

# Ourmon and Botnet Detection

James R. Binkley  
Computer Science Dept.  
Portland State University  
Portland, OR, USA  
*jrb@cs.pdx.edu*

December 30, 2005

## Abstract

*Ourmon* is a near real-time network monitoring and anomaly detection system that captures packets using port-mirroring on Ethernet switches. It displays data on the web as graphics or ASCII reports. In this paper we first briefly introduce *Ourmon* and architectural components relevant to botnet detection. Then we present its current *anomaly-based* botnet detection algorithm which is based on a combination of simple statistics, detection of TCP scanners using a heuristic called the TCP work weight, and a fast Layer 7 IRC message tokenizer. *Ourmon*'s current implementation has detected three instances of botnet servers in the last year on our campus including one server with over 60000 off-campus hosts. It has also detected multiple instances of small botnet client meshes reporting to off-campus botservers. We will then present a brief discussion of improvements aimed at commercialization. These improvements include a new algorithm to detect encrypted L7 network meshes (encrypted botnets), as well as optimizations to the *Ourmon* front-end probe aimed at improving its packet per second performance.

Please note: ideas in this document are considered to be the intellectual property of PSU and should not be disclosed to commercial interests without permission.

## 1 Introduction

*Ourmon* [1] is a network monitoring and anomaly detection tool. Originally it was intended as an open-source system somewhat akin to SNMP RMON [11] with the goal of providing web-based graphics and reports about current network statistics. Over the last few years it has evolved into a network security tool that provides information about important network anomalies including TCP and UDP scanners, worms, denial of service attacks, and IRC botnets.

*Ourmon* has two conceptual parts, a front-end probe which sniffs data from an Ethernet switch and samples packets for thirty-second periods, and a back-end graphics display and logging system. The entire system uses various filters that in general capture integer counts or list tuples of various kinds (roughly similar to flows). However our version of "flow data" allows for funda-

mentally different kinds of tuples which include traditional flows, but also provide other kinds of keys for lists including IRC channel names, or IP source addresses. Back-end data is displayed on the web, and consists of various kinds of graphs and ASCII reports including data for the last 30-second sample period. The back-end also summarizes 30-second data over longer periods thus providing hourly reports, daily reports, and in some cases yearly summarizations.

The current Ourmon system is open-source and is available on Sourceforge[10]. It is also available at PSU where much of the statistics for our DMZ may be seen live[7]. Architectural details and user information are available on the web as well at: <http://ourmon.cat.pdx.edu/ourmon/info.html>.

In this white paper we will first provide a short introduction to the Ourmon architecture that focuses on salient features used for botnet detection and neglects its more general network monitoring and anomaly detection features. We then will discuss how our current open-source system can easily detect various kinds of botnets including client botnets where the server is off-campus, and server botnets where the server is on-campus. We also provide a short section that discusses various commercially-oriented improvements including a proposed algorithm that will allow us to detect encrypted L7 botnets.

## 2 Ourmon Architecture

Ourmon is divided up architecturally into two sub-systems, a front-end probe and a back-end data processing engine which may be combined on a single computer but for efficiency and security reasons should ideally run on two computers. The probe is placed in a network DMZ and

gathers packets from an Ethernet switch that is programmed to capture all packets passing through its other interfaces. Thus we can use the probe to watch a set of servers or the Internet ingress/egress point for a network. Front-end data is typically condensed (we do not send all flows, but instead aggregate data) and if the two-system design is used, data may be shipped to the back-end system typically via secure shell or wget. Ideally the back-end system is not a bastion host like the probe and is protected in some sort of secure enclave.

Front-end data consists of a small set of ASCII files which are copied at the end of the thirty second sample period to the back-end. The file format is internal to the Ourmon system and is not standardized in any way but may change from release to release. This allows us to rapidly customize and change Ourmon as the system evolves over time. The back-end system both logs and processes the data producing hourly reports, web graphics, and other forms of output. The back-end system is expected to run an open-source web server such as Apache.

Ourmon is a near real-time system, as the worst case delay in front-end to back-end processing is never more than one minute. The back-end (using UNIX cron) produces data that reflects the (near) current situation for the network, and also summarizes data on the hour during the current day. Depending on the data type, the back-end also produces daily reports (typically for tuples) or RRDTOOL [9] graphics that can span a year.

The front-end implements a set of configurable filters (they may be turned on or off) that can be loosely divided into three basic major types. Filters consist either of list tuples that are often treated as top N lists (for example, we may want only the top 100 tuples in terms of activity),

Berkeley Packet Filter [5] expressions that may be grouped together into a composite back-end RRDTOOL graph or certain hardwired scalar filters that are also displayed as RRDTOOL graphics.

RRDTOOL graphs baseline a year's worth of data, and come in four time formats per filter, current day, weekly, monthly, and yearly. BPF filters and hardwired scalar filters both produce small sets of integer counters bound to a user-supplied (BPF) or hardwired (scalar) filter name. For the purpose of this paper, we can neglect the user-programmable BPF mechanism as it is primarily useful in terms of network management and has little role to play in our discussion of botnet detection. As a result we will focus on the other two kinds of filters in the front-end: *list-based filters* (where memory is dynamically allocated), and *scalar filters* that consist of a small set of scalar integer counters.

