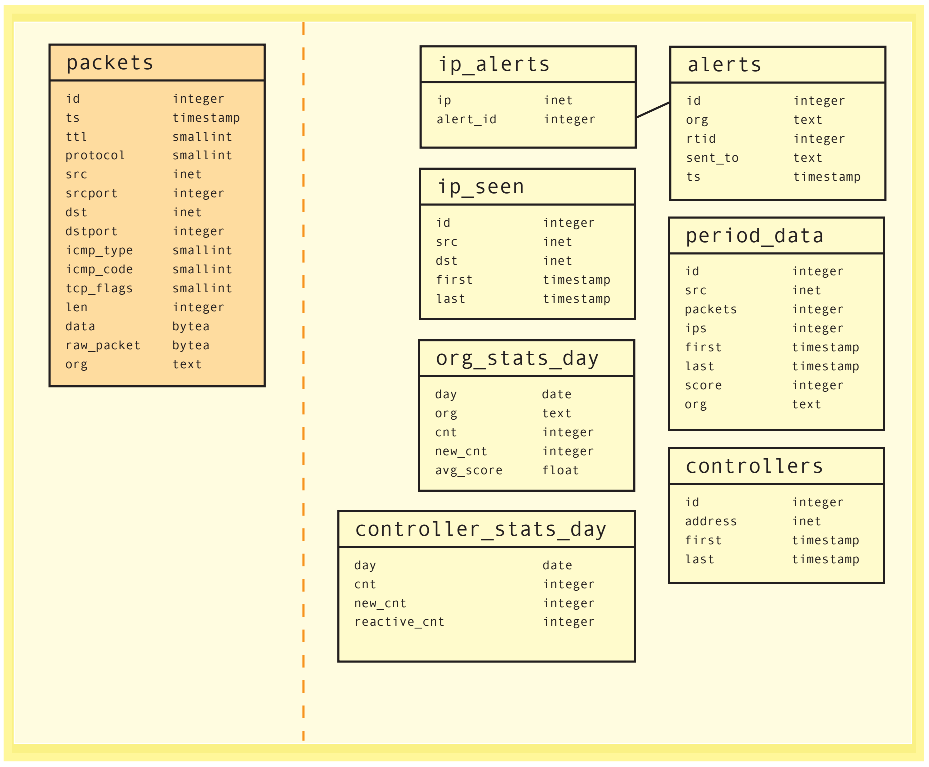


Figure 4.2: The storage system, with the packets table in the bottom layer.

packets

The `packets` table is by far the biggest and most important table in the storage system. It is designed to contain one row per packet captured. A description of the columns is given in table 4.1. The inclusion of the raw, unmodified packet with the `raw_packet` column is costly in terms of storage, but might prove beneficial in a forensic setting.

To speed up queries against the `packets` table several indices are maintained. The table has an index on each of the following columns:

- `id`, primary key
- `dst`
- `dstport`
- `src`

`id`	Autoincremented primary key.
`ts`	Timestamp in UTC.
`ttl`	The Time-to-live value from the IP header
`protocol`	The IP protocol number (e.g. 6 for TCP)
`src`	The IP source address
`srcport`	The UDP or TCP source port number
`dst`	The IP destination address
`dstport`	The UDP or TCP destination port number
`icmp_type`	The type value from the ICMP header
`icmp_code`	The code value from the ICMP header
`tcp_flags`	The integer representation of the TCP bitflags
`len`	The length of the packet payload
`data`	The binary representation of the packet payload
`raw_packet`	The raw packet as seen by libpcap
`org`	The primary domain name of the organisation using the source IP address.

Table 4.1: The data stored in the packets table.

- `srcport`
- `org`
- `protocol`
- `tcp_flags`
- `ts`

Unfortunately heavy indexing means reduced performance when inserting rows into the table; as the indices are updated there is a lot of extra disk I/O.

4.4.2 Top Layer

ip_seen

The `ip_seen` table serves as a record of all IP addresses (both source and destination) that have been observed by the system at some point. This provides a helpful speedup to components needing to extract information about newly seen addresses for instance. The columns are described in table 4.2.

period_data

This table is the main source of aggregated data. It is used by a lot of components in the prototype system. The table design facilitates a periodic aggregation, and

`id`	Autoincremented primary key.
`src`	The IP source address
`dst`	The IP destination address
`first`	The timestamp when this address was first seen by the system.
`last`	The timestamp when this address was last seen by the system.

Table 4.2: The data stored in the ip_seen table.

stores the most interesting data for that period per source address and organisation. The columns are described in table 4.3.

`id`	Autoincremented primary key.
`src`	The IP source address
`first`	The timestamp of the first packet seen from the IP address
`last`	The timestamp of the last packet seen from the IP address
`packets`	Number of packets seen from the source IP address during the period.
`ips`	Number of unique destinations contacted by the source IP address during the period.
`score`	A calculated measure of the immediate risk the source IP address poses.
`org`	The primary domain name of the organisation using the source IP address.

Table 4.3: The data stored in the period_data table.

org_stats_day

This table stores aggregated data used for presenting the number of unique source IP addresses seen for a given day. The figures are for a given organisation. The columns are described in table 4.4.

`day`	A date, part of primary key.
`org`	The organisation, part of primary key
`cnt`	Number of unique source IP addresses seen this day.
`new_cnt`	Number of new, unique source IP addresses seen this day.
`avg_score`	The average score of all unique source IP addresses seen this day.

Table 4.4: The data stored in the org_stats_day table.

This table is used by the presentation component to give a snapshot-picture of the current situation and trends at a given organisation.

alerts

The `alerts` table stores information about an alert generated by the system. The columns are described in table 4.5.

`id`	Autoincremented primary key.
`org`	The organisation the alert applies to.
`sent_to`	In the prototype system, the email address who received the alert.
`ts`	Timestamp when the alert was generated.
`rtid`	An identifier connecting the alert to some external trouble ticket system.

Table 4.5: The data stored in the alerts table.

ip_alerts

This table maps source IP addresses to generated alerts, in order to know which IP addresses were part of any given alert. It contains two columns: `ip` and `alert_id` where the last references an entry in the `alert` table through a foreign key relationship.

controllers

This table serves as a historical record of all controller addresses that have ever been in the system. In order to track addresses that reappear a new row is created if a newly seen address has not been seen in the past 24 hours. The columns are described in table 4.6.

`id`	Autoincremented primary key.
`address`	The IP address of the botnet controller.
`first`	Timestamp when the IP address was first seen by the system.
`last`	Timestamp when the IP address was last seen by the system.

Table 4.6: The data stored in the controllers table.

controller_stats_day

This is a day-by-day aggregation of data found in the `controllers` table. This data is used to get a better understanding of the dynamics of the botnet controllers. The columns are described in table 4.7.

`day`	The date of this data.
`cnt`	The number of unique controller IP addresses seen during the day.
`new_cnt`	The number of new, unique controller IP addresses seen during the day.
`reactive_cnt`	The number of new controller IP address who have previously been active (but not in the previous 24 hours).

Table 4.7: The data stored in the controllers_stats_day table.

4.5 The Logging Component

The logging component will be discussed in this section; from the design decisions to the implementation details. The logging component is designed to be as simple as possible, focusing on capturing packets and committing them to stable storage. It has three primary functions:

1. Capture packets routed to the sinkhole.
2. Extract wanted data from the packets.
3. Insert the extracted data and raw packet into the storage system.

These steps form a sequential path that all packets follow upon entering the system.

4.5.1 Design

For packet capture the libpcap library was chosen, a popular choice in a number of network tools. Basing the packet capture on libpcap gives a couple of immediate advantages:

- A proven, robust interface to packet capture.
- Flexibility to use any input data in pcap-format, such as that generated by the tcpdump software.

Of course, capture performance with commodity hardware and a userspace library like libpcap is not suitable for full scale monitoring at speeds of 100Mbit and upward. But for the purpose of the prototype system and the relatively small traffic volume expected it certainly should suffice. Besides, the limiting factor is expected to be the work done for each packet, particularly the database insertion.

Once packets are in the system they are processed to extract suitable data for storage. The question of what constitutes suitable data is perhaps not straightforward, and to ensure that no data was thrown away, the decision was made to store the entire raw packet in addition to these (extracted) data:

- Timestamp. To be able to provide customers with precise information regarding the time of incidents it is essential to store the time the packet was seen. To ensure consistent and universal information, the time is stored in Universal Time Code (UTC) format.
- Time-to-live (TTL). The TTL information from the IP header can often serve as an indication of the Operating System used to create the packet. With todays TCP/IP implementations the starting TTL-values differ between OS'es to such an extent that even when the packet arrives it is usually possible to guess from which operating system it originated.

- Protocol. The IP protocol number is stored to identify the type of IP packet (e.g. TCP, UDP, ICMP).
- Source Address. The source address allows the system to find the organisation using the address. It is also the most unique identifier of a host that our system can provide (without using other external systems to help track hosts, see chapter 6).
- Source Port (TCP and UDP).
- Destination Address. The destination address identifies the particular controller with which the source host has communicated.
- Destination Port (TCP and UDP). The destination port might serve as an indication of the type of service the source host tried to contact at the destination; however any service can of course be hosted at any port.
- ICMP Type. The type of the ICMP message.
- ICMP Code. The code of the ICMP message.
- TCP Flags. The bitflags from the TCP header can provide useful state information with regards to the TCP connection. Of course, the prototype system will not allow any connections to be established so the data might be of limited use. However, it will make it possible to find those source hosts actively trying to establish a TCP session with a TCP SYN packet.
- Payload length.
- Payload.
- Organisation. Source IP addresses are mapped to organisations, and the primary domain name of the organisation is stored. This is used to group data, and of course to alert and inform the organisation in question.

When storing the data, the system will update the storage system for each packet, not caching any data. This will impact the throughput of the system as the storage system updates indices for each insert as explained in section 4.4.1. However, this ensures that the data we process are committed, and for the sake of the prototype this method of storing packets is considered adequate.

4.5.2 Implementation

The component is implemented using the Python programming language, a modular, high-level, interpreted language especially well suited for rapid prototyping. Packet capture is provided by the pylibpcap library [2], a Python interface to the libpcap library written in C. The database functions are provided by the pyPgSQL module. This module provides an object oriented DB 2.0-API compliant interface to the PostgreSQL DBMS from Python.

The pylibcap `pcapObject` provides the methods used for capture, and is used to set up a loop with a callback function:

```
import pcap

p = pcap.pcapObject()

# open device or file

try:
  p.loop(-1, capture_func)
except:
  ...
```

The capture function has the following signature:

```
capture_func(packet_length, data, timestamp)
```

The data contained in the IP, TCP, UDP and ICMP-headers is extracted using direct access into the datastructure given to the capture function. The following excerpt shows how the IP-level information is extracted from the header:

```
     if data[12:14] == '\\x08\\x00':
       timestamp = mx.DateTime.gmtime(timestamp)
       ip_data = data[14:]
       ip_hlen = ord(ip_data[0]) & 0x0f

       ttl = ord(ip_data[8])
       proto = ord(ip_data[9])
       src = pcap.ntoa(struct.unpack('i',ip_data[12:16])[0])
       dst = pcap.ntoa(struct.unpack('i',ip_data[16:20])[0])

       ip = ipreg.lookup(src)
```

Here `data` represents the entire packet as captured by libpcap, while `ip_data` is the IP packet. The final `ipreg.lookup` call gathers information on the organisation using the source IP address.

Next, the type of IP packet is switched, the appropriate information extracted and inserted into the database. For the TCP-packets this procedure looks like this:

```
       # TCP
       elif proto == socket.IPPROTO_TCP:
         try:
           tcp_data = ip_data[4*ip_hlen:]
```

```
    srcport = socket.ntohs(struct.unpack('H', tcp_data[0:2])[0])
    dstport = socket.ntohs(struct.unpack('H', tcp_data[2:4])[0])
    tcp_hdr_len = (ord(tcp_data[12]) & 0xf0) >> 4
    tcp_flags = ord(tcp_data[13]) & 0x3f

    cu.execute("""
    insert into packets (ts, ttl, protocol, src, dst,
    srcport, dstport, tcp_flags, data, raw_packet, org) values
    (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
    """,
               (timestamp, ttl, proto, src, dst, srcport, \
                dstport, tcp_flags, \
                PgSQL.PgBytea(tcp_data[4*tcp_hdr_len:]), \
                PgSQL.PgBytea(data), org))
  except:
    traceback.print_exc(sys.stderr)
```

Here cu is the database cursor object from the pyPgSQL module.

4.6 The Aggregation Component

The aggregation component will be discussed in this section; from the design decisions to the implementation details. The aggregation component is designed to be run periodically. It has three main functions:

- Provide a summary of commonly used data to other components.
- Provide simple caching of IP address information.
- Gather and store daily data for presentation and visualisation.

4.6.1 Design

The prototype system only provides summery data for TCP SYN-packets. This is done to focus on traffic initiated by customer hosts, and to keep the noise to a minimum. Obviously this is a narrow approach; as seen in chapter 2 botnets could use many possible means of control traffic. But for the purpose of the prototype system focusing on TCP SYN seems a reasonable choice, as it provides information on hosts who have actively tried to initiate contact with controller hosts.

The summery data focuses on the two most important identifiers in the system: Organisation and source IP address. The data stored periodically for every source IP address is:

- Organisation.
- TCP Destination Port.
- Number of TCP SYN packets sent.
- Number of unique controller hosts contacted.
- Time of first and last packet seen during the period, i.e. the active period of the host.
- A score representing the immediate risk the host poses. The details on this measure are given in section 4.7.

These provide the most commonly needed information and ensures that the other components will not have to use expensive queries against the storage system on a per-packet level.

The aggregation component also provides the system with simple caching of IP addresses. By examining packet data for a given period, the component updates lists in the storage system containing information on IP addresses, such as when the address was first and last seen by the system. These lists hold information on both controller hosts and customer hosts. This cached information is used by other components for instance when determining whether any given host is "new" in the given context. Again, the primary goal is to ensure that there is no need for expensive queries to obtain such information.

The presentation component, discussed later in section 4.8 relies on a couple of specialised data combinations in order to plot data needed for daily reports. The aggregation component makes sure the needed information is available on a per-day basis. The following information is gathered and stored:

- For any given organisation, data about activity levels and number of new hosts seen is available. In addition an average score for all hosts in the organisation is calculated.
- The number of active and new controllers seen.

4.6.2 Implementation

The component is implemented in Python. The implementation is fairly straightforward and follows the same basic structure for all aggregating functions:

1. Select the data from the appropriate tables in the storage system.
2. Iterate over the resulting rows, update temporary datastructures and do any calculations.

3. Iterate over the datastructures and insert the aggregated data into the appropriate tables in the storage system.

Whenever possible calculations and aggregations are done directly in SQL to take advantage of the optimiser in the DBMS. A typical query would be something like:

```
SELECT src, org, dstport, COUNT(id) AS packets,
COUNT(DISTINCT dst) AS controllers, MIN(ts) AS first, MAX(ts) AS
last FROM packets WHERE (ts BETWEEN %s AND %s) AND
tcp_flags=2 GROUP BY src, dstport, org ORDER BY controllers DESC
```

The result is a list of source IP addresses, organisations and TCP destination ports, each having aggregated information on the number of TCP SYN packets sent, the number of controller (unique destination) IP addresses contacted and the timestamp of the first and last packets sent during a certain period of time. Only TCP SYN packets are considered by including the condition `tcp_flags=2` in the `WHERE` clause of the SQL statement. Results are sorted according to the number of controllers contacted. `%s` is a placeholder, replaced by a value at the time of the query by the pyPgSQL library.