*Scalar filters* are graphed in the back-end using RRDTOOL stripcharts and have a few integer counters per graph. Ourmon uses RRDTOOL to construct graphs based on various individual integer counters. See figure 1 at the end of this paper for one example of an BPF/RRDTOOL graph. This particular graph shows total IRC message counts in our DMZ for the four fundamental IRC message types parsed by the front-end IRC tokenizer, JOIN, PING, PONG, and PRIVMSG IRC commands.<sup>1</sup> The graph illustrated is a weekly graph and illustrates the effect an on-campus botserver had on basic message counts during the November 2005 thanksgiving weekend. We will discuss this graph more below.

*List-based filters* are an innovative part of the

---

<sup>1</sup>Our campus network has over ten thousand live switch ports, and peak traffic runs around 200 megabits total to and from the Internet.

Ourmon system. One of Ourmon's design goals is to allow the construction of first-order tuples with unique keys as opposed to second-order parsing of traditional flow data. Thus our front-end can aggregate data and we do not need to store all flows. Where possible the high-level goal is to point out the big picture as opposed to swamping an analyst in unnecessary details. Another difference with traditional flow storage is that we are willing to look at Layer 7 payloads as this is very necessary for determining P2P data, IRC channel names, and message types. List-based filters currently include traditional flow tuples with a 5-field index, top talker L4 TCP and UDP ports with ports as an index, top talker scanners with IP source addresses as an index, and four kinds of tuples that are of interest to botnet detection including the following:

1. *the TCP syn tuple* - this tuple is per IP host and counts TCP control packets as well as other counters. It also includes a set of sampled destination ports which is not of interest here, but is very useful for helping an analyst get the big picture for TCP-based scanning and worm detection. We will discuss this tuple more below as it is important to our anomaly-based botnet detection scheme.
2. *the UDP/ICMP error tuple* - this tuple is similar in design to the TCP syn tuple and for the most part focuses on UDP scanners. It is also keyed on an IP host. It is not currently used for botnet detection, but will play a role in our future efforts aimed at encrypted mesh detection. We will not discuss this tuple in more detail - if desired please see [8] for more information.
3. *the IRC channel tuple* - this tuple is keyed

to an IRC channel name, and collects a list of IP sources (hosts) associated with that channel, as well as basic message counts for IRC messages associated with the channel. As a result we can collect a complete set of IRC channel names.

4. the IRC host tuple - this tuple compute various statistics for IP hosts found in IRC channels. Data includes counters for IRC message types, and also a very valuable piece of information extracted from the TCP syn tuple for the host in question. This heuristic attribute is a weight called the *TCP work weight* and informally tells us whether or not a host has been acting as a TCP scanner.

In summary, our current botnet detection system uses the TCP syn, IRC channel, and IRC host list tuples. We also intend to implement a new sub-system that will look for "stealthy meshes" by which we mean a Layer 7 mesh that may be using Layer 7 encryption. This new system will use the UDP/ICMP error tuple as well.

In the next section we will look at our current botnet detection system in more detail and also provide examples of how Ourmon has detected botnets on our campus.

### 3 Current Ourmon Botnet Architecture

In this section we first discuss some important background assumptions in terms of IRC and botnets and then proceed to explain our current Ourmon IRC detection architecture used for detecting IRC-based botnets. We then show how our system has worked in the last year with two examples of captured attacks from fall of 2005.

One example is of a large botnet server detected on campus. The other is of a more modest client botnet detected in our dormitories. We first deployed our IRC measurement system in the spring of 2005. Since that period we have seen numerous examples of small botnets with servers off-campus, and three examples of large botnets with the botnet server located on campus.

We assume the reader is familiar with the IRC architecture, although in point of fact IRC is a very loose phenomenon and there is no guarantee that any bot network using IRC necessarily strictly conforms to IETF notions of IRC as a protocol. Please see the honeynet project [4] for further information on botnets. For further IRC information please see wikipedia [12] as well as the fundamental IRC RFC [6].

Before we present our architecture, there are a number of points that need clarification. First of all we do not rely on ports for capture of any IRC information. For example, it cannot be assumed that botnet control plane IRC traffic uses the traditional IRC server destination port of 6667. This is certainly the case with botnets that we have observed as any port may be in use at any time and may change at any time. Our IRC analysis software does not currently use ports in any way.

When we use the conventional term *botnet* we have not said whether a botnet is good or bad. Botnets seem to have originated from IRC-based command subsystems that were used for file copying (of so-called "warez" meaning illegotten files, which may include data including audio or video recordings or pirated software). Botnets may be games, reminder systems, or other relatively benign systems that attempt to fool people on conventional IRC chat channels into conversations with automata.

For the rest of our discussion, when we re-

fer to a *botserver mesh*, we are talking about an instance of a "evil" botserver on our university network. Note that typically an on-campus botserver will have thousands of off-campus captive hosts in a few IRC channels. The botserver meshes we have seen vary from 30000 hosts to 60000 hosts. Normal IRC servers used for chat on our campus never have more than one hundred hosts and usually have far less hosts in any channel. When we refer to a *botclient mesh*, we mean a set of communicating hosts with the botserver off campus. Examples of this phenomenon as seen on our campus have a few tens of captive hosts at most.