4.7 The Response Component

The response component will be discussed in this section; from the design decisions to the implementation details. The response component is designed to be run periodically. It has two main functions:

- Calculate the immediate "risk" posed by each host.
- Evaluate the risk and take operational action towards the customer and/or host in question.

In addition the component must commit metainformation to the storage system to ensure that every response can be tracked by security personnel, or combined with a incident response trouble ticket system at some future point in time.

4.7.1 Design

The main challenge when trying to derive a measure for the immediate "risk" posed by any given host is the fact that no connections between the bots and controllers are ever established, and so no actual payload data can be used in the case of TCP. The prototype system is concerned with TCP SYN packets as they represent an attempt to establish a connection. In order to get from data on TCP SYN packets to a risk assessment the following measure is proposed:

For every source IP address seen during a given period of time, let the tuple $(i, \mathbf{p_i})$ be part of the set $\mathbf{P}$, where i is the IP address. Each of the elements $\mathbf{p_i}$ is another set, namely that of all TCP destination ports used by the source IP address i. This means that every $\mathbf{p_i}$ is a set containing at least one element k representing a TCP destination port, that is:

$$\forall\, (i, \mathbf{p_i}) \in \mathbf{P}\, \exists\, k | k \in \mathbf{p_i}.$$

Now, assign to every such IP address i a number s_i representing the "risk" score as:

$$s_i = max(f(u_i) + g(\frac{n_i}{\Delta t_i}) + \sum_{k=0}^{|\mathbf{p_i}|} h(p_{i_k}), 0) \tag{4.1}$$

The score is expressed as the sum of three functions, all contributing some aspect of the risk measurement. If the score is less than 0 the score is considered 0.

The first function, f, considers the number of controller hosts the source IP has tried to contact, denoted u_i in equation 4.1. It is defined as follows:

$$f : \mathbb{N}^+ \mapsto [1, 10]$$

$$f(x) = \begin{cases} x, & x < 10 \\ 10, & x >= 10 \end{cases}$$

The function maps the number of hosts to a number between 1 and 10. More specifically it just represents the number of controllers contacted, but if the source IP has contacted more than 10 controllers it is assigned the number 10, as such a big number nearly outweighs the other parts of the score anyway.

The function g contributes a small positive part of the total score if the source IP address has a sent a large number of packets to controllers per unit of time. It is defined as follows:

$$g : \mathbb{R} \mapsto [0, 2]$$

$$g(x) = \begin{cases} 1, & 5000 < x < 10000 \\ 2, & x > 10000 \\ 0, & \text{otherwise} \end{cases}$$

If the ratio of packets to time is above 5000 (per minute) some small positive adjustment of the total score is in order to catch particularly noisy hosts. The ratio is denoted $\frac{n_i}{\Delta t_i}$ in equation 4.1. It is the number of packets sent by the host during the period in question, n_i divided by the length of that period, Δt_i.

The third function represents a small adjustment for a few well-known TCP destination ports. Of course, port numbers are just that, numbers, and while it is true that any service could use any port, a few port numbers make it *more likely* that a host is compromised and thus a small adjustment is in order. Such an adjustments needs to be kept current, and some kind of feedback-mechanism might be considered. For the prototype system, the function h is defined as follows:

$$h : \mathbb{N} \mapsto [-1,3]$$

$$h(x) = \begin{cases} 3, & x \in \{6667\} \\ 2, & x \in \{1337\} \\ 1, & x \in \{8080\} \\ -1, & x \in \{80,25\} \\ 0, & \text{otherwise.} \end{cases}$$

The function h maps a TCP destination port number to some small number if the port number falls within a predefined set of port numbers. For instance, if the traffic is destined for the standard IRC service port 6667 the function contributes 3 to the overall score.

This function differs from the two others. It represents a loss of generality, as traffic might use other protocols than TCP and also constitutes some maintenance work, in order to keep such a set of ports relevant. However, the prototype system is limited to TCP SYN when analysing data anyway, and the adjustments for port numbers should prove an interesting experiment, without corrupting the total score which is highly dependant on the number of controllers anyway.

The result of this risk assessment is a mapping from IP address to a number, $i \mapsto s_i$ where $s_i \in [0,18]$. This number is believed to be a fair estimate of the immediate "risk" posed by that IP address during the timeperiod in question.

To determine whether to take operational action with regards to a specific IP address one can classify the different addresses according to their score. By partitioning all addresses based on threshold values for instance, the different partitions can be expressed as:

$$\begin{aligned} \mathbf{A_1} &= \{(i,s_i)|(i,\mathbf{p_i}) \in \mathbf{P} \wedge s_i >= T_1\} \\ \mathbf{A_2} &= \{(i,s_i)|(i,\mathbf{p_i}) \in \mathbf{P} \wedge s_i >= T_2\} \\ &\vdots \\ \mathbf{A_n} &= \{(i,s_i)|(i,\mathbf{p_i}) \in \mathbf{P} \wedge s_i >= T_n\} \end{aligned}$$

Each partition $\mathbf{A_i}$ is a set of tuples (i,s_i) of source IP addresses and their corresponding score values.

A suitable operational action can then be assigned to each partition. This could span from a simple notification, via quarantine, to disconnecting the host, for instance through nullrouting. In the case of the prototype system the only action assigned is an automated e-mail message to an organisation with a list of IP addresses and time of activity, in accordance with UNINETT policy.

4.7.2 Implementation

The response component is implemented in the Python language. The score calculation code is put in a module of its own, as it is used by the aggregation component as well. The scoring functions are implemented by the `compute_score` function in a straightforward manner:

```
def compute_score(u, n, p, dt):
    s = 0

    if u > 10: s += 10
    else: s+= u

    mins = dt.minutes
    if mins > 0:
        if float(n)/mins >= 5000: s += 1
        elif float(n)/mins >= 10000: s += 2

    for port in p:
        if port == 6667: s += 3
        elif port == 1337: s += 2
        elif port == 8080: s += 1
        elif port == 80: s -= 1
        elif port == 25: s -= 1

    if s < 0: s = 0

    return s
```

Here `u` is the number of controllers, `n` is the number of packets, `p` is a list of port numbers and `dt` is the amount of time the source IP address was active during the period.

To generate the alerts, data on all source IP addresses having a score above a given threshold are put in a temporary datastructure indexed by organisation. This datastructure is then iterated and e-mail is sent to all organisations, using the address `abuse@<organisation>`. For this purpose the `smtplib` module of the standard Python library is used. A record of the alert is inserted into the `alerts` table in the storage system:

```
cursor.execute("""
insert into alerts(sent_to, ts, org) values (%s, %s, %s)
""", (mail_to, mx.DateTime.gmt(), org))
```

The IP addresses are split into two sections when formatting the mail, one with new addresses and one with addresses previously seen by the system. By using the ip_seen table it is easy to check whether any given address has been seen before:

```
cursor.execute("""
SELECT src FROM ip_seen WHERE src = %s and first <= %s
""",
               (src, fr))
```

Here the start time for the period is given as the argument fr.

4.8 The Presentation Component

The presentation component will be discussed in this section; from the design decisions to the implementation details. The presentation component is designed as dynamic modules for presentation on the World Wide Web, that is modules for generating HTML and images dynamically. The main tasks of this component is:

- Present statistics and daily reports.
- Provide several levels of detail, and make the different levels easy to navigate.
- Generate dynamic images showing recent history.

4.8.1 Design

Daily reports provide security staff with a means to view the recent development for any given organisation and the AS in general. This helps understanding trends, evaluating measures and leads to a better understanding of the problem in general.

The component also provides three different levels of information through the web modules. On the top level information is presented for the entire AS with recent history for every organisation. From there navigation is possible to one specific organisation, showing the details for that particular customer on the given day. Information such as highest scoring source IP addresses and alert information is to be presented at this level.

Finally, the component should offer a per-address view offering more specific information on the offending IP address, such as destination ports used, number of packets sent and score history for the IP address.

To present this information in a convenient manner, the component should be able to visualise historic data dynamically.

4.8.2 Implementation

The presentation component is realised as a set of Mason components. Mason is a Perl-based web site development and delivery engine [18].

The three different levels of information are realised by three separate components, all essentially wrapping the database data in HTML and linking components together. Data access is provided by the Perl DBI-library. Graphs are provided by Mason components outputting binary image data dynamically, using the GD::Graph Perl module:

```
my $sql = "SELECT day, cnt, new_cnt " .
          "FROM org_stats_day " .
          "WHERE day >= (? - reltime('7 days')) and day <= ?" .
          "and org = ? ORDER BY day";

my $most_active = $dbh->selectall_arrayref($sql, undef, $from, $from,
$org);
.
.
use GD::Graph::lines;
$graph = GD::Graph::lines->new(200,70);.
.
.
for my $row (@$most_active) {
  push(@$xvalues, $row->[0]);
  push(@$yvalues, $row->[1]);
  push(@$y2values, $row->[2]);
}

my @data = ($xvalues, $yvalues, $y2values);
my $gd = $graph->plot(\@data) or die $graph->error;

$r->content_type('image/png');
$r->send_http_header;
binmode(STDOUT);
print $gd->png();
```

This simple, yet powerful and flexible approach means it is unnecessary to keep updating and maintaining lots of image files, and also makes it easy to view data from different angles just by changing an URL.

Chapter 5

Observations and Results

The last chapter showed the realisation of the approach, through the prototype system. This chapter will present the observations and results from running the system with live data. All IP addresses and names of customers will be kept anonymous for privacy reasons.

5.1 Environment

All results are based on data gathered by the prototype system setup in a live environment at UNINETT. The data consists of traffic seen between 2005-02-20 12:16:49.28 UTC and 2005-05-10 04:25:06.14 UTC, 79 days in total. All data were first gathered by tcpdump, producing one file per day. These were used to input data to the system, running the logging component script with the filename as input.

5.2 Traffic Characteristics

In this section the overall characteristics of the traffic data will presented. This is traffic destined for known botnet controllers which is redirected to the sinkhole host. These characteristics are important to the understanding of the data. The main figures were:

- Total number of IP Packets: 41 499 052
- Average number of IP packets per second: 9.8
- Total unique source IP addresses: 67679
- Total unique destination IP addresses: 1544

A breakdown by IP protocol is given in table 5.1. TCP accounts for more than 90% of the total number of packets. A further breakdown of the TCP traffic is given in table 5.2. By far, most of the TCP traffic is TCP SYN traffic, attempts to establish outbound connections.

There is also a fair amount of SYN/ACK packets, responses to connection attempts on open ports. These would occur when a known botnet controller tried to access some service at the customer host on an open port. The ACK/RST packets are most likely similar responses from closed ports.

Protocol	Percentage	Packets
TCP	93.02%	38 600 958
UDP	3.73%	1 548 271
ICMP	3.25%	1 349 823

Table 5.1: Data breakdown by IP protocol.

TCP Flags	Percentage (TCP)	Percentage (total)	Packets
SYN	88.27%	82.11%	34 072 907
SYN/ACK	10.76%	10.01%	4 155 105
ACK/RST	0.80%	0.75%	308 973
ACK	0.14%	0.13%	54 012

Table 5.2: TCP Data breakdown by TCP flags.

Although packet filtering rules were in place from the very beginning on the sinkhole host to drop all traffic destined for the capture interface libpcap still saw some external traffic destined for the interface, i.e. traffic not originating within the UNINETT AS. This introduced some extra packet storms and scans from hosts not part of the UNINETT network. This traffic had to be filtered out of all aggregated tables.

The focus of the prototype system became the TCP SYN packets, and all the aggregated tables contain information based only on TCP SYN packets. Now, comparing the number of unique IP addresses seen with the the unique number of IP addresses sending TCP SYN traffic the results are:

- Total number of unique source IP addresses seen: 67 679
- TCP SYN only: 6 116 (9.03%)
- Total number of unique destination IP addresses seen: 1 544
- TCP SYN only: 1 107 (71.72%)

So, while most of the packets seen are TCP SYN packets, the number of unique source IP addresses sending such packets represent only 9% of the total number

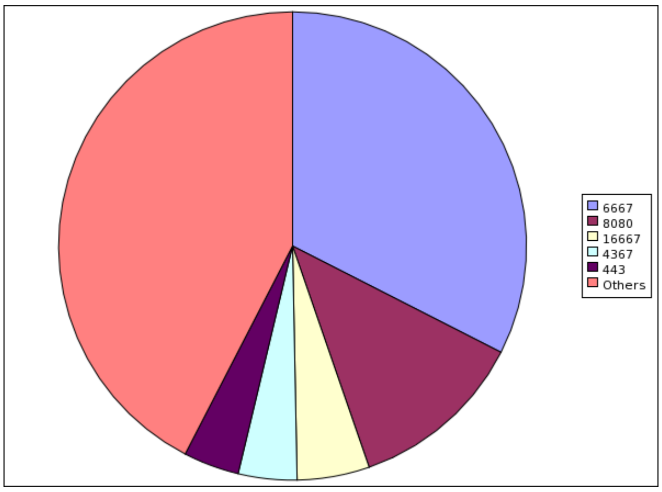


Figure 5.1: A breakdown of the TCP SYN traffic by top 5 TCP destination ports.

of source IP addresses seen. This seems to suggest that the focus on TCP SYN packets was justified.

A breakdown into TCP destination ports for the TCP SYN traffic can be seen in table 5.3, and also in figure 5.1. Nearly one third of all TCP SYN packets seen target port 6667, the standard IRC service port. Of course, a lot of botnets choose other port numbers for their IRC control traffic; other popular choices are 8080, traditionally used for HTTP proxy services and 16667, a simple extension of the number 6667. Together, packets to these three port numbers represent nearly 50% of the total number of TCP SYN packets.

TCP Destination port	Percentage	Packets
6667	32.50%	11 074 045
8080	12.19%	4 155 105
16667	5.00%	1 703 187
4367	4.05%	1 378 931
443	3.85%	1 310 220

Table 5.3: Breakdown of TCP SYN traffic by top 5 TCP destination ports.

A similar breakdown of the TCP SYN/ACK traffic with regards to TCP source port can be seen in table 5.4. Somewhat surprisingly port 23, the telnet service port accounts for nearly 90% of the SYN/ACK traffic.

Upon further investigation the relatively large amount of SYN/ACK traffic is found to comprise of packets from two customer routers, most likely in response to sustained telnet attempts, thus skewing the data heavily. The port 25 response traffic

TCP Source port	Percentage	Packets
23	89.34%	3 712 025
25	4.58%	190 217
80	4.14%	172 049
139	0.30%	12 438
1025	0.28%	11 412

Table 5.4: Breakdown of TCP SYN/ACK traffic by top 5 TCP source ports.

mostly stems from attempts to use SMTP-servers spread throughout the network.