How then do we know if a botserver or botclient mesh is malign? There are two ways. First within our own Ourmon system, we know that a botclient mesh is malign by observing the number of anomalous TCP-scanning hosts in a given IRC channel. Roughly if an IRC channel called "F7" has ten hosts, with nine clients and one external server, and a majority of the IRC client hosts have been spotted during any period of the day performing TCP SYN scanning we can assume that the mesh in question is malign.<sup>2</sup>

The second mechanism for determination of "badness" is to simply extract the IRC channel server or server addresses from our Ourmon reports, and use a well-known tool like "ngrep" to investigate particular IRC hosts or channels for IRC messages. Fundamentally an analyst can observe the message traffic and decide if a malign botnet is active. In addition, one can ob-

serve botnet messages that may be familiar or unknown to the analyst and learn what kinds of botnet messages may exist. For example, one may see message traffic like "exploited ip" or "lsass scan" commands. We originally used ngrep daily over a period of six months (winter 2005 to summer 2005) to gain confidence in our Ourmon system. At this point we do not feel the need to use it any more to prove that a botnet mesh is "evil" and feel that we can completely rely on Ourmon data. Of course ngrep is still useful for forensic reasons to learn more about the current message set used by botnets. This information can prove useful in terms of forensic-based cleanup and analysis of infected hosts. It is quite reasonable to regard ngrep and Ourmon as companion tools. Ourmon provides the big picture, pointing out the IP addresses in a bot mesh, and ngrep can provide details about the botnet code and attacks.

We should further point out that we live on a busy university network and IRC is popular on campus for conventional reasons. As a consequence, we believe that we have a very good basis for observing both bad and "normal" IRC. In this paper, we do not choose to define the use of IRC for multimedia file exchange as a security problem, although others might have differing opinions.

As mentioned previously IRC is a very fuzzy protocol. Our IRC network extraction system is based on parsing L7 payload data and as a result tries to focus on statistics for as few IRC message types as possible including JOIN, PRIVMSG, and PING and PONG. We believe that these messages are typical of IRC in general and that various forms of IRC software conform with a high degree of generality in how these message types are used. JOIN and PRIVMSG both provide channel names. PRIVMSG is used

---

<sup>2</sup>Note that a botclient mesh is harder to spot than a botserver mesh. Botserver meshes are in fact trivial to detect simply because their IRC message statistics are completely outlandish when compared to normal IRC channels. This is true in terms of message counts and other anomalies seem to exist as well. We will return to this subject below.

for data payload exchange (chat traffic or bot commands). JOIN, PING, and PONG are all used to keep IRC meshes fresh in terms of client-server state. Some meshes may use JOIN alone to keep the net fresh. Some may rely more on PING and PONG. Thus we define IRC in a very restricted sense (four message types, with JOIN and PRIVMSG providing channel names). The implicit goal is to use a relatively simple mechanism to capture IRC traffic, illicit or otherwise.

We observe that botnet detection is not principally a "real-time" or near real-time phenomenon. On our campus normal IRC traffic is slow and is driven by human chat. This is also generally true of botnets as well, although benign automated channels may exhibit higher traffic levels than normal human chat channels. Even the control plane of IRC consisting of JOIN, PING, and PONG messages may be as slow as one message per minute. Thus one of the principle tenets of our system, is that while we do gather data every thirty seconds, we perform hourly analysis of IRC data during the current day. Thirty-second analysis will show which IRC nets are alive "now", but the hourly analysis is more useful and can thus deal with the relatively slow phenomenon of IRC traffic. Another reason for hourly analysis is that botnet scanning is turned on or off by remote commands, and an analyst who only observes "now" can easily miss the activation of a scanning network or host. Thus we believe it is crucial to analyze IRC data over a longer period of time (hours and/or days). As a result, in addition to hourly reports during the current day, Ourmon also provides a week worth of summarized IRC reports (which are summarized at midnight of the current day and rolled over to the previous day). An analyst can both study the current day's data and may also use previous days in the week to study the

history of a channel of interest.

One of the problems with botnets which has not been alluded to elsewhere, is that hosts using P2P programs like Gnutella may at times appear to be TCP scanners.<sup>3</sup> In general one can observe that P2P software has two architectural tenets: 1. a host must somehow keep a list of IP peers, and success in P2P terms involves keeping that list (or cache) fresh, and 2. given the set of cached IP peers one tries to contact them (in parallel) using TCP to exchange data. If the set of cached peers are unreachable for some reason (possibly due to a stale IP cache), an unsuccessful P2P client may make a host appear as a TCP scanner. Because our botnet detection system relies fundamentally on TCP scanner detection, it can occasionally become important for us to determine if an individual botnet client host is a P2P-based host. Ultimately P2P determination helps us in some cases to rule out false positives. As a consequence Ourmon can detect some forms of P2P and anomaly-based P2P detection is an area of active research for us. We will not say much more about this topic in this paper, but we do want to point out that P2P detection and botnet detection are not completely unrelated.