Using the TTL values from the packet data collected, it is possible to break the traffic down in terms of likely initial TTL value, and thereby likely Operating System. Such a breakdown of the TCP SYN traffic can be seen in table 5.5. Packets with a initial TTL value of 128 dominate the data, suggesting that about 96.4% of the packets originated from hosts running a Windows-based OS.

When examining packets sent from the top 500 hosts in terms of score the results are similar, so there is no indication that hosts with higher scores are biased towards any given TTL value, and thereby OS.

TTL-range	Likely initial value	Likely OS	% of packets
107–127	128	Windows	96.40%
43–63	64	Linux	3.2%
Rest	N/A	Other/Unknown	0.4%

Table 5.5: Breakdown of TCP SYN traffic by TTL values.

5.3 Scope and Activity

The number of unique source IP addressees attempting to establish TCP connections with known controllers amount to an estimated 2.3–5.4% of the 150 000–350 000 hosts connected to the UNINETT network.

In some cases one source IP address has attempted to establish contact with more than one controller host on any given day. This has been seen with 2124 of the 6116 addresses, about one in three. Looking at source IP addresses attempting contact with more than five controller hosts, the number is 554, or about 9%. The highest number observed was 36 different controller hosts contacted by the same source IP address over the course of 15 hours (!). As many as 200 hosts have attempted to contact more than 10 controller hosts during a 24 hour period. On average, each source IP address contacted 1.81 controller hosts every 24 hour period (00–24, i.e. every day).

There are many possible reasons why hosts might exhibit this behaviour, some of the more likely are:

- Different hosts may have used the same IP address, as is often the case with VPN, 802.1x and dynamic DHCP-pools.
- Hosts might be connecting to a botnet server consisting of several peering IRC-servers.
- Host may actually be infected multiple times, in effect being part of several botnets at once.

It is the two last possibilities that motivated the scoring system putting emphasis on the number of controllers contacted, as described in section 4.7.1.

The number of unique source IP addresses trying to establish contact with controllers are plotted in figure 5.2 on a day to day basis. The figure also shows the number of new IP addresses seen each day[1]. The dip in activity at the end of March is most likely the result of the easter holidays, and a lot of computers being switched off. On average, there were a total of 200 unique source IP addresses active every day, of which about 76 were new.

Looking at the rate of new IP addresses seen on a daily basis, the following questions are now asked:

- How many new source IP addresses can be expected to be seen each day on average?
- Whats the maximum and minimum number of new source IP addresses one can expect to see each day?
- Once a source IP address is seen, how long does it stay active?

To answer the first question, a 95% confidence interval estimating the mean with basis in the 79 days of empirical data. The following are the parameters of the empirical data:

$$n = 79$$
$$\bar{x} = 76.29$$
$$s = 51.18.$$

Here, n is the number of observations, $\bar{x}$ is the observed mean and s is the observed standard deviation. Now, we cannot assume normality in the distribution, and the variance (σ) of the distribution is unknown. However, since the number of observations is large ($n \geq 30$), s replaces σ and so a confidence interval can be found as:

$$\bar{x} \pm z_{\alpha/2}\, s/\sqrt{n}.$$

[1]Note that the sudden drop in early April was caused by a crash in tcpdump, and data from that day is thus based on 1/8 of a day

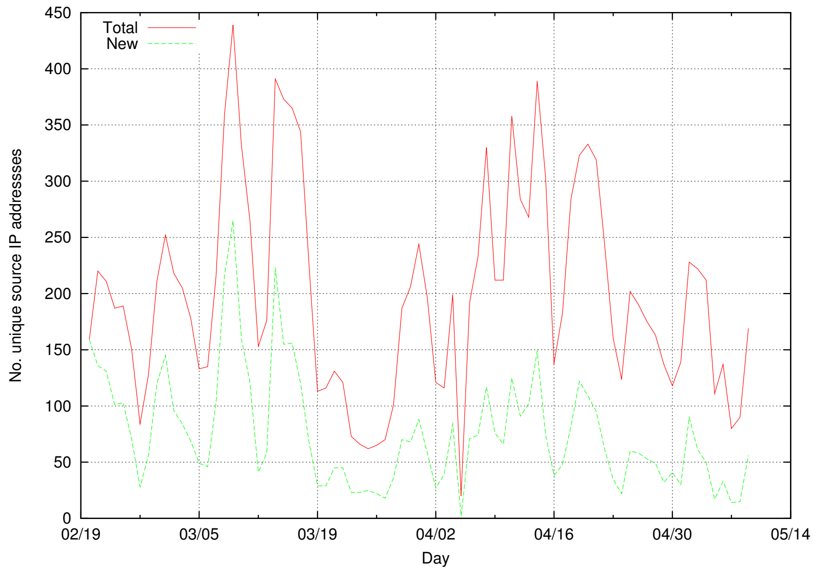


Figure 5.2: Number of unique source IP addresses seen by day.

With $1 - \alpha = 0.95$ a 95% confidence interval on μ is found to be:

$$\mu = 76.29 \pm 11.28,$$

or in other words the interval $[65, 88]$. So, with 95% confidence, the average number of new source IP addresses per day lies in this interval.

To answer the second question, one can calculate a so-called tolerance interval. This allows, based on the empirical data, an interval to be found which would cover a specific percentage of future observations. Such a tolerance interval can be found as:

$$\bar{x} \pm ks,$$

where the parameter k depends on the wanted precision and confidence. To assert with 99% confidence that 95% of future observations will be contained within the interval, a table lookup gives the value $k = 2.414$ (for $n = 80, \gamma = 0.01, 1 - \alpha = 0.95$). This means the interval becomes:

$$76.29 \pm (2.414)(51.18) = 76.29 \pm 123.54,$$

or in other words $[0, 200]$. That is, based on observed data, we can be 99% confident that on 95% of observed days, the number of new source IP addresses seen will lie in this interval.

Now, knowing the prevalence of new IP addresses, it becomes interesting to look at the length of these observations, that is, question three from above. If new source IP addresses appear, and then disappear quickly, never to be seen again the likelihood of there being more than one host using that address seems rather small.

The data show that as many as 3774 or nearly 62% of source IP addresses are never seen again after one day of activity. An 95% confidence interval of the mean, μ is found to be:

$$n = 6116$$
$$\bar{x} = 8.93\,\text{days}$$
$$s = 16.83\,\text{days}$$

With $1 - \alpha = 0.95$ we obtain the following interval for the mean:

$$\mu = 8.93 \pm 0.42,$$

Thus a source IP address is expected to be active between 8,51 and 9,35 days on average, or in other words just over a week. However, the observed distribution is skewed, and the median might be a better estimate of the true μ. The median is found to be about 2,55 hours. That would place the mean between 0 and 13 hours, a very different estimate indeed.

The empirical data is very puzzling, and no possible explanation can be given as to why the activity length is so different between IP addresses. If the observations of a long-lived source IP address had been grouped in two or more distinct clusters one could argue that they were likely to be different physical hosts at different times. However, examining the data indicates a more or less continuous flow of packets from the long-lived IP addresses.

In addition to the source IP address data the system has also gathered information on the controller activity, based on the dynamics of the list used. The activity can be seen in figure 5.3. On average there were 752 controllers on the list, with the number as high as 802 and as low as 698. There were 28 additions every day, on average.

Considering that the number of controllers on the list represents only part of the *actual* amount of botnet controllers active on the net, the actual number of infected hosts in the network ought to be higher than the data indicates. Estimating the amount of compromised hosts in the network is not an easy task for several reasons:

- Without feedback and/or forensics the system can never say with 100% certainty that any host is indeed compromised, although indications can be strong. There are bound to be false positives, even when considering only TCP SYN traffic. However without more extensive testing with customers the false positive rate could be almost anything.

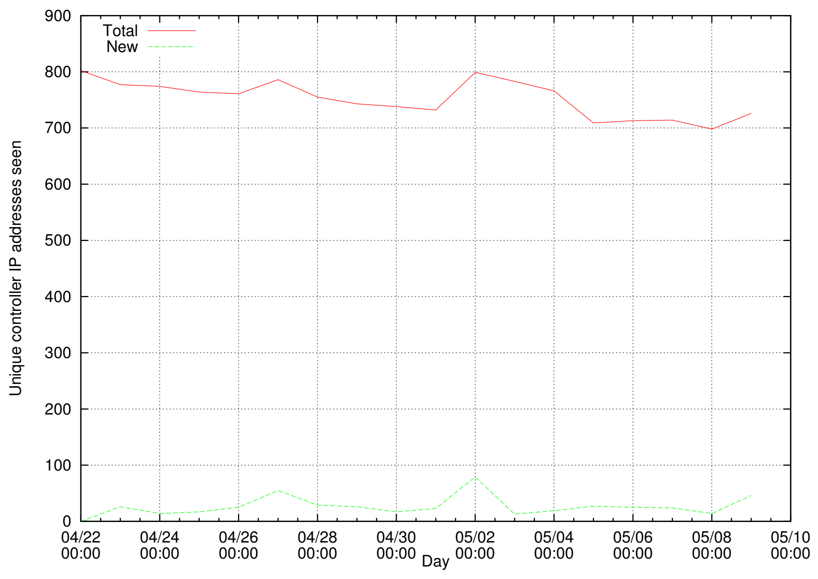


Figure 5.3: Number of controllers on the list on a daily basis.

- There are a lot of "roaming targets"; laptops, VPN, 802.1x and dynamic DHCP-pools complicate matters, there is no one-to-one mapping between an IP address and a host. This undoubtably means that it is easy to overestimate.
- The rate with which compromised hosts are discovered and cleaned up or reinstalled is unknown, but it is safe to assume that cleanups do occur at some rate (however small).
- The controller hosts on the list used only represent a small part of the actual controllers present on the net; however, how small is an open question. In addition, by looking only at TCP SYN traffic when aggregating and scoring, some infected hosts are undoubtably missed.

5.4 An Incident Response Test Case

In order to test the alert component, and to get feedback on the accuracy of the warnings generated, UNINETT contacted two large Norwegian universities to perform a real world test case, with the hope of getting feedback that could help evaluate the precision of the system.

The alert component was run on data for a week, and all source IP addresses with

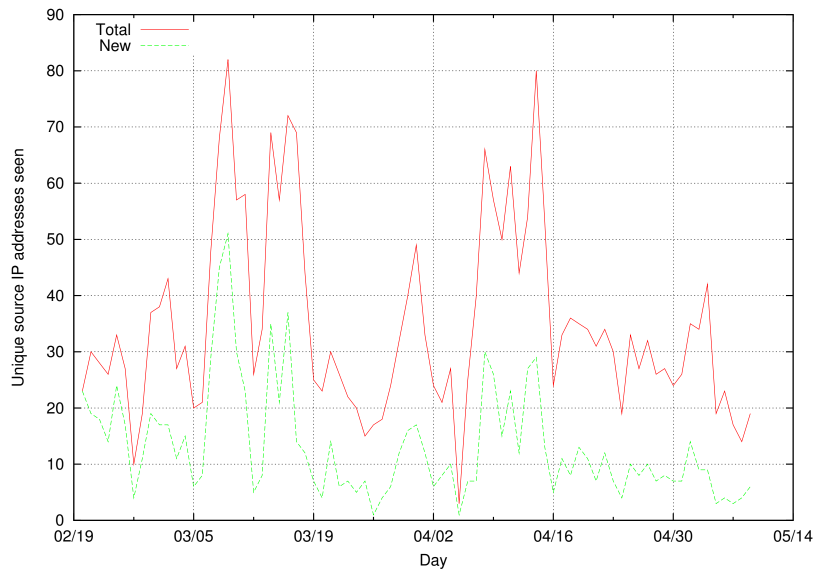


Figure 5.4: Number of unique source IP addresses seen by day for University A.

a calculated score of 2 or more reported by e-mail to the respective universities' abuse addresses. They were given about a month to investigate, and report any findings back. Their approach and findings are discussed next.

5.4.1 University A

University A has shown quite high botnet activity as seen by the system, the activity level is shown in figure 5.4. The data shows some big spikes, and as many as 80 source IP addressees seen on a single day.

The abuse section at university A received a list of 41 unique source IP addresses with details on the period of activity and the number of controller addresses contacted by each address. In addition, the list was split in two parts; one contained new IP addresses (not previously seen by the system), and the other contained IP addresses previously seen by the system.

Methodology

The response team at the university provided the following information on their methodology:

- 41 addresses was considered a big amount, and with other pressing matters also at hand the need for classification arised.
- IP addresses on campus were given priority, and VPN traffic was disregarded due to the man-hours needed to manually examine the logdata.
- The more controllers an IP address had contacted, the higher priority it became.

Also, the team started from the top of the list, with the IP addresses seen previously by the system, and never got to the bottom section, containing new IP addresses in time for the feedback to be included in this thesis.

Results

After a month feedback was given on 20 of the 41 IP addresses reported. When classifying the results, the following categories were chosen:

- **Confirmed positive**. The host was confirmed compromised with some form of malware.
- **Likely negative**. The host was examined, but showed no evidence of compromise.
- **Not investigated**. The host was either not tracked down, or the owner had not responded to inquiries.
- **Reinstalled**. The host was reinstalled before it could be examined.

The results are shown in table 5.6. The amount of likely negative hosts is quite big, however 3 of those 5 hosts represent a special case. Those 3 hosts are part of a web caching project, and so contacts a very large amount of random hosts on port 80. This suggests the need for explicit whitelisting of some sort, as such hosts with special behaviour easily results in false positives.

The other two IP addresses listed as likely negative were reinstalled anyway, to be on the safe side. These had been used by a total of 7 different hosts. These hosts were scanned for viruses. Some even had their traffic logged for a period of time, however no sign of infection could be found.

Category	Hosts	Percentage
Confirmed positive	7	35.0%
Likely negative	5	25.0%
Not investigated	5	25.0%
Reinstalled	3	15.0%

Table 5.6: Results from the abuse team at University A.

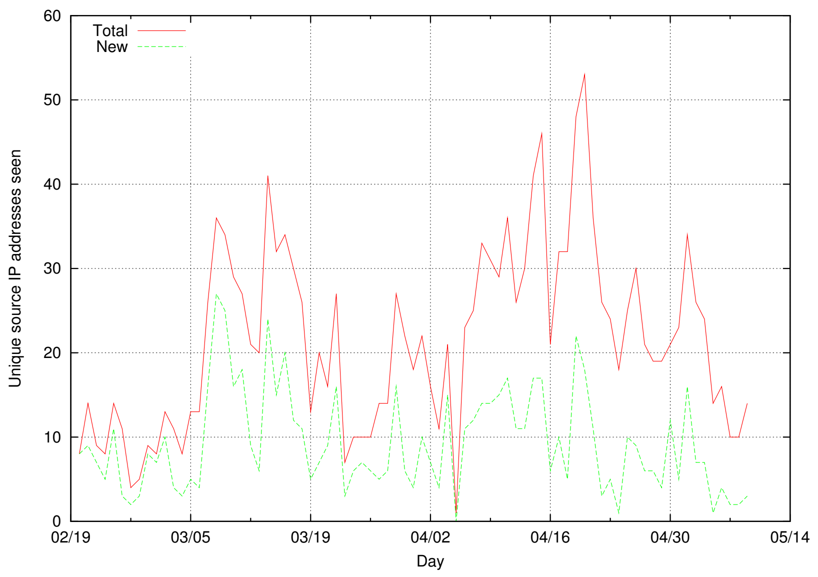


Figure 5.5: Number of unique source IP addresses seen by day for University B.

Most of the hosts not investigated were roaming hosts, typically laptops difficult to track down. Some were other student hosts and although owners were notified, not all replied in time to get a conclusive answer. Unfortunately, these hosts were among the hosts with the highest scores, and without data on these, evaluating the scoring mechanism is difficult. The highest scoring host actually investigated (disregarding the web caching hosts) had contacted 6 controller hosts, and was found to be infected with 6 different trojans.

5.4.2 University B

University B has not shown as much botnet activity as university A, although the two are of comparable size. The activity level is shown in figure 5.5, with activity peaking at about 50 active source IP addresses. Traditionally, university B has had a more restrictive network policy, especially regarding hosts owned by students; this might explain the somewhat smaller activity level.

The list sent to university B contained 40 unique source IP addresses, again in two parts, and sorted by number of controllers contacted.

Methodology

After a month it became clear that university B had taken a somewhat different approach than university A. None of the source IP addresses were hosts owned by the university, that is, they were dial-up systems, VPN and hosts owned by students. As the university had limited access to those, they revoked network access for the addresses and the users and/or owners were asked to clean up the hosts in question. At the time of this writing there has been no further information on this effort.

Results

Regretfully, no hard facts have been provided by university B for any of the hosts on the list; however the feedback indicated that several of the IP addresses had been flagged by their own surveillance systems. At the time of this writing the details on this matter are unfortunately not known.

5.4.3 Summary

In summary, some facts seem clear after the incident response test case:

- Roaming targets, dial-in systems and VPN hosts all make life difficult for the response personnel in terms of tracking down and identifying the problem.
- Customers are bound to take different approaches when dealing with possible infections, especially when the hosts in question are "fringe hosts", not owned by the customer themselves.
- Reports from a system such as this actually *increases* the workload on the personnel at the customer, at least in the short-term; some customers may feel they have little incentive to make such investigation a priority.

5.5 The Prototype System

After running the prototype system in a real-world setting for nearly 80 days the experiences and results gathered are discussed in this section.

5.5.1 Storage

The storage demands of the test setup are split in two. The daily tcpdump packet capture data used as input needed about 6GB of storage space, files averaging about 60-70MB. The PostgreSQL database grew to 20GB during the 80 days, including all indices and other metadata. Even with a 500% increase in daily activity it is likely one could setup a system with a years worth of historic tcpdump raw data and 90 days of database data using about 250GB of storage space.

5.5.2 Performance

A full scale performance test is outside the scope of this thesis, however a few noteworthy points have arisen regarding the performance of the system:

- The big amount of rows means the indices also grow quite big. It becomes important to ensure the DBMS has enough physical memory available to fit the biggest index in RAM.
- `INSERT` statements as used for every packet result in a big performance hit, because the database has to update all indices for every packet. Caching and using `COPY FROM` instead should increase write performance.
- To ensure gathering as many packets as possible, the current approach with tcpdump data as input should be kept, as a live-capture approach in Python with a lot of work per packet is likely to have a higher number of dropped packets.

5.5.3 User Interface

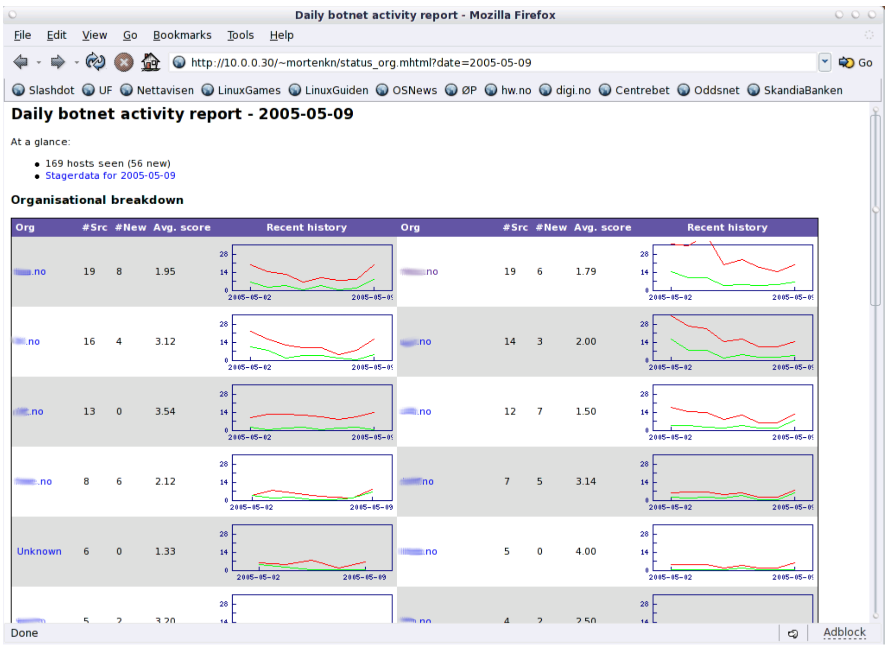
In the prototype system the user interface is comprised of the web frontend with statistics and daily reports. These are dynamically generated from aggregated tables in the storage system. The pages take a top-down approach, presenting information in three levels of increasing detail.

Figure 5.6 shows the topmost level, giving a breakdown per organisation for any given day. For each organisation, a weeks worth of historical data is plotted, showing the activity level with active and new IP addresses seen.

From this page, the user has easy access to more detail on any organisation, full size graphs as well as a link to external NetFlow data for the day. When clicking on a given organisation, the user is presented with the page shown in figure 5.7

Here source IP addresses are ranked according to score, and varying shades of red are used to help visualise the top scorers. For every source IP address the number of TCP SYN packets sent and the number of controllers contacted is listed, as well as the hostname. In addition, if the IP address has been part of an alert, a link to information about the alert is available (number of hosts reported, timestamps, and possibly information from some other incident response ticket system).

The report in figure 5.7 is also available in a stand-alone manner, when no organisation is specified it ranks all IP addresses seen during the specified day, across all


Daily botnet activity report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://10.0.0.30/~mortenkn/status_org.mhtml?date=2005-05-09
Go
Slashdot UF Nettavisen LinuxGames LinuxGuiden OSNews ØP hw.no digi.no Centrebet Oddsnet SkandiaBanken
Done
Adblock

Daily botnet activity report - 2005-05-09

At a glance:

- 169 hosts seen (56 new)
- Stagerdata for 2005-05-09

Organisational breakdown

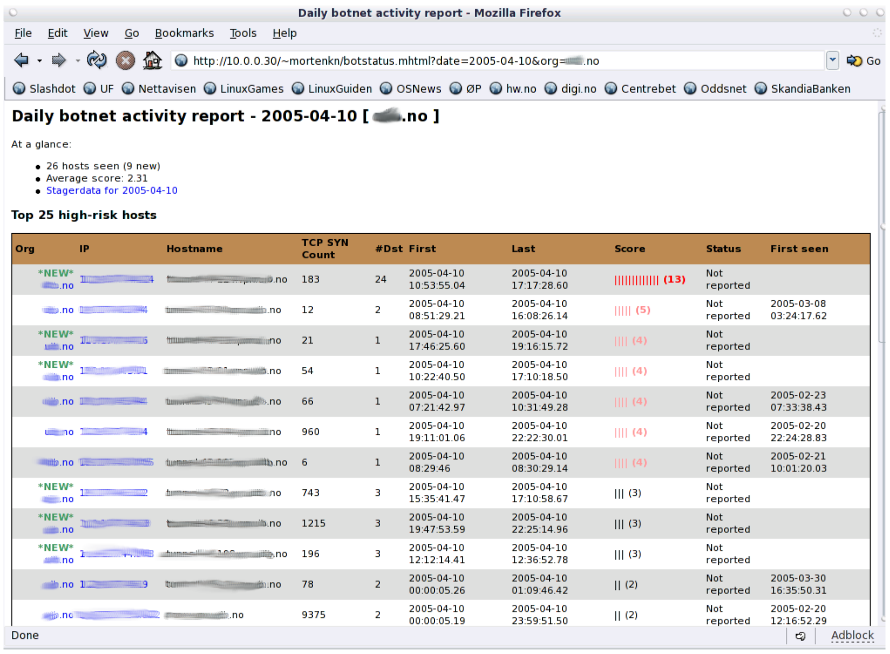
Org	#Src	#New	Avg. score	Recent history	Org	#Src	#New	Avg. score	Recent history
[redacted].no	19	8	1.95		[redacted]no	19	6	1.79	
[redacted].no	16	4	3.12		[redacted].no	14	3	2.00	
[redacted].no	13	0	3.54		[redacted].no	12	7	1.50	
[redacted].no	8	6	2.12		[redacted]no	7	5	3.14	
Unknown	6	0	1.33		[redacted].no	5	0	4.00	
[redacted]	5	2	3.20		[redacted].no	4	2	2.50	

Figure 5.6: Screenshot from the web frontend, showing a breakdown per organisation.

organisations. When clicking on a IP address the user is taken to the third and final level of information, the per IP address detail. The page is shown in figure 5.8. This page shows the activity of the specified IP address on the given day, as well as providing historical data on the TCP destination ports used and the historical score.

5.5.4 Interoperability

The prototype system is built in a fairly modular fashion, with the main component being the storage system. No higher level interface than raw SQL statements exist. However, the modules are relatively simple, well documented, and can be exchanged or expanded with relative ease. This should make it easy to correlate findings with data from other systems and integrate the system in a “whole”.


Daily botnet activity report - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://10.0.0.30/~mortenkn/botstatus.mhtml?date=2005-04-10&org=[redacted].no
Go
Slashdot UF Nettavisen LinuxGames LinuxGuiden OSNews ØP hw.no digi.no Centrebet Oddsnet SkandiaBanken
Done
Adblock

Daily botnet activity report - 2005-04-10 [[redacted].no]

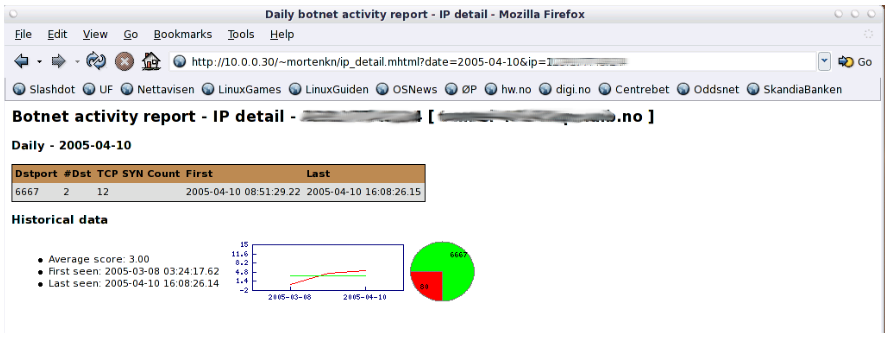
At a glance:

- 26 hosts seen (9 new)
- Average score: 2.31
- Stagerdata for 2005-04-10

Top 25 high-risk hosts

Org	IP	Hostname	TCP SYN Count	#Dst	First	Last	Score	Status	First seen
NEW [redacted].no	[redacted]	[redacted].no	183	24	2005-04-10 10:53:55.04	2005-04-10 17:17:28.60	||||||||||||| (13)	Not reported	
[redacted].no	[redacted]	[redacted].no	12	2	2005-04-10 08:51:29.21	2005-04-10 16:08:26.14	||||| (5)	Not reported	2005-03-08 03:24:17.62
NEW [redacted].no	[redacted]	[redacted].no	21	1	2005-04-10 17:46:25.60	2005-04-10 19:16:15.72	|||| (4)	Not reported	
NEW [redacted].no	[redacted]	[redacted].no	54	1	2005-04-10 10:22:40.50	2005-04-10 17:10:18.50	|||| (4)	Not reported	
[redacted].no	[redacted]	[redacted].no	66	1	2005-04-10 07:21:42.97	2005-04-10 10:31:49.28	|||| (4)	Not reported	2005-02-23 07:33:38.43
[redacted].no	[redacted]	[redacted].no	960	1	2005-04-10 19:11:01.06	2005-04-10 22:22:30.01	|||| (4)	Not reported	2005-02-20 22:24:28.83
[redacted].no	[redacted]	[redacted].no	6	1	2005-04-10 08:29:46	2005-04-10 08:30:29.14	|||| (4)	Not reported	2005-02-21 10:01:20.03
NEW [redacted].no	[redacted]	[redacted].no	743	3	2005-04-10 15:35:41.47	2005-04-10 17:10:58.67	||| (3)	Not reported	
NEW [redacted].no	[redacted]	[redacted].no	1215	3	2005-04-10 19:47:53.59	2005-04-10 22:25:14.96	||| (3)	Not reported	
NEW [redacted].no	[redacted]	[redacted].no	196	3	2005-04-10 12:12:14.41	2005-04-10 12:36:52.78	||| (3)	Not reported	
[redacted].no	[redacted]	[redacted].no	78	2	2005-04-10 00:00:05.26	2005-04-10 01:09:46.42	|| (2)	Not reported	2005-03-30 16:35:50.31
[redacted].no	[redacted]	[redacted].no	9375	2	2005-04-10 00:00:05.19	2005-04-10 23:59:51.50	|| (2)	Not reported	2005-02-20 12:16:52.29

Figure 5.7: Screenshot from the web frontend, showing the most active hosts for a given organisation.


Daily botnet activity report - IP detail - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://10.0.0.30/~mortenkn/ip_detail.mhtml?date=2005-04-10&ip=[redacted]
Go
Slashdot UF Nettavisen LinuxGames LinuxGuiden OSNews ØP hw.no digi.no Centrebet Oddsnet SkandiaBanken
15
11.6
8.2
4.8
1.4
-2
2005-03-08
2005-04-10
6667
80

Botnet activity report - IP detail - [redacted] [[redacted].no]

Daily - 2005-04-10

Dstport	#Dst	TCP SYN Count	First	Last
6667	2	12	2005-04-10 08:51:29.22	2005-04-10 16:08:26.15

Historical data

- Average score: 3.00
- First seen: 2005-03-08 03:24:17.62
- Last seen: 2005-04-10 16:08:26.14

Figure 5.8: Screenshot from the web frontend, providing detail on the given IP address.

Chapter 6

Summary and Further Work

In this thesis a technique for uncovering and reporting botnet activity in a service provider environment has been presented and tested. The technique has proven to be a feasible approach, and has been used in a live environment to reroute botnet traffic to a sinkhole host. Compared to the existing manual, ad-hoc approach to botnet detection and mitigation the approach shows great promise.

A modular prototype system has been developed, enabling traffic analysis of the botnet activity and incident response. This system was run on live traffic data gathered from the Norwegian national research and education network, providing aggregation and classification of packet data and dynamic reports to support the incident response. The system will be moved to production and used by the CERT team as soon as possible.

To test the incident response two Norwegian universities were provided with automatically generated lists of likely compromised hosts over a 7 day period. This was done to get some insight as to the precision of the data and the incident response process as such. Unfortunately the data gathered from this test case was somewhat limited, and it is impossible to conclude on the precision of the system and the effectiveness of the scoring mechanism. The test case did allow us to positively confirm the compromised state of several hosts. It also highlighted the need for the whitelisting of certain specialised hosts. In addition it highlighted the problems with "roaming" hosts, identifying them and getting to them.