### 3.1 Current IRC Botnet Detection Architecture

In this section we will present the current Ourmon architecture for botnet detection. We point out that our system is not signature-based in any way – it does not use ports or known botnet command strings. It is completely anomaly-based.

Our architecture principally relies on the observation that IRC hosts are grouped into chan-

---

<sup>3</sup>Possibly because our approach to botnet detection is anomaly-based and also because our Ourmon system was born on a university network where P2P is popular.

nels by a channel name (for example, "F7", or "ubuntu" might be channel names), and that an evil channel is an IRC channel with a majority of hosts performing TCP SYN scanning. This allows us to detect botclient meshes and to a lesser extent even seems to work for botserver meshes.

Additional simple but important anomalies exist for botserver meshes including the count of IP hosts in a channel, and the average number of PINGS/PONGS per time period for a channel. Counting IP hosts per channel can be viewed as the fundamental way to detect a botserver mesh. When compared to ordinary chat IRC channels, hosts per channel, PING/PONG counts, and even basic messages per server, typically go off scale with on-campus botservers due to the size of the meshes (per channel host counts) involved. It should be pointed out that it is always possible that an IRC botnet server might infect on-campus hosts and thus all the IRC measures mentioned above can be important.

The front-end Ourmon probe gathers three kinds of list tuples that are useful for IRC and botnet detection. The tuples consist of two kinds of IRC tuples and the TCP syn tuple. The probe gathers these tuples over its thirty second sampling period. All IRC tuples are sent to the back-end for further processing. In the probe, the entire campus TCP syn tuple is filtered into a smaller subset which informally consists of hosts observed sending anomalous amounts of TCP syns. This syn tuple subset is called the "worm set" and is typically orders of magnitude smaller than the entire set of IP sources found in the campus TCP syn set. It can be regarded as the set of TCP scanners.

The TCP syn scanner list tuple has the following rough form:

(IP source address, SYNS, SYNACKS,

FINSENT, FINSBACK, RESETS, ICMP ERRORS, PKTSSENT, PKTSBACK, APPFLAGS, port signature data)

The logical key in this tuple is an IP source address. SYNS, FINS (all kinds), and RESETS are counts of TCP control packets. SYNS are counts of SYN packets sent from the IP source, and SYNACKS are a subset of only those SYNS sent with the ACK flag set. FINS sent both ways are counted. RESETS are counted when sent back to the IP source. ICMP ERRORS refers to certain ICMP errors like unreachable or TTL errors returned by receivers. The PKTSSENT counts the total packets sent by the IP source. PKTSBACK counts the total pkts returned to the IP source. APPFLAGS is signature based and shows whether the host was performing IRC, or P2P traffic based on a few popular P2P protocols including BitTorrent, Edonkey, and Gnutella. There is also a small fixed set of sampled TCP destination ports that we will neglect here.

This information is useful for determining what kind of scanning is occurring and often gives a rough network-based indication of the kind of exploit in use.<sup>4</sup>

We define a metric which we call the *TCP work weight*. The *work weight* is easy to compute and is computed by the probe per IP source as follows:

$$w = (S_s + F_s + R_r)/T_{sr}$$

It is expressed as a percent. The rough idea is that we take the count of TCP control packets and divide that count by the total number of

---

<sup>4</sup>It can be useful to note when a host is scanning ports 445 or 139, as such scanning is often part of the botnet phenomenon.

TCP packets. Obviously 100% here is a bad sign and implies a true anomaly of some sort. Such a value is typically associated with a scanner or worm. The IRC module in the probe uses the TCP list as an underlying "tool", and extracts the TCP work weight from it for any IRC host.

We should point out that we have over two years worth of experience with the work weight at this point. We have learned that high work weights with hosts are caused by three possible causes including 1. scanners (typically syn scanners), 2. clients lacking a server for some reason or 3. P2P hosts (usually Gnutella is the application) which cannot for some reason find a fresh set of IP peers. In general scanners are the most common reason for a high work weight. We also know that the work weight clusters into either high values (say 70..100%) or low values (say 0..30%). Attackers fall into the higher range. P2P clients on average fall into the lower range.

Typically the average over many samples is of interest. However in the case of IRC we decided to simply take the maximum work weight seen over all the thirty second samples for a day. This is because an otherwise normal host may be ordered remotely to do scanning for a short period of the day. One host by itself in an IRC channel with a high work weight may not be anomalous. However if a channel has ten hosts out of twelve with high work weights suspicion is justified. As a result, work weights associated with IRC channels in our summarization reports are maximum weights seen across all the samples in a daily report.

There are two IRC lists, called the *channel list* and the *node list*. The channel list has the following tuple structure:

```
(CHANNAME, HITS, JOINS, PRIVMSGS,
NOIPS, IP_LIST)
```

The channel name is the case-insensitive IRC channel name extracted from JOIN and PRIVMSG IRC messages by the IRC scanner. The probe's scanner is hand-crafted C code that looks at the first 256 bytes of the L7 payload for TCP messages only and extracts IRC tokens for the four kinds of messages of interest. HITS is the total count of JOINS and PRIVMSGS, JOINS and PRIVMSGS are counts of that particular kind of message. NOIPS is the number of IP addresses in the IP\_LIST, which follows the tuple. Thus a channel tuple gives a key (the channel name) with a few message count statistics and a list of IRC hosts in the channel expressed as IP addresses.