Examining the scope of the problem revealed that 6116 unique source IP addresses tried to establish contact with botnet controllers during a 79 day period. Estimating the amount of bots in the network is a very complex matter and the collected data is not conclusive enough to perform such an estimate. However, the numbers suggest that the scope of the problem is severe. The number of expected new likely compromised IP addresses seen per day was found to be between 65 and 88 on average. About one in three compromised IP addresses have contacted two or more unique botnet controllers.

Most of the addresses seen by the system are seen only once, on the day they appear, while others remain active for a longer period of time. While some of this (lack of) sustained activity might be explained by changes in the list used by the system, this observation is difficult to explain.

There are plans to further develop the system and there are many potential areas for future work, some of them are:

- Improving the quality of forensic data. For instance, if the port numbers of the control service of the botnet controllers could be integrated into the system and scoring functions, the precision of the system would certainly increase significantly.
- Providing customers with data to help identify hosts, for instance by lookup in NAV [33] databases where applicable. Customers also need some simple way to report back when a host has been cleaned for instance. A per-customer web portal could enable customers to do this, as well as access historical reports using the same components already developed.
- Correlation across security tools, for instance flow monitoring systems and dark IP space systems. Could be used both to increase precision and as support in forensic work to uncover new controller hosts.
- Tighter integration with trouble ticket incident response system. The system already has mechanisms to store an id with every alert, with this in mind, and the CERT team is looking at the possibility of integration with RTIR [6].

Bibliography

[1] Dynamic Network Services Inc. http://www.dyndns.org/.

[2] pylibcap: Python module for libpcap. http://pylibpcap.sourceforge.net/.

[3] UNINETT: The Norwegian National Research and Education Network. http://www.uninett.no/.

[4] WASTE: Anonymous, secure, encrypted sharing. http://waste.sf.net, 2003–2004.

[5] David Barry. Zombies, Trojans, bots and worms: What have we wrought? *Cisco Packet Magazine*, 17:19–22, 2005.

[6] Best Practical Solutions LLC. Request Tracker for Incident Response. http://www.bestpractical.com/rtir/.

[7] CERT/CC. CERT Advisory CA-2003-20 W32/Blaster worm. http://www.cert.org/advisories/CA-2003-20.html, August 2003.

[8] Cisco. Unicast Reverse Path Forwarding (uRPF) Enhancements for the ISP-ISP Edge. ftp://ftp-eng.cisco.com/cons/isp/documents/uRPF_Enhancement.pdf, February 2001.

[9] Cisco. NetFlow Services and Applications. http://www.cisco.com/warp/public/cc/pd/iosw/ioft/neflct/tech/napps_wp.pdf, July 2002.

[10] Cisco. Remote Triggering Black Hole Filtering. ftp://ftp-eng.cisco.com/cons/isp/essentials/Remote%20Triggered%20Black%20Hole%20Filtering-02.pdf, August 2002.

[11] David Dittrich. The 'stacheldraht' distributed denial of service attack tool. http://staff.washington.edu/dittrich/misc/stacheldraht.analysis, December 1999.

[12] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), March 1997. Updated by RFC 3396.

[13] eEye Digital Security. .ida "Code Red" Worm. http://www.eeye.com/html/research/advisories/al20010717.html, July 2001.

[14] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *IEEE Symposium on Security and Privacy*, pages 326–344, 1989.

[15] David Geer. Malicious Bots Threaten Network Security. *IEEE Computer Magazine*, 38(1):18–20, January 2005.

[16] Michael Glenn. A Summary of DoS/DDoS Prevention, Monitoring and Mitigation Techniques in a Service Provider Environment. August 2003.

[17] Kevin J. Houle, George M. Weaver, Neil Long, and Rob Thomas. Trends in Denial of Service Attack Technology, October 2001.

[18] Jonathan Swartz et.al. Mason: A powerful Perl-based web site development and delivery engine. `http://masonhq.com`.

[19] C. Kalt. Internet Relay Chat: Architecture. RFC 2810 (Informational), April 2000.

[20] mIRC Co. Ltd. Khaled Mardam-Bey. mIRC: A Winsock IRC-client. `http://www.mirc.com`.

[21] Darrell M. Kienzle and Matthew C. Elder. Recent worms: A Survey and Trends. In *WORM'03: Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 1–10. ACM Press, 2003.

[22] Hyang-Ah Kim and Brad Karp. Autograph: Towards Automated, Distributed Worm Signature Detection. In *The 13th USENIX Security Symposium*, pages 271–286, August 2004.

[23] John Kristoff. Botnets. `http://www.nanog.org/mtg-0410/pdf/kristoff.pdf`, October 2004.

[24] LURHQ Threat Intelligence Group. Phatbot Trojan Analysis. `http://www.lurhq.com/phatbot.html`, March 2004.

[25] Brian McWilliams. *Spam Kings*. O'Reilly, 2005.

[26] Daniel Medina. Digging Up Worms, Herding BotNets. `http://www.columbia.edu/acis/networks/advanced/papers/medina-resnet2004.pdf`, June 2004.

[27] Corey Merchant and Joe Stewart. Detecting and Containing IRC-Controlled Trojans: When Firewalls, AV and IDS Are Not Enough. `http://www.securityfocus.com/infocus/1605`, July 2002.

[28] Jelena Mirkovic. *D-WARD: Source-End Defense Against Distributed Denial-of-Service Attacks*. PhD thesis, University of California, 2003.

[29] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004.

[30] P. Mockapetris. RFC 1034: Domain Names - Concepts and Facilities, 1987.

[31] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security & Privacy Magazine*, 1:33–39, 2003.

[32] Network Research Group, Lawrence Berkely National Laboratory. libpcap: A system-independent interface for user-level packet capture. `http://ee.lbl.gov/`.

[33] Norwegian University of Science and Technology. NAV: Network Administration Visualized. `http://metanav.ntnu.no/moin.cgi/NAV`, 1999.

[34] J. Oikarinen and D. Reed. RFC 1459: Internet Relay Chat Protocol, May 1993. Status: EXPERIMENTAL.

[35] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[36] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[37] Puri Ramneek. Bots & Botnet: An Overview. August 2003.

[38] Y. Rekhter and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771 (Draft Standard), March 1995.

[39] Clay Shields. What do we mean by Denial of Service? In *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*, June 2002.

[40] Joseph M. Soricelli and Wayne Gustavus. Tutorial: Options for Blackhole and Discard Routing. `http://www.nanog.org/mtg-0410/soricelli.html`, October 2004.

[41] Spitzner et. al. The Honeynet Project. `http://project.honeynet.org/`, October 1999.

[42] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The Top Speed of Flash Worms. In *WORM '04: Proceedings of the 2004 ACM workshop on Rapid malcode*, pages 33–42. ACM Press, 2004.

[43] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167. USENIX Association, 2002.

[44] swatit.org. Botnet Gallery. `http://swatit.org/bots/gallery.html`, 2003.

[45] The Honeynet Project & Research Alliance. Know your Enemy: Tracking Botnets. `http://www.honeynet.org/papers/bots/`, March 2005.

[46] The Honeynet Project & Research Alliance. Know your Enemy: Tracking Botnets - Spreading. `http://www.honeynet.org/papers/bots/`, March 2005.

[47] Vitalwerks Internet Solutions, LLC. No-IP.com: The Dynamic DNS leader. `http://www.no-ip.com/`.

[48] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A Taxonomy of Computer Worms. In *WORM'03: Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 11–18. ACM Press, 2003.

[49] Nicholas Weaver, Stuart Staniford, and Vern Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the 13th USENIX Security Symposium*, pages 29–44, August 2004.

Appendix A

BGP Configuration

The following BGP configuration was used for the sinkhole router:

```
!
!
router bgp 224
 neighbor bc-feed peer-group
 neighbor bc-feed remote-as 65330
 neighbor bc-feed description Botnet route feed
 neighbor bc-feed ebgp-multihop 200
 neighbor bc-feed password -omitted-
 neighbor bc-feed update-source Loopback0
 !
 neighbor x.x.x.x peer-group bc-feed
 neighbor y.y.y.y peer-group bc-feed
 !
 address-family ipv4
 !
 neighbor bc-feed activate
 neighbor bc-feed prefix-list nothing out
 neighbor bc-feed route-map bc-feed-in in
 neighbor bc-feed maximum-prefix 1000 90
 !
 neighbor x.x.x.x peer-group bc-feed
 neighbor y.y.y.y peer-group bc-feed
 !
 exit-address-family
!
!
route-map bc-feed-in permit 10
```

```
 description Filter Botnet controller routes
 match ip address prefix-list bc-prefixes
 match community 10
 set ip next-hop 128.39.47.174
 set community no-export
!
route-map nothing permit 10
 match ip address 194
!
access-list 194 deny ip any any
!
!
ip prefix-list bc-prefixes description Prefixes to be routed to logger
ip prefix-list bc-prefixes seq 5 permit 0.0.0.0/0 ge 32
!
```

Appendix B

Configuration Scripts

B.1 iptables-drop-eth1.sh

```
#!/bin/sh

# Flush all chains and delete LOG_DROP-chain
iptables -F
iptables -X LOG_DROP

# Add a LOG_DROP-chain
iptables -N LOG_DROP
iptables -A LOG_DROP -j LOG --log-tcp-options --log-ip-options --log-prefix '[IP
TABLES DROP]: '
iptables -A LOG_DROP -j DROP

# Drop and log all incoming packets on eth1
iptables -A INPUT -i eth1 -j LOG_DROP

# Dont accept packets in the forwarding chain
iptables -P FORWARD DROP

# Dont let any packets out through this interface
iptables -A OUTPUT -o eth1 -j LOG_DROP
```

B.2 logrotate.conf

```
.
.
```

```
/home/templog/rawlog {
    missingok
    daily
    postrotate
        killall tcpdump
        nohup /usr/sbin/tcpdump -i eth1 -s 0 -w /home/templog/rawlog &
    endscript
    olddir old
    rotate 365
}
.
.
```

Appendix C

Code Listing for the Prototype System

In this appendix the full source code listing for the prototype system is given. It is also available in electronic form from the thesis home page at: `http://www.idi.ntnu.no/~mortenkn/thesis/`.