The *node list* gives per IP statistics for any IP address in any IRC channel. Informally a channel may be viewed as a directory, and a host may be viewed as a directory entry (although a host may actually be in more than one channel). The node list has the following tuple structure (not all counters shown):

```
(IPSRC, TOTALMSG, JOINS, PINGS, PONGS,
PRIVMSGS, CHANNELS, SERVERHITS, WW)
```

The key per tuple is an IP source address. Various message statistics are given including JOIN, PING, PONG, and PRIVMSG counts. The number of observed per host channels is supplied. SERVERHITS indicates the number of messages sent to/from a host. Thus this counter indicates whether a host is acting as an IRC server. The WW (work weight) as mentioned previously is derived from the TCP syn module.

One additional IRC statistic is gathered by the front-end which consists of total counts of the four kinds of IRC messages seen by the probe



during the sample period. This tuple is displayed by the back-end as an RRDTOOL-based graph and we will see an example of it below. Note that its values are for the network as a whole and are not per channel or host.

The IRC tuples are passed to the backend for report generation. The backend program produces an hourly text report (updated on the hour) which is called *ircreport\_today.txt*. This file is available on the web for analysis. Data in this report is broken up into three major sections including global counts, channel statistics, and per host statistics. Channel statistics and per host statistical sections are further broken up into various sub-reports where data is typically sorted by some key statistic.

We can distinguish the following IRC report sub-sections:

1. evil channels - channels with too many hosts with a high work weight
2. channels sorted by maximum messages.
3. channels with host statistics - each channel shows the host IP in the channel with host stats.
4. servers sorted by max messages - hosts that are IRC servers are sorted by max messages.
5. hosts with join messages but no privmsgs - JOINS only but no data payloads.
6. hosts with any signs of worminess - hosts with high work weights.

For purposes of illustration in table 1 we look at one benign example which comes from the per channel host statistics section. Counts given for our example were taken from twelve hours of

data (since midnight) and are typical for a small IRC chat group.<sup>5</sup>

In our example 1, a channel named "ubuntu" has four hosts in it. Three are local and the server (S) is remote. Total message counts (of the 4 kinds parsed) and JOIN, PING, PONG, and PRIVMSG counts are given. Maxchans is the number of channels seen during the period for that host, and maxworm is the maximum work weight seen. We do not believe this channel based on the above data is "evil".

Now let us see how this data may be correlated to plainly point out anomalous IRC-based botnet behavior.

### 3.2 Botnet Examples

Let us look first at the relatively simple case of a botclient mesh. The server is off-campus and a few hosts have been captured on-campus to become part of the botnet. We look at two sub-sections of the hourly IRC report to find our evil channel which is named "F7". We look at our evil channel sort, and discover that F7 shown in table 2 is named as a channel in that list and occupies a high rank in the list.

Channel F7 is high in the evil channel list simply because it has 4 out of 6 hosts with high work weights. The "evil" flag at the end of the column is set to E if a potential evil channel has more than 1 anomalous host. Next we look at the report sub-section which breaks host statistics out for the channel F7.

In table 3 we see the part of the report that shows hosts in a channel. In channel F7, we

---

<sup>5</sup>All IP addresses have been changed and are represented as symbolic addresses. In addition the reader should note that our current output format is simply an ASCII report. However we represent it here in tabular format.

Table 1: Benign IRC Channel - Channel/Host Report

channel/ip	tmsg	tjoin	tping	tpong	tprivmsg	maxchans	maxworm	server
ubuntu/net1.host1	11598	1282	1912	1910	6494	4	43	H
ubuntu/net1.host2	7265	938	619	622	5086	3	0	H
ubuntu/net1.host3	17218	1926	4123	4100	7069	5	37	H
ubuntu/net2.host1	28152	3222	3913	3904	17113	8	0	S

Table 2: Malign IRC Client Botnet - Evil Channel Report

channel	msgs	joins	privmsgs	ipcount	wormyhosts	evil
F7	118	19	99	6	4	E

Table 3: Malign Botnet Channel F7 - Channel/Host Report

channel/ip	tmsg	tjoin	tping	tpong	tprivmsg	maxchans	maxworm	server
F7/net1.1	1205	24	377	376	428	2	42	H
F7/net1.2	113	6	39	43	25	1	96	H
F7/net1.3	144	2	60	61	21	1	94	H
F7/net1.4	46	3	12	14	17	1	90	H
F7/net1.5	701	2	343	345	11	1	90	H
F7/net2.1	1300	19	587	593	101	1	16	S

have one remote server and five infected local hosts. Four of those hosts have very high maximum work weights. We know from experience with the work weight (and also by looking at logs from both Ourmon and other systems) that the hosts are performing SYN scanning. Ourmon logs for the syn tuple will typically show that the hosts in question have been performing scanning aimed at Microsoft exploits on port 445 (typically lsass-based exploits, for example, see [2]).