C.1 alert.py

```
#!/usr/bin/env python
#
# AlertEngine is part of the Alert Component of the
# botnet detector prototype software.
#
# Written by Morten.Knutsen@idi.ntnu.no
#
# $Id: alert.py,v 1.6 2005/06/11 19:49:23 mortenkn Exp $

import mx.DateTime
import traceback, sys, socket
from smtplib import SMTP

from util import compute_score

class AlertEngine:

    def __init__(self, connection, threshold=2):
        """
        Construct an AlertEngine.

        Construct an AlertEngine, passing a python DB-API 2.0
        compliant connection object and optionally specifying a
        threshold value.
        """
        self.cx = connection
        self.cu = self.cx.cursor()
        self.threshold = threshold
        self.smtp_conn = SMTP("tyholt.uninett.no")

    def alert(self, first, last, org, new, ips):
        """
        Send a mail to an organisation with list of IP addresses.

        Generate and send an alert-mail to the specified org, using
        abuse@org as the To address. first and last specify the period
        of time, and are mx.DateTime.Date objects. new is a list of IP
        addresses that are new to the system. ips is a list of IP
        addresses that have previously been seen by the system.

        """

        # Ensure we have an extra cursor available.
        cursor = self.cx.cursor()

        # Get timestamp of latest alert to this org.
        self.cu.execute("""
        select ts from alerts where org=%s order by ts desc limit 1
        """, (org,))

        ts = self.cu.fetchone()

        # Set TBA on ips whose orgs we have mailed in the last 24 hrs
        # FIXME: New datastructure
        #if ts and mx.DateTime.gmt() - ts[0] < mx.DateTime.Time(24):
        #    for ip in ips + new:
        #        print "Would set tba on ip", ip
        #        cursor.execute("""
        #        SELECT src,
        #        #cursor.execute("""
        #        #UPDATE packets SET tba=%s WHERE src=%s
        #        #""", (True, ip))
        #        #cursor.execute("""
        #       #UPDATE period_data SET tba=%s WHERE src=%s
        #       #""", (True, ip))
        #
        #    return

        # Do normal alert

        mail_from = "cert@uninett.no"
        mail_to = "abuse@" + org
#        mail_to = "mortenk@uninett.no"
        mail_cc = ["cert@uninett.no", "mortenk@uninett.no"]
        mail_subject = "Automatisk varsel om infiserte maskiner"

        # Store information on the alert
        cursor.execute("""
        insert into alerts(sent_to, ts, org) values (%s, %s, %s)
        """, (mail_to, mx.DateTime.gmt(), org))

        # Get the ID
        oid = cursor.oidValue
        cursor.execute("""
        select id from alerts where oid=%s
        """, (oid,))
        alert_id = cursor.fetchone()[0]

        # Format mail
        mail_content = "To: " + mail_to + "\r\n"
        mail_content += "Cc: " + ", ".join(mail_cc) + "\r\n"
        mail_content += "Subject: " + mail_subject + \
                " [#" + str(alert_id) + "]\r\n"
        mail_content += "From: " + mail_from + "\r\n"
        mail_content += """

ADVARSEL OM MULIGE INFISERTE MASKINER
=====================================

Vi har registrert oppførsel som tyder på infeksjon på maskiner
på deres nett. Maskinene har forsøkt å etablere kontakt med
maskiner som er kjente kontrollere, dvs. som fjernkontrolerer
andre maskiner. For mer om botnets se http://en.wikipedia.org/wiki/Botnet.
Ta kontakt om det er noe dere lurer på. Vi håper på tilbakemelding på
funn / desinfeksjoner.

"""
```

```
    mail_content += "Varselet gjelder for perioden %s - %s (UTC).\r\n" % \
        (first, last)
    mail_content += "Følgende maskiner under %s er berørt:\r\n" % (org,)

    if new:
      mail_content += """
* NYE adresser i perioden:
--------------------------

"""
      mail_content += self._generate_list(first, last, new)

    if ips:
      mail_content += """
* Adresser vi har sett tidligere:
---------------------------------

"""

      mail_content += self._generate_list(first, last, ips)

    # And, send it..
    self.smtp_conn.sendmail(mail_from, [mail_to]+mail_cc, mail_content)

  # Not used
    # For future use?
  def escalate(self, org, ips):
    pass

  # Helper method to return a formatted list of IP, period and
  # reverse lookup
  def _generate_list(self, fr, to, l):
    hostnames = {}
    for ip in l:
      hn = ""
      try:
        h = socket.gethostbyaddr(ip)
        if h: hn = h[0]
      except:
        pass
      hostnames[ip] = hn

    ret = ""
```

```
        if len(l) > 1:
            self.cu.execute("""
            select min(ts) as first, max(ts) as last, src, count(distinct dst)
            from packets where (ts between %s and %s) and src in %s and tcp_flags=%s
            group by src order by count desc
            """, (fr, to, tuple(l), 2))
        else:
            self.cu.execute("""
            select min(ts) as first, max(ts) as last, src, count(distinct dst)
            from packets where (ts between %s and %s) and src=%s and tcp_flags=%s
            group by src order by count desc
            """, (fr, to, l[0], 2))

        i = 0
        for row in self.cu.fetchall():
            ret += '-' * 72 + '\n'
            ret += "Tidsrom:\t%s - %s\r\nMaskin:\t\t%s [%s]\r\n" + \
                        "Kontaktet:\t%s kontroller(e)\r\n" % \
            (row[0], row[1], row[2], hostnames[row[2]], row[3])
            i += 1

        return ret

    # Generate and store alert data for every organisation
    def _generate_alerts(self, fr, to):
        cursor = self.cx.cursor()

        try:
            self.cu.execute("""
            SELECT src, org, dstport, count(id) as packets,
            count(distinct dst) as ips, min(ts) as first, max(ts) as
            last, alert_id, tba FROM packets WHERE (ts between %s and
            %s) and tcp_flags=%s and (not dst='128.39.47.174' or dst='128.39.47.150')
            GROUP BY src, dstport, org, alert_id, tba ORDER BY
            ips DESC
            """,
                            (fr, to, 2))

            period_data = {}

            # First, store and index by source IP address
            for row in self.cu.fetchall():
                if not row["src"] in period_data:
                    period_data[row["src"]] = {'src': row["src"],
                                   'packets': row["packets"],
                                   'org': row["org"],
                                   'alert_id': row["alert_id"],
                                   'tba': row['tba'],
                                   'first': row["first"],
                                   'last': row["last"],
                                   'ports': [row["dstport"]],
                                   'ips': row["ips"],
                                   }
    else:
        entry = period_data[row["src"]]
        entry["packets"] += row["packets"]
        if row["first"] < entry["first"]:
            entry["first"] = row["first"]
        if row["last"] > entry["last"]:
            entry["last"] = row["last"]
        if row["dstport"] not in entry["ports"]:
            entry["ports"].append(row["dstport"])
        entry["ips"] += row["ips"]

alert_data = {}

# Loop through and seperate new from old,
# index by org. Calculate scores and check
# against threshold.
for src in period_data:
    cursor.execute("""
    SELECT src FROM ip_seen WHERE src = %s and first <= %s
    """,
                   (src, fr))

    old = cursor.fetchone()

    entry = period_data[src]
    score = compute_score(entry["ips"], entry["packets"],
                          entry["ports"],
                                                              entr

    if score >= self.threshold:
        if not entry["alert_id"]:
            org = entry["org"]
            if not org in alert_data:
                alert_data[org] = {'first': entry["first"],
                                   'last': entry["last"],
                                                'ips': [],
                                                'new': []
                                                }
                                if old:
                                    alert_data[org]["ips"] = [src]
                                else:
                                    alert_data[org]["new"] = [src]
                            else:
                                if entry["first"] < alert_data[org]["first"]:
                                    alert_data[org]["first"] = entry["first"]
                                if entry["last"] > alert_data[org]["last"]:
                                    alert_data[org]["last"] = entry["last"]
                                if old:
                                    if src not in alert_data[org]["ips"]:
                                        alert_data[org]["ips"].append(src)
                                else:
                                    if src not in alert_data[org]["new"]:
                                        alert_data[org]["new"].append(src)
                        else:
                            pass
                            # self.escalate(entry) if last > 14 days

                # Make it happen, call alert() for every org
                for org in alert_data:
                    entry = alert_data[org]
                    self.alert(entry["first"], entry["last"],
                                                    org, entry["new"], entry["ips"])

            except:
                traceback.print_exc()


        # Not currently in use...
        def _handle_tba(self, fr, to):
            cursor = self.cx.cursor()
            try:
                self.cu.execute("""
                select min(ts) as first, max(ts) as last, src, org
                            from packets where (ts between %s and %s) where tcp_flags=%s
                            and (not dst='128.39.47.150' or dst='128.39.47.174')
                            and tba=%s group by org, src, dstport
                """,
                                (fr, to, 2, True))
            alert_data = {}

            for row in self.cu.fetchall():
                (first, last, src, org, dstport) = row

                cursor.execute("""
                select ts from alerts where org=%s order by ts desc limit 1
                """, (org,))

                ts = cursor.fetchone()[0]

                if mx.DateTime.gmt() - ts < mx.DateTime.Time(24):
                    continue

                if not org in alert_data:
                    alert_data[org] = {'first': first,
                                       'last': entry["last"],
                                       'ips': [src]
                                       }
                else:
                    if first < alert_data[org]["first"]:
                        alert_data[org]["first"] = first
                    if last > alert_data[org]["last"]:
                        alert_data[org]["last"] = last
                    if src not in alert_data[org]["ips"]:
                        alert_data[org]["ips"].append(src)


            for org in alert_data:
                entry = alert_data[org]
                self.alert(entry["first"], entry["last"], org, entry["ips"])

        except:
            traceback.print_exc()

# Unit test main()
if __name__ == '__main__':

    from pyPgSQL import PgSQL

    conn = PgSQL.connect(database="boned", host="localhost", port="8888", use
                         password="XXXXX")

    conn.autocommit = 1
```

```
        test = AlertEngine(conn)

        startDate = mx.DateTime.DateTime(2005, 4, 17)
        stopDate = mx.DateTime.DateTime(2005, 4, 23)

        test._generate_alerts(startDate, stopDate)
```

C.2 sniff.py

```
#!/usr/bin/env python
#
# sniff.py is part of the Collector Component of the
# botnet detector prototype software.
#
# Written by Morten.Knutsen@idi.ntnu.no
#
# $Id: sniff.py,v 1.4 2005/06/11 19:49:23 mortenkn Exp $

import sys
import pcap
import string
import time
import socket
import struct
import traceback
import mx.DateTime

protocols={socket.IPPROTO_TCP:'tcp',
           socket.IPPROTO_UDP:'udp',
           socket.IPPROTO_ICMP:'icmp'}

cu = None
cnt = 0

from ipreg import IPreg
from pyPgSQL import PgSQL

def store_packet(pktlen, data, timestamp):
  """
  Per-packet function to analyse headers and store info in a table.
  """
```

```
    global cu
    if not data:
      return

    # Check if we have an IP packet
    if data[12:14] == '\x08\x00':
      timestamp = mx.DateTime.gmtime(timestamp)
      ip_data = data[14:]
      ip_hlen = ord(ip_data[0]) & 0x0f

      ttl = ord(ip_data[8])
      proto = ord(ip_data[9])
      src = pcap.ntoa(struct.unpack('i',ip_data[12:16])[0])
      dst = pcap.ntoa(struct.unpack('i',ip_data[16:20])[0])

      # Lookup organisation from source IP address
      ip = ipreg.lookup(src)
      org = "Unknown"
      if not (len(ip) < 2 or ip[2] == '-'):
        org = ip[2]

      # Get ICMP specific information, and insert
      if proto == socket.IPPROTO_ICMP:
        try:
          icmp_data = ip_data[4*ip_hlen:]
          icmp_type = ord(icmp_data[0])
          icmp_code = ord(icmp_data[1])

          cu.execute("""
          insert into packets (ts, ttl, protocol, src, dst,
          icmp_type, icmp_code, data, raw_packet, org) values (
          %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
          """,
                     (timestamp, ttl, proto, src, dst, icmp_type, \
                      icmp_code, PgSQL.PgBytea(icmp_data[4:]), \
                      PgSQL.PgBytea(data), org))
        except:
          traceback.print_exc(sys.stderr)

      # Get UDP specific information, and insert
      elif proto == socket.IPPROTO_UDP:
        try:
          udp_data = ip_data[4*ip_hlen:]
          srcport = socket.ntohs(struct.unpack('H', udp_data[0:2])[0])
          dstport = socket.ntohs(struct.unpack('H', udp_data[2:4])[0])
          length = socket.ntohs(struct.unpack('H', udp_data[4:6])[0])

          cu.execute("""
          insert into packets (ts, ttl, protocol, src, dst,
          srcport, dstport, len, data, raw_packet, org) values (
          %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
          """,
                     (timestamp, ttl, proto, src, dst, \
                      srcport, dstport, length, PgSQL.PgBytea(udp_data[8:]), \
                      PgSQL.PgBytea(data), org))
        except:
          traceback.print_exc(sys.stderr)

      # Get TCP specific information, and insert
      elif proto == socket.IPPROTO_TCP:
        try:
          tcp_data = ip_data[4*ip_hlen:]
          srcport = socket.ntohs(struct.unpack('H', tcp_data[0:2])[0])
          dstport = socket.ntohs(struct.unpack('H', tcp_data[2:4])[0])
          tcp_hdr_len = (ord(tcp_data[12]) & 0xf0) >> 4
          tcp_flags = ord(tcp_data[13]) & 0x3f

          cu.execute("""
          insert into packets (ts, ttl, protocol, src, dst,
          srcport, dstport, tcp_flags, data, raw_packet, org) values
          (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
          """,
                     (timestamp, ttl, proto, src, dst, srcport, \
                      dstport, tcp_flags, PgSQL.PgBytea(tcp_data[4*tcp_hdr_len:]), \
                      PgSQL.PgBytea(data), org))
        except:
          traceback.print_exc(sys.stderr)

  # main() wants a filename with an input file
  # in libpcap format
  if __name__=='__main__':
    global cnt
    p = pcap.pcapObject()

    if not len(sys.argv) > 1:
      print "Need filename.."

  p.open_offline(sys.argv[1])

  cx = PgSQL.connect(host="localhost", database="boned", user="boned")
  cx.autocommit = 1
  cu = cx.cursor()

  # Init ipreg
  ipreg = IPreg()

  try:
    p.loop(-1, store_packet)

  except KeyboardInterrupt:
    p.close()
    print '%s' % sys.exc_type

  except:
    p.close()
    pass
```

C.3 report.py

```
#!/usr/bin/env python
#
# report.py is part of the Aggregation Component of the
# botnet detector prototype software.
#
# Written by Morten.Knutsen@idi.ntnu.no
#
# $Id: report.py,v 1.7 2005/06/11 19:49:23 mortenkn Exp $

import mx.DateTime
import traceback, sys

from util import compute_score

class ReportEngine:

    def __init__(self, connection):
                """
                Construct a ReportEngine.

                Construct a ReportEngine, passing a python DB-API 2.0
                compliant connection object.
                """
            self.cx = connection
            self.cu = self.cx.cursor()

            # Aggregate data pr. source IP address for a given period
        def _update_period_stats(self, fr, to):
            cursor = self.cx.cursor()

            try:
                self.cu.execute("""
                SELECT src, org, dstport, count(id) as packets,
                count(distinct dst) as ips, min(ts) as first, max(ts) as
                last, alert_id, tba FROM packets WHERE (ts between %s and
                %s) and tcp_flags=%s and (not dst='128.39.47.150' or dst='128.39.47.174')
                      GROUP BY src, dstport, org, alert_id, tba ORDER BY ips DESC
                """,
                        (fr, to, 2))

                period_data = {}

                for row in self.cu.fetchall():
                    if not row["src"] in period_data:
                        period_data[row["src"]] = {'src': row["src"],
                                                   'packets': row["packets"],
                                                   'org': row["org"],
                                                   'alert_id': row["alert_id"],
                                                   'tba': row['tba'],
                                                   'first': row["first"],
                                                   'last': row["last"],
                                                   'ports': [row["dstport"]],
                                                   'ips': row["ips"]
                                                   }
                    else:
                        entry = period_data[row["src"]]
                        entry["packets"] += row["packets"]
                        if row["first"] < entry["first"]:
                            entry["first"] = row["first"]
                        if row["last"] > entry["last"]:
                            entry["last"] = row["last"]
                        if row["dstport"] not in entry["ports"]:
                            entry["ports"].append(row["dstport"])
                        entry["ips"] += row["ips"]

            for src in period_data:
                entry = period_data[src]
                score = compute_score(entry["ips"], entry["packets"],
                                      entry["ports"], entry["last"] - entry["f

                cursor.execute("""
                insert into period_data(src, org, ips, packets, first, last,
                score, alert_id, tba) values (%s, %s, %s, %s, %s, %s, %s, %s,
                """,
                (src, entry["org"], entry["ips"], entry["packets"],
                                 entry["first"], entry["last"], score,
                                 entry["alert_id"], entry["tba"]))
        except:
            traceback.print_exc()

        # Aggregate controller data per day
    def _update_controller_stats(self, fr, to):

        try:
            #
            # Controller-stats
            #
            self.cu.execute("""
            SELECT count(address) FROM controllers WHERE
            last >= %s and first <= %s
            """,
                          (fr, to))

            tot, new, reactive = 0,0,0

            row = self.cu.fetchone()
            if row: tot = row[0]

            self.cu.execute("""
            SELECT count(address) FROM controllers WHERE
            first >= %s and first <= %s and address NOT IN
            (SELECT address FROM controllers WHERE last <= %s)
            """,
                             (fr, to, fr))

            row = self.cu.fetchone()
            if row: new = row[0]
```

```
                self.cu.execute("""
                SELECT count(address) FROM controllers WHERE
                first >= %s and first <= %s and address IN
                (SELECT address FROM controllers WHERE last <= %s)
                """,
                                (fr, to, fr))

                row = self.cu.fetchone()
                if row: reactive = row[0]

                self.cu.execute("""
                INSERT INTO controller_stats_day(day, cnt, new_cnt, reactive_cnt)
                VALUES(%s, %s, %s, %s)
                """,
                                (fr, tot, new, reactive))

            except:
                traceback.print_exc()

            # Aggregate activity level per org per day.
        def _update_graph_stats(self, fr, to):

            try:
                self.cu.execute("""
                SELECT distinct org, src, score FROM period_data WHERE
                first >= %s and last <= %s
                """,
                                (fr, to))

                org_data = {}

                for row in self.cu.fetchall():
                    if not row["org"] in org_data:
                        org_data[row["org"]] = {'ips': [row["src"]],
                                                'sum': row["score"],
                                                'new': 0,
                                                }
                    else:
                        entry = org_data[row["org"]]
                        entry["ips"].append(row["src"])
                        entry["sum"] += row["score"]

                self.cu.execute("""
                SELECT src FROM ip_seen WHERE first < %s and src NOTNULL
            """,
                            (fr,))

            old_ips = {}
            for row in self.cu.fetchall():
                old_ips[row["src"]] = 1

            for org in org_data:
                entry = org_data[org]
                for src in entry["ips"]:
                    if src not in old_ips:
                        entry["new"] += 1

                self.cu.execute("""
                INSERT INTO org_stats_day(day, org, cnt, new_cnt, avg_score)
                values(%s, %s, %s, %s, %s)
                """,
                                (fr, org, len(entry["ips"]), entry["new"],
                                 float(entry["sum"])/len(entry["ips"])))


        except:
            traceback.print_exc()

        # Update ip_seen table with nem timestamps and IP addresses
    def _update_ips_seen(self, fr, to):
        try:
            self.cu.execute("""
            select min(ts) as first, max(ts) as last, src, dst from packets
                        where (ts between %s and %s) and tcp_flags=%s and
                        (not dst='128.39.47.150' or dst='129.39.47.174') grou
            """,
                            (fr, to, 2))

            for row in self.cu.fetchall():
                (first, last, src, dst) = row
                self._insert_or_update_ip_seen(first, last, src, dst)

        except:
            traceback.print_exc()

        # Helper method for _update_ip_seen()
    def _insert_or_update_ip_seen(self, first, last, src, dst):
        try:
```

```
        cursor = self.cx.cursor()
        cursor.execute("""
        select id, first, last from ip_seen where src=%s
        """,
                    (src,))

        row = cursor.fetchone()
        if row:
            (id, f, l) = row
            if first < f:
                cursor.execute("""
                update ip_seen set first=%s where id=%s
                """,
                            (first, id))
            elif last > l:
                cursor.execute("""
                update ip_seen set last=%s where id=%s
                """,
                            (last, id))
        else:
            cursor.execute("""
            insert into ip_seen (src, first, last) values (%s, %s, %s)
            """,
                            (src, first, last))

        cursor.execute("""
        select id, first, last from ip_seen where dst=%s
        """,
                    (dst,))

        row = cursor.fetchone()
        if row:
            (id, f, l) = row
            if first < f:
                cursor.execute("""
                update ip_seen set first=%s where id=%s
                """,
                            (first, id))
            elif last > l:
                cursor.execute("""
                update ip_seen set last=%s where id=%s
                """,
                            (last, id))
        else:
                cursor.execute("""
                insert into ip_seen (dst, first, last) values (%s, %s, %s)
                """,
                               (dst, first, last))
        except:
            traceback.print_exc()

# Unit test main()
if __name__ == '__main__':

    from pyPgSQL import PgSQL

    conn = PgSQL.connect(database="boned", host="localhost", port="8888", use
                         password="XXXXX")

    conn.autocommit = 1

    test = ReportEngine(conn)

    startDate = mx.DateTime.Date(2005, 5, 10)
    stopDate = mx.DateTime.Date(2005, 5, 11)
    test._update_ips_seen(startDate, stopDate)
    test._update_period_stats(startDate, stopDate)
    test._update_graph_stats(startDate, stopDate)
    test._update_controller_stats(startDate, stopDate)
```

C.4 util.py

```
#
# Small utility module for use with the
# botnet detector prototype software.
#
# Written by Morten.Knutsen@idi.ntnu.no


def compute_score(u, n, p, dt):
        """
        Score a source IP address based on behaviour.

        Score a source IP address based on bahaviour. Takes the following
        arguments:
```

```
        u: Number of unique botnet controllers contacted.
        n: Number of TCP SYN packets sent.
        p: List of destination port numbers used.
        dt: Interval of time IP address was active (mx.DateTime)

        Returns a computed score.
        """
    s = 0

    if u > 10: s += 10
    else: s+= u

    mins = dt.minutes
    if mins > 0:
        if float(n)/mins > 5000: s += 1

    for port in p:
        if port == 6667: s += 3
        elif port == 1337: s += 2
        elif port == 8080: s += 1
        elif port == 80: s -= 1
        elif port == 25: s -= 1

    if s < 0: s = 0

    return s
```

C.5 packets.sql

```
--
-- The packets table definition,
-- part of the botnet detector prototype software.
--
-- Written by Morten.Knutsen@idi.ntnu.no
--

drop table packets cascade;

create table packets(
    id serial primary key,
    ts timestamp,
    ttl int2,
    protocol int2,
    src inet,
    srcport int,
    dst inet,
    dstport int,
    icmp_type int2,
    icmp_code int2,
    tcp_flags int2,
    len int,
    data bytea,
    raw_packet bytea,
    org text,
    alert_id int
);

create index src_index on packets (src);
create index srcport_index on packets (srcport);
create index time_index on packets (ts);
create index dstport_index on packets (dstport);
create index dst_index on packets (dst);
create index org_index on packets (org);
create index proto_index on packets (protocol);

--
-- Small plpgsql-function to convert tcp_flags integer
-- to human-readable formatted string. Useful when
-- querying the database directly.
--
create or replace function print_tcp_flags(smallint) returns text as '
declare
 retstr text;
 flags alias for $1;
begin
 retstr := ''S A R P U F'';
 if not (flags & 1 = 1) then
   select into retstr translate(retstr,''F'','' '');
 end if;
 if not (flags & 2 = 2) then
   select into retstr translate(retstr, ''S'', '' '');
 end if;
 if not (flags & 4 = 4) then
   select into retstr translate(retstr, ''R'', '' '');
 end if;
```

```
  if not (flags & 8 = 8) then
    select into retstr translate(retstr, ''P'', '' '');
  end if;
  if not (flags & 16 = 16) then
    select into retstr translate(retstr, ''A'', '' '');
  end if;
  if not (flags & 32 = 32) then
    select into retstr translate(retstr, ''U'', '' '');
  end if;
  return retstr;
end;
' language 'plpgsql';
```

C.6 aggregated tables.sql

```
--
-- The table definitions of the aggregated tables,
-- part of the botnet detector prototype software.
--
-- Written by Morten.Knutsen@idi.ntnu.no
--


drop table period_data cascade;

create table period_data (
  id serial primary key,
  src inet,
  packets int,
  ips int,
  first timestamp,
  last timestamp,
  score int,
  alert_id int,
  tba boolean
);

drop table org_stats_day cascade;

create table org_stats_day (
  day date,
  org text,
  cnt int,
  new_cnt int,
  avg_score float,
  primary key(day, org)
);

drop table ip_alerts cascade;

create table ip_alerts (
  ip inet,
  alert_id int,
  tba boolean,
  primary key(ip, alert_id)
);
drop table ip_seen cascade;

create table ip_seen (
  id serial primary key,
  src inet,
  dst inet,
  first timestamp,
  last timestamp
);

drop table controllers cascade;

create table controllers (
  id serial primary key,
  address inet,
  first timestamp,
  last timestamp
);

drop table controller_stats_day cascade;

create table controller_stats_day (
  day date primary key,
  cnt int,
  new_cnt int,
  reactive_cnt int
);
```

C.7 status org.mhtml

```
<%doc>
Component to display botnet activity for a given day.
Breaks down data on a per-organisation level, and presents
recent trend for every organisation.

Takes one argument, date, a date string such as:
"2005-04-09" for 9th of April 2005. If no date is given,
the current date is assumed.
</%doc>

<html>
<head>
<title>Daily botnet activity report</title>
<style type="text/css">
table { border: solid 1px; border-spacing: 0px; }
td,th { padding: 0.3em; text-align: left }
td { padding: 0.5em }
th { padding-left: 0.4em; color: white; background-color: #6355a4 }
td.hilight { background-color: #ddff88 }
td.bar { background-color: #ddd }
</style>
</head>

<body>
<h2>Daily botnet activity report - <% $from %></h2>

<p>At a glance:</p>
<ul>
<li><% $total %> hosts seen (<% $total_new %>
new)</li>
<li><a
  href="https://stager.uninett.no/index.php?ss_db=stager&ss_tablelimit=20&ss_reportsty
</ul>

<h3>Organisational breakdown</h3>

<table>
  <tr>
    <th>Org</th><th>#Src</th><th>#New</th><th>Avg. score</th>
    <th style="text-align: center">Recent history</th>
    <th>Org</th><th>#Src</th><th>#New</th><th>Avg. score</th>
    <th style="text-align: center">Recent history</th>
  </tr>
% my $i = 0;
% my $class = "foo";
% for ($i=0; $i < $#{@$most_active}; $i+=2) {
% my ($left, $right) = ($most_active->[$i], $most_active->[$i+1]);
% $class = ($class eq "foo") ? "bar" : "foo";
  <tr>
    <td class="<% $class %>">
      <a href="botstatus.mhtml?date=<% $from %>&org=<% $left->[0] %>">
          <% $left->[0] %></a></td>
    <td class="<% $class %>"><% $left->[1] %></a></td>
    <td class="<% $class %>"><% $left->[2] %></td>
    <td class="<% $class %>" align="right"><% sprintf("%.2f", $left->[3]) %>
        </td><td class="<% $class %>" align="right"><a
    href="plot_org_history.mhtml?org=<% $left->[0]%>&date=
        <% $from %>&full=1"><img style="border: 0" src="plot_org_history.mhtml
        org=<% $left->[0]%>&date=<% $from %>"/></a></td>

% $class = ($class eq "foo") ? "bar" : "foo";
      <td class="<% $class %>">
      <a href="botstatus.mhtml?date=<% $from %>&org=<% $right->[0] %>">
          <% $right->[0] %></a></td>
    <td class="<% $class %>"><% $right->[1] %></a></td>
    <td class="<% $class %>"><% $right->[2] %></td>
    <td class="<% $class %>" align="right"><% sprintf("%.2f", $right->[3]) %>
        </td><td class="<% $class %>" align="right"><a
    href="plot_org_history.mhtml?org=<% $right->[0]%>&date=<% $from %>&full=1"
        <img style="border: 0" src="plot_org_history.mhtml?org=
        <% $right->[0]%>&date=<% $from %>"/></a></td>
  </tr>
% $class = ($class eq "foo") ? "bar" : "foo";
% }
</table>

</body>
</html>

<%args>
  $date => undef
</%args>

<%init>

```

```
use Socket 'AF_INET';
use DBI;

my $from = $date;

if (not defined($from)) {
  my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime();
  $from = sprintf "%4d-%02d-%02d 00:00", $year+1900,$mon+1,$mday;
}

my ($year, $month, $day) = split /-/, $from;
my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT org, cnt, " .
          "new_cnt, avg_score FROM org_stats_day " .
          "WHERE day = ? ORDER BY cnt DESC";

my $most_active = $dbh->selectall_arrayref($sql, undef, $from);

my $total = 0;
my $total_new = 0;

for my $row (@$most_active) {
  $total += $row->[1];
  $total_new += $row->[2]
}

</%init>
```

C.8 ip detail.mhtml

```
<%doc>
Component to display botnet activity for a specific source
IP address on a given day. Shows traffic amounts and number
of controllers contacted, as well as port numbers.

Also presents historical port distribution and score history
for the IP address in question.

Takes two arguments:

date: a date string such as:
```

```
"2005-04-09" for 9th of April 2005.

ip: the source IP address as a string.
</%doc>

<html>
<head>
<title>Daily botnet activity report - IP detail</title>
<style type="text/css">
table { border: solid 1px; border-spacing: 0px; }
td,th { padding: 0.3em; text-align: left }
th { background-color: #bb8855 }
td.hilight { background-color: #ddff88 }
td.bar { background-color: #ddd }
</style>
</head>

<body>
<h2>Botnet activity report - IP detail - <% $ip %>
% my @h = gethostbyaddr(pack('C4',split('\.', $ip)),2);
% if (@h) {
 [ <% $h[0] %> ]
% }
</h2>

<h3>Daily - <% $date %></h3>
<table>
  <tr>
    <th>Dstport</th><th>#Dst<th>TCP SYN Count</th>
    <th>First</th><th>Last</th>
  </tr>
% my $class = "foo";
% for my $entry (@$detail_data) {
% $class = ($class eq "foo") ? "bar" : "foo";
  <tr>
    <td class="<% $class %>"><% $entry->[1] %></td>
    <td class="<% $class %>"><% $entry->[0] %></td>
    <td class="<% $class %>"><% $entry->[2] %></td>
    <td class="<% $class %>"><% $entry->[3] %></td>
    <td class="<% $class %>"><% $entry->[4] %></td>
  </tr>
% }
</table>

```

```
<h3>Historical data</h3>
<table style="border: none"><tr><td>
<ul>
<li>Average score: <% $avg_scr %></li>
<li>First seen: <% $historical_data[0] %></li>
<li>Last seen: <% $historical_data[1] %></li>
</ul>
<td>
<td><a href="plot_score_ip.mhtml?ip=<% $ip %>&stop=<% $date %>&avg=<%
$avg_scr %>&full=1">
<img style="border: none" src="plot_score_ip.mhtml?ip=<% $ip %>&stop=<%
$date %>&avg=<% $avg_scr %>"/>
</a>
</td>
<td><img style="border: none" src="plot_pie_chart.mhtml?ip=<% $ip
%>&whole=1"/>
</td>
</tr></table>

%# my $pingres = `ping -w 2 $ip`;
%# my $p = 1;
%# if ($pingres =~ /(\d+) received/g) {
%#   $p = 0 if $1 == 0;
%# }

%#<h3>Traceroute</h3>
%# if (!$p) {
%#  <p>Host seems <span style="font-weight: bold; color: red">
%# down</span></p>
%# }
%# else {
%#<pre>
%#<% `traceroute -w 2 $ip` %>
%#</pre>
%# }

%#<h3>Whois-info:</h3>
%#<pre>
%# $whois->lookup('Domain' => $ip);
%#<% $whois->response() %>
%#</pre>

</body>
</html>
```

```

<%args>
  $date
  $ip
</%args>

<%init>

use Socket 'AF_INET';
use DBI;
use Net::XWhois;

my $from = $date;

my ($year, $month, $day) = split /-/, $from;
my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT count(distinct dst) as ips, dstport, count(id)," .
          "min(ts), max(ts) FROM packets " .
          "WHERE (ts between ? and (? + reltime('1 day'))) and " .
          "src=? GROUP BY dstport ORDER BY ips DESC";

my $detail_data = $dbh->selectall_arrayref($sql, undef, $from, $from, $ip);

$sql = "SELECT min(first), max(last), avg(score) FROM period_data ".
       "WHERE src=?";

my @historical_data = $dbh->selectrow_array($sql, undef, $ip);

my $avg_scr = sprintf("%.2f", $historical_data[2]);
#my $whois = new Net::XWhois('Server' => 'whois.ripe.net');
</%init>
```