We have used ngrep in the past to prove beyond a shadow of a doubt that examples like our F7 botnet client are indeed malign. At this point in time, we no longer feel the need to use a tool like ngrep to prove that ourmon has detected an evil mesh. However the reader might desire to see such proof and in addition ngrep can still be very useful to aid in host forensics. For example, one may be able to gather valuable clues about the exploit used. An ngrep sent from a local client to the server in question (net2.1) showed messages like the following:

```
# ngrep -q host net2.1
T net1.1:1053 -> net2.1:30591 [AP]^
  PRIVMSG #F7 :[Lsass]: Fuxed IP: net1.2
```

Here we see a report from a bot client back to the server that host net1.2 has been exploited. The exploit used is also mentioned.

Next we turn to botserver mesh detection. In point of fact, detecting a botserver in a network domain is fairly trivial once you know the "signs". We can distinguish at least four kinds of anomalies and there are probably more simply due to the large amounts of IRC traffic. The main fact to note is that a "successful" botserver will produce many messages. Over thanksgiving 2005 (when IT staff were on vacation), our campus experienced the largest botserver we have

ever seen. We believe that the botserver in question at its peak had around 60000 hosts (all off campus) reporting to it. We should point out that as is generally the case with anomaly detection, the analyst should have some notion of normal in terms of local domain statistics. In this case one has to have observed local IRC activity to know what is unusual. Even so, a bot server is easy to spot.

Outstanding anomalies included:

1. the number of hosts per channel is a very strong indicator of a botserver.
2. the number of messages for any given IRC server is a strong indicator.
3. the overall basic IRC message counts for our network domain as a whole are a strong indicator.
4. lastly it can sometimes be the case that the overall number of wormy hosts (high work weights) counted in our entire network (and within the bot server's IRC channel) becomes very high.

We do not currently have a sort for hosts per channel but we note anomalies one, two, and four in our list above by looking at our "channels sorted by max messages" subsection as seen in table 4.

From experience we know that "studentchannel" is benign (although automated in terms of both JOINS and PRIVMSGs), and in any case has a historically normal count of messages and hosts. On the other hand, channels f, x, and f-exp are new. The message count for f alone is historically very high and an order of magnitude higher when compared to "studentchannel". The above statistics are for the entire day,

Table 4: Malign IRC Botnet Server - Max Messages in Channels Report

channel	msgs	joins	privmsgs	ipcount	wormyhosts	evil
f	157403	156956	447	36727	1704	E
x	81161	40196	40965	13821	712	E
f-exp	20845	0	20845	5074	562	E
studentchannel	11560	5509	6051	12	0	

and busy chat channels on our campus driven by human text messages usually have no more than 2000 messages in one day. Therefore channel x and its 40965 PRIVMSGs are an anomaly. A more significant anomaly appears with the ipcount (number of hosts in the channel) for the three channels in question. On our campus we have never seen a normal chat IRC channel with more than 100 hosts in it. Of course, further analysis shows that these numbers are even more significant because the three channels in question are shown elsewhere to share the same botnet server. The number of hosts with high work weights in the channels is also an interesting anomaly.

In table 5 these three channels along with a fourth related channel called "exp" all appeared at the top of our evil channel subsection on the first day of the botnet server.

We frankly do not completely understand why botservers tend to appear in the evil channel list. Given that we know that one local campus host is acting as the server for the channel, this means that remote hosts are sending anomalous amounts of TCP syns to our campus. It is of course possible that those hosts are attacking our campus, but data from the TCP syn report shows that those hosts are trying to contact only the server itself. We suspect this is because the botserver has somehow stretched its own host resources to the limit, and simply

cannot respond to any other hosts that are trying to join the botnet. Thus these remote hosts appear as SYN scanners, but actually are botnet client hosts trying to connect over and over again to their server. Figure 2 shows our so-called "worm graph" which in summary graphs the total number of scanners dividing them two sets, off-campus (red) and on-campus (green). In this case the number of hosts with high work weights seems to be a side effect of the botserver mesh "at work". The number of scanners is of course less than the total number of clients, because most clients are connected.

Let us turn to anomaly number three. In a previous section we mentioned that we keep global count statistics for all IRC messages on our network and graph them using the RRD-TOOL graph mechanism. Please refer to Figure 1. This graph shows a weekly view of the four basic IRC message types. Normal daily counts for all message types are typically in the hundreds. Here we can see that on Friday and Saturday PING and PONG messages suddenly are in the thousands. In all three bot servers we have seen on campus, PING and PONG messages were always elevated. In one case, PRIVMSG was elevated as well. All bot servers so far have been enough to significantly perturb this graph. This is easy to understand because the PSU campus as a whole at peak periods sans a botserver has only a couple of hundred IRC hosts total.

Table 5: Malign IRC Botnet Server - Evil Channel Report

channel	msgs	joins	privmsgs	ipcount	wormyhosts	evil
f	157403	156956	447	36727	1704	E
x	81161	40196	40965	13821	712	E
f-exp	20845	0	20845	5074	562	E
exp	6825	0	6825	423	101	E

Because of our experience with bot servers, we are in the process of making some short-term and simple modifications to Ourmon to improve its ability to point out bot server related anomalies. Ourmon has an event log which in theory points out significant security-related events including large outburst of UDP packets, and large counts of hosts with high work weights. We have added an IRC-related event that reports an IRC channel with a large number of hosts. We also intend to add another RRDTool related graph that will simply graph the largest number of hosts in any IRC channel every thirty seconds, and a new sort sub-section to the IRC report which will sort by channels with the biggest number of hosts.