C.9 botstatus.mhtml

```
<%doc>
Component to display botnet activity on a specific day.
Activity is broken down per source IP address, and the
top 25 scored IP addresses are listed.

Takes two arguments:
```

```
date: a date string such as:
"2005-04-09" for 9th of April 2005. If not given,
the current date is used.

org: An optional organisation can be given to limit the
list of IP addresses to those of one organisation only.
</%doc>

<html>
<head>
<title>Daily botnet activity report</title>
<style type="text/css">
table { border: solid 1px; border-spacing: 0px; }
td,th { padding: 0.3em; text-align: left }
th { background-color: #bb8855 }
td.hilight { background-color: #ddff88 }
td.bar { background-color: #ddd }
</style>
</head>

<body>
<h2>Daily botnet activity report - <% $from %><% defined($org) ? " [
$org ]" : "" %></h2>

<p>At a glance:</p>
<ul>
<li><% $#{@$most_active} + 1 %> hosts seen (<% $#new_ips +1 %>
new)</li>
<li>Average score: <% sprintf "%.2f", $stats{'avg_score'} %></li>
<li><a
  href="https://stager.uninett.no/index.php?ss_db=stager&ss_tablelimit=20&ss_reportsty
</ul>

<h3>Top 25 high-risk hosts</h3>

<table>
  <tr>
    <th>Org</th><th>IP</th><th>Hostname</th><th>TCP SYN Count</th>
    <th>#Dst</th><th>First</th><th>Last</th><th>Score</th>
    <th>Status</th><th>First seen</th>
  </tr>
% my $i = 0;
% my $class = "foo";
% for my $entry (@$most_active) {
```

```
% last if $i == 24;
% $class = ($class eq "foo") ? "bar" : "foo";
  <tr>
    <td class="<% $class %>" style="text-align: right">
% if ($hilight{$entry->[0]}) {
  <span style="font-weight: bold; color: #449966">*NEW*</span>
% }
      <a href="#"><% $entry->[1] %></a></td>
    <td class="<% $class %>"><a href="ip_detail.mhtml?date=<% $from
    %>&ip=<% $entry->[0] %>"><% $entry->[0] %></a></td>
% my @h = gethostbyaddr(pack('C4',split('\.',$entry->[0])),2);
% my $hn = "";
% $hn = $h[0] if (@h);
    <td class="<% $class %>"><% $hn %></td>
    <td class="<% $class %>" align="right"><% $entry->[2] %></td>
    <td class="<% $class %>" align="right"><% $entry->[3] %></td>
    <td class="<% $class %>" align="right"><% $entry->[4] %></td>
    <td class="<% $class %>" align="right"><% $entry->[5] %></td>

% my $color = undef;
% if ($entry->[6] > 3) {
%  $color = $scorecolors{$entry->[6]};
% }
    <td class="<% $class %>" style="width: 9em; text-align: left;
      <% $color ? " color: $color; font-weight: bold;\"" : "\"" %>>
      <% '|' x $entry->[6] . " (" . $entry->[6] . ")" %>
    </td>
% my $alert_status = "Not reported";
% $alert_status = "<a href=\"alert.mhtml?id=" . $entry->[7] .
% "\">Alert #" . $entry->[7] . "</a>" if $entry->[7];
    <td class="<% $class %>"><% $alert_status %></td>
    <td class="<% $class %>"><% $old{$entry->[0]} %></td>
  </tr>
% $i++;
% }
</table>

<h3>Top 25 new IPs</h3>

<table>
  <tr>
    <th>Org</th><th>IP</th><th>Hostname</th><th>TCP SYN Count</th>
    <th>#Dst</th><th>Score</th><th>Status</th>
  </tr>
```

```
% $i = 0;
% for my $entry (@new_ips) {
% last if $i == 24;
% $class = ($class eq "foo") ? "bar" : "foo";
  <tr>
    <td class="<% $class %>"><a href="#"><% $entry->[1] %></a></td>
    <td class="<% $class %>"><% $entry->[0] %></td>
% my @h = gethostbyaddr(pack('C4',split('\.',$entry->[0])),2);
% my $hn = "";
% $hn = $h[0] if (@h);
    <td class="<% $class %>"><% $hn %></td>
    <td class="<% $class %>" align="right"><% $entry->[2] %></td>
    <td class="<% $class %>" align="right"><% $entry->[3] %></td>
    <td class="<% $class %>" align="right"><% $entry->[6] %></td>
% my $alert_status = "Not reported";
% $alert_status = "<a href=\"alert.mhtml?id=" . $entry->[7] .
% "\">Alert #" . $entry->[7] . "</a>" if $entry->[7];
    <td class="<% $class %>"><% $alert_status %></td>
    <td class="<% $class %>"><% $old{$entry->[0]} %></td>
  </tr>
% $i++;
% }
</table>

</body>
</html>



<%args>
  $date => undef,
  $org => undef
</%args>

<%init>

use Socket 'AF_INET';
use DBI;

my $from = $date;

if (not defined($from)) {
  my ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime();
  $from = sprintf "%4d-%02d-%02d 00:00", $year+1900,$mon+1,$mday;
}
```

```

my ($year, $month, $day) = split /-/, $from;
my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT src, org, packets, ips," .
          "first, last, score FROM period_data " .
          "WHERE first >= ? and last <= (? + reltime('1 day')) ";

if (defined($org)) {
  $sql .= " and org=? ";
}

$sql .= "ORDER BY score DESC";

my $most_active;

if (defined($org)) {
  $most_active = $dbh->selectall_arrayref($sql, undef, $from, $from, $org);
} else {
  $most_active = $dbh->selectall_arrayref($sql, undef, $from, $from);
}

$sql = "SELECT src, first FROM ip_seen WHERE first < ? and src NOTNULL";

my $old_ips = $dbh->selectall_arrayref($sql, undef, $from);

my %old = ();
my @new_ips = ();
my %hilight = ();
my %stats = ();

for my $entry (@$old_ips) {
  $old{$entry->[0]} = $entry->[1];
}

$sql = "SELECT alert_id " .
       "FROM ip_alerts " .
       "WHERE ip = ?";

my $sum = 0.0;
for my $entry (@$most_active) {
  $sum += $entry->[6];
  my ($ai) = $dbh->selectrow_array($sql, undef, $entry->[0]);
  push (@$entry, $ai);
```

```
    if (not ($old{$entry->[0]})) {
      push @new_ips, $entry;
      $hilight{$entry->[0]}++;
    }
  }


  $stats{'avg_score'} = $sum/($#{@$most_active} + 1);

  my %scorecolors = (
                      '4' => '#ff9999',
                      '5' => '#ff8888',
                      '6' => '#ff7777',
                      '7' => '#ff6666',
                      '8' => '#ff5555',
                      '9' => '#ff4444',
                      '10' => '#ff3333',
                      '11' => '#f22',
                      '12' => '#f11',
                      '13' => '#f00',
                      '14' => '#f00',
                      '15' => '#f00',
                      );
</%init>
```

C.10 alert.mhtml

```
<%doc>
Component to display all IP addresses reported in a given
alert. The time of alert and the e-mail address the alert
was sent to is also given.

Takes one argument:

id: The unique identifier of the alert.
</%doc>

<html>
<head>
<title>Daily botnet activity report - Alert detail</title>
<style type="text/css">
table { border: solid 1px; border-spacing: 0px; }
```

```
td,th { padding: 0.3em; text-align: left }
th { background-color: #bb8855 }
td.hilight { background-color: #ddff88 }
td.bar { background-color: #ddd }
</style>
</head>

<body>
<h2>Botnet activity report - Alert detail - #<% $id %></h2>

<ul>
<li>Sent <% $ts %> to <% $st %>.</li>
<li><% $#{@$alert_data} + 1 %> addresses reported in this alert.</li>
</ul>

<table>
  <tr>
    <th>IP address</th><th>Hostname<th>Score History</th>
  </tr>
% my $class = "foo";
% for my $entry (@$alert_data) {
% my @h = gethostbyaddr(pack('C4',split('\.', $entry->[0])),2);

% $class = ($class eq "foo") ? "bar" : "foo";
  <tr>
    <td class="<% $class %>"><% $entry->[0] %></td>
    <td class="<% $class %>"><% @h ? $h[0] : "" %></td>
    <td class="<% $class %>"><a href="plot_score_ip.mhtml?ip=<%
        $entry->[0] %>&full=1">
    <img style="border: none" src="plot_score_ip.mhtml?ip=<%
        $entry->[0] %>"/>
</a>
</td>
  </tr>
% }
</table>

</body>
</html>

<%args>
  $id
</%args>
```

```
<%init>

use DBI;

#my $from = $date;

#my ($year, $month, $day) = split /-/, $from;
my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT ip " .
          "FROM ip_alerts " .
          "WHERE alert_id = ? ";

my $alert_data = $dbh->selectall_arrayref($sql, undef, $id);

$sql = "SELECT sent_to, ts " .
          "FROM alerts " .
          "WHERE id = ? ";

my ($st, $ts) = $dbh->selectrow_array($sql, undef, $id);

</%init>
```

C.11 plot org history.mhtml

```
<%doc>
Component to plot the number of active and new source IP addresses for
a given organisation, from a given day, and one week back in time. The
output is a 200x70 pixel png image.

An optional argument, full, can be set to get a full size png image,
with full labels on the axis.
</%doc>

<%args>
  $date
  $org
  $full => 0
</%args>

<%init>
```

```
use DBI;

my $from = $date;

my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT day, cnt, new_cnt " .
          "FROM org_stats_day " .
          "WHERE day >= (? - reltime('7 days')) and day <= ?" .
          "and org = ? ORDER BY day";

my $most_active = $dbh->selectall_arrayref($sql, undef, $from, $from, $org);

my $graph;
if ($full) {
  use GD::Graph::linespoints;
  $graph = GD::Graph::linespoints->new(800,500);
  $graph->set(
             y_min_value       => 0,
             skip_undef        => 1,
             x_label_skip      => 7,
           ) or die $graph->error;
} else {
  use GD::Graph::lines;
  $graph = GD::Graph::lines->new(200,70);
  $graph->set(
             y_label_skip      => 2,
             x_label_skip      => 7,
             y_min_value       => 0,
             y_max_value       => 35,
             skip_undef        => 1,
           ) or die $graph->error;
}


my $xvalues = [];
my $yvalues = [];
my $y2values = [];

for my $row (@$most_active) {
  push(@$xvalues, $row->[0]);
  push(@$yvalues, $row->[1]);
  push(@$y2values, $row->[2]);
}
```

```
use Data::Dumper;

my @data = ($xvalues, $yvalues, $y2values);
my $gd = $graph->plot(\@data) or die $graph->error;

$r->content_type('image/png');
$r->send_http_header;
binmode(STDOUT);
print $gd->png();
</%init>
```

C.12 plot score ip.mhtml

```
<%doc>
Component to plot the score of a given source IP addresses, from a
given day, and backwards. The output is a 200x70 pixel png image.

An optional argument, full, can be set to get a full size png image,
with full labels on the axis.

An optional argument, avg, can be set to get a the average score
overlaid on the plot.
</%doc>

<%args>
  $stop => 'now'
  $ip
  $full => 0
  $avg => 0
</%args>

<%init>

use DBI;

my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT first::date, score " .
          "FROM period_data " .
          "WHERE last <= (? + reltime('1 day'))" .
          "and src = ? order by first";
```

```
my $most_active = $dbh->selectall_arrayref($sql, undef, $stop, $ip);

my $graph;
if ($full) {
  use GD::Graph::linespoints;
  $graph = GD::Graph::linespoints->new(800,500);
  $graph->set(
              y_min_value       => -2,
              y_max_value       => 15,
              skip_undef        => 1,
            ) or die $graph->error;
} else {
  use GD::Graph::lines;
  $graph = GD::Graph::lines->new(200,70);
  $graph->set(
              y_label_skip      => 1,
              x_label_skip      => 7,
              y_min_value       => -2,
              y_max_value       => 15,
              skip_undef        => 1,
            ) or die $graph->error;
}


my $xvalues = [];
my $yvalues = [];
my $y2values = [];

for my $row (@$most_active) {
  push(@$xvalues, $row->[0]);
  push(@$yvalues, $row->[1]);
  if ($avg) {
    push(@$y2values, $avg);
  }
}

my @data = ($xvalues, $yvalues, $y2values);
my $gd = $graph->plot(\@data) or die $graph->error;

$r->content_type('image/png');
$r->send_http_header;
binmode(STDOUT);
print $gd->png();
```

```
</%init>
```

C.13 plot controllers history.mhtml

```
<%doc>
Component to plot the number of active controllers, from a
given day, and backwards. The output is a 200x70 pixel png image.

An optional argument, full, can be set to get a full size png image,
with full labels on the axis.

An optional argument, new, can be set to get a plot of the new
controllers instead.
</%doc>

<%args>
  $date
  $new => 0
  $full => 0
</%args>

<%init>

use DBI;

my $from = $date;

my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT day, cnt, new_cnt, reactive_cnt " .
          "FROM controller_stats_day " .
          "WHERE day >= (? - reltime('7 days')) and day <= ?" .
          " ORDER BY day";

my $most_active = $dbh->selectall_arrayref($sql, undef, $from, $from);

my $graph;
my $graph2;
if ($full) {
  use GD::Graph::linespoints;
  $graph = GD::Graph::linespoints->new(800,500);
  $graph2 = GD::Graph::linespoints->new(800,500);
  $graph->set(
              y_min_value       => 0,
              skip_undef        => 1,
            ) or die $graph->error;
  $graph2->set(
              y_min_value       => 0,
              skip_undef        => 1,
            ) or die $graph2->error;
} else {
  use GD::Graph::lines;
  $graph = GD::Graph::lines->new(200,70);
  $graph2 = GD::Graph::lines->new(200,70);
  $graph->set(
              y_label_skip      => 2,
              x_label_skip      => 7,
              y_min_value       => 0,
              y_max_value       => 850,
              skip_undef        => 1,
            ) or die $graph->error;
  $graph2->set(
              y_label_skip      => 2,
              x_label_skip      => 7,
              y_min_value       => 0,
              y_max_value       => 50,
              skip_undef        => 1,
            ) or die $graph2->error;
}


my $xvalues = [];
my $yvalues = [];
my $y2values = [];
my $y3values = [];

for my $row (@$most_active) {
  push(@$xvalues, $row->[0]);
  push(@$yvalues, $row->[1]);
  push(@$y2values, $row->[2]);
  push(@$y3values, $row->[3]);
}

use Data::Dumper;

my @data = ($xvalues, $yvalues);
my @data2 = ($xvalues, $y2values, $y3values);
my $gd = $graph->plot(\@data) or die $graph->error;
my $gd2 = $graph2->plot(\@data2) or die $graph2->error;
$r->content_type('image/png');
$r->send_http_header;
binmode(STDOUT);
unless ($new) {
print $gd->png();
} else {
print $gd2->png();
}
</%init>
```

C.14 plot pie chart.mhtml

```
<%doc>
Component to plot the used destination port numbers for a given
source IP address, from one point (from) in time to another (to).
The output is a 200x70 pixel png image.

An optional argument, full, can be set to get a full size png image,
with full labels on the axis.

An optional argument, whole, can be set to get the graph for the
entire period of activity for the source IP address.
</%doc>

<%args>
  $ip
  $from => undef
  $to => undef
  $whole => 0
  $full => 0
</%args>

<%init>

use DBI;
use GD::Graph::pie;

$to = $from if (not $to);
my $most_active;
my $dsn = 'dbi:Pg:dbname=boned;host=127.0.0.1;port=8888';
my $dbh =  DBI->connect($dsn, 'boned', 'XXXXX') or die $DBI::errstr;
my $sql = "SELECT dstport, count(distinct dst) " .
          "FROM packets ";

if ($whole) {
  $sql .= "WHERE src = ? and tcp_flags=? GROUP BY dstport";
  $most_active = $dbh->selectall_arrayref($sql, undef, $ip, 2);
}

else {
  if ($to eq $from) {
    $sql .= "WHERE ts >= ? and ts <= ? + reltime('1 day') ";
  } else {
    $sql .= "WHERE (ts between ? and ?) ";
  }

  $sql .= "and src = ? and tcp_flags = ? GROUP BY dstport";
  DBI->trace(2);
  $most_active = $dbh->selectall_arrayref($sql, undef, $from, $to,
  $ip, 2);
}

my $graph;
if ($full) {
  $graph = GD::Graph::pie->new(800,500);
} else {
  $graph = GD::Graph::pie->new(75,70);
  $graph->set('3d' => 0) or die $graph->error;
}

my $xvalues = [];
my $yvalues = [];

for my $row (@$most_active) {
  push(@$xvalues, $row->[0]);
  push(@$yvalues, $row->[1]);
}

my @data = ($xvalues, $yvalues);
my $gd = $graph->plot(\@data) or die $graph->error;

$r->content_type('image/png');
```

```
$r->send_http_header;
binmode(STDOUT);
print $gd->png();
</%init>
```