## 4 Proposed Commercial Improvements

In this section we will present a brief discussion of intended commercial improvements to Ourmon. We should point out that our open source code including the current botnet detection mechanism has been available to the world since September 2005 and is currently findable on Sourceforge as well as at PSU. Furthermore, the front-end of Ourmon written in C is stable, and neither leaks memory space or crashes. The back-end written in perl needs to run to completion within its time allotted but given that the front-end minimizes the data sent to the back-

end, this is easily achieved. Ourmon works well as long as it is deployed on a P4 class platform.

In this section, we will discuss improvements we would make to Ourmon that will both improve its ability to detect stealthy meshes and in general approve Ourmon’s appearance and functionality in terms of commercialization.

### 4.1 Encrypted Mesh Algorithm

Our current botnet detection mechanism relies on Layer 7 payload extraction of IRC message types and channel names. It also relies on the TCP work weight which is keyed on a source IP address and counts TCP control messages versus data messages (SYNS, etc versus non-control plane packets). What happens when and if black hat hackers use encryption for their Layer 7 payloads? For example, our IRC tokenization could be defeated by simply using the Caesar cipher at Layer 7. With encryption we lose IRC tokenization, although our TCP syn module, would still capture individual TCP-based scanning hosts. We would lose the ability to see the network mesh organization. Another obvious question is what might happen if a mesh is deployed that uses UDP-based attacks (or ICMP attacks) as opposed to TCP-based attacks? Encrypted meshes may exist in the near future or exist now. We simply do not know. As a result, we intend to create a new list tuple and associated algorithm that will address these problems.

Before we discuss our proposed encrypted mesh capture algorithm, we want to offer some analysis of the underlying situation. First of all it is reasonable to ask if our TCP anomaly capture system in terms of botnets is liable to "go off the air" due to encrypted botnets? The answer is no. This is because scanning is not liable to be encrypted between "owned" host A, and "unowned" host B. They have no implicit or explicit trust relationship. We do not believe that TCP-based scanning will disappear in the short term.

On the other hand, we also possess a UDP-based metric that we have not mentioned yet. We call this metric the *UDP work weight*. For each IP source address of interest (the key), one simply counts packets sent, packets returned, and ICMP errors (like destination unreachable) and calculates a weight as follows:

$$w = (U_s - U_r) * I_r$$

The resulting weight tends to be high if the network in question returns ICMP errors. This has a tendency to reveal UDP scanners and works well, although unlike the TCP work weight, the UDP work weight is unbounded. The reader should note that the UDP work weight depends heavily on ICMP errors and thus if an attack is TCP-based, it may work less well, as in general TCP returns resets. On the other hand, the UDP work weight is actually based on Layer 3 (IP layer) counts, not UDP Layer 4 counts and might actually defeat encryption at Layer 4. We propose to pair this work weight with the TCP work weight in our new tuple.

The encrypted mesh tuple will take an IP source address as a fundamental element. The tuple will have the rough following form:

(IP source address, counters, UDP

work weight, TCP work weight, list of IP peers)

This is only a sketch of the tuple and the tuple form itself will be more complex and include packet counts and other bits of information gleaned from the underlying TCP syn, and UDP work weight modules that already exist.

The algorithm in the probe will be stateful and will run to completion over two thirty second sample periods. The first sample period is used for capturing a small set of anomalous hosts. The second sample period is used to determine the set of peer IP hosts for the anomalous hosts (who they talk to). Thus we will be able to gather a mesh of hosts where the mesh exhibits a high rate of anomalies. Thus there is a state machine here with two states: "anomaly collection", and "peer collection".

The first sample period will be used for collecting two sets of anomaly-based hosts. The first set will consist of those hosts with a TCP work weight higher than T, a configured value. The second set will consist of those hosts with a UDP work weight higher than U, a configured value. Experience has shown that in current 30 second periods, our campus will generate a peak total of 15000 IP source tuples for TCP, and far fewer for UDP. These sample sets will typically be far smaller. Anomalous TCP work weight sets are currently less than 100 (and would be smaller yet if P2P hosts could be ruled out). The two sets are merged to make one set of anomalous hosts each of which has work weights for TCP and UDP.

In the second sample period, we ask the question: who talks to the scanners? Note the important point that here we look for packets (at Layer 3) returned to the anomalous hosts in question. If no packets are returned, we will drop the can-

didate host from our anomaly subset as we interested in mesh traffic, not individual wormy hosts. At the conclusion of the second sample period, tuples are written to an output file which is copied to the backend. The backend will run an hourly analysis summarizer similar to our current IRC code that will correlate the IP address meshes and produce unions of hosts. We suspect that encrypted IRC-based botnets will be revealed by simple traffic analysis of packet counts. Large encrypted meshes will of course be very suspicious. In point of fact, we should be able to correlate our current "evil channel" botnets when they exist with results from this tuple so the current IRC work will help us debug the new tuple. Note there is currently no intention to insist that payloads be encrypted. In addition, we know that servers typically have higher packet counts. Thus it may be easy to spot the server in an encrypted IRC mesh.

## 4.2 Commercial Enhancements to Ourmon

In this section we are going to briefly suggest some work aimed at improving Ourmon for a commercial product.

1. Improved GUI - We would expect to improve the GUI on Ourmon as the current version is quite primitive. Ourmon's GUI is web-based and that is good, but it is not sufficiently web-based as it often falls back on ASCII reports that lack hypertext links to interesting data elsewhere. Thus hypertext connectivity should be an important GUI design improvement. For example, one should be able to drill down on per IP host information anywhere in the GUI (which will need database support). In general, it would also be good to provide a uniform interface to help as Ourmon is quite complex.
2. MySQL database - Although Ourmon as an overall system tries to produce less data in a more summarized fashion, it still produces a lot of data, some of which is available as reports on the web, and some of which is logged and is not available through the user interface. It may be useful then to support queries that enable us to find information for an individual host in the current day, or across the current weekly period of logging for a single host. It would also be of interest to support a query that would allow the user to find information on all of the hosts in a botnet. Thus we believe it would be reasonable to use an open source database like MySQL for some carefully chosen Ourmon data. To begin with, this would include IRC information, and TCP and UDP data for individual anomalous hosts. A few query types could thus be introduced into the web interface to allow an analyst to more easily determine information about particular IP hosts. (Currently one must use grep on the logs and reports).
3. Probe Optimization - Even though Ourmon is a statistically-based system, the efficiency of the front-end probe is always a problem. Hence packet loss is an issue (although not the issue it would be with a signature-based system, where loss of one packet may mean the system failed to see an SQL slammer attack). As time passes, one always wants to do more work in a world that provides more packets. There is also the added factor of DOS attacks with small UDP or TCP

packets to consider, which can cause loss in a system that under normal circumstances is not lossy.

The current Ourmon system works well at peak 50k packet per second rates on a P4 2.8 Ghz hyperthreaded SMP FreeBSD system. We know from experience that multiple CPUs are important even when done cheaply with a hyperthreaded Pentium CPU, simply because one CPU is used for interrupt processing and the other CPU is thus free to allow our probe application to run in parallel. As a result, we think that two target hardware probe systems might be considered, a cheap version that consists of one hyperthreaded CPU (or cheap dual platform), and a more expensive dual (not hyperthread) or quad-based platform. We possess student generated code for the probe that shows how the front-end may be made parallel, and as a result we expect that the development of a thread-based Ourmon probe will be a short-term project.

## 5 Conclusion

In this white paper we have presented the architecture of Ourmon and discussed how we have combined TCP-based anomaly detection with IRC tokenization and IRC message statistics to create a system that can clearly reveal both client and server botnets.

In addition we have discussed how we can take our current system and improve it so that it can perform more efficiently, provide a better user interface, and quite probably capture stealthy botnets that use encryption. Consequently we believe that the commercialization of Ourmon should prove quite straightforward and should be

achievable within the time frame of one year.

## References

- [1] J. Binkley, B. Massey, Ourmon and Network Monitoring Performance. *Proceedings of the Spring 2005 USENIX Conference, Freenix track*, Anaheim, April 2005.
- [2] CERT Advisory CIAD-2004-10 Multiple Vulnerabilities in Microsoft Products <http://www.cert.org/advisories/ciad-2004-10.htm>, April 2004.
- [3] Cisco Systems. Cisco CNS NetFlow Collection Engine. [http://www.cisco.com/en/US/products/sw/netmgts/ps1964/products\\_user\\_guide\\_chapter09186a00801ed569.html](http://www.cisco.com/en/US/products/sw/netmgts/ps1964/products_user_guide_chapter09186a00801ed569.html), April 2004.
- [4] Know Your Enemy, Tracking Botnets. The HoneyNet Project and Research Alliance. <http://honeynet.org/papers/bots>, March 2005.
- [5] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. *Proceedings of the Winter 1993 USENIX Conference*, San Diego, January 1993.
- [6] J. Oikarinen, D. Reed. Internet Relay Chat Protocol. IETF RFC 1459, May 1993.
- [7] Ourmon web page. <http://ourmon.cat.pdx.edu/ourmon>, December 2005.
- [8] Ourmon architecture description web page. <http://ourmon.cat.pdx.edu/ourmon/info.html>, December 2005.



- [9] RRDTOOL web page.  
<http://people.ee.ethz.ch/~oetiker/webtools/rrdtool>. December 2005.
- [10] Sourceforge Ourmon web page.  
<http://ourmon.sourceforge.net>,  
December 2005.
- [11] Waldbusser, S. Remote Network  
Monitoring Management Information Base  
Version 2. IETF. RFC 2021, January 1997.
- [12] Wikipedia web page.  
[http://en.wikipedia.org/wiki/Internet\\_Relay\\_Chat](http://en.wikipedia.org/wiki/Internet_Relay_Chat), December 2005.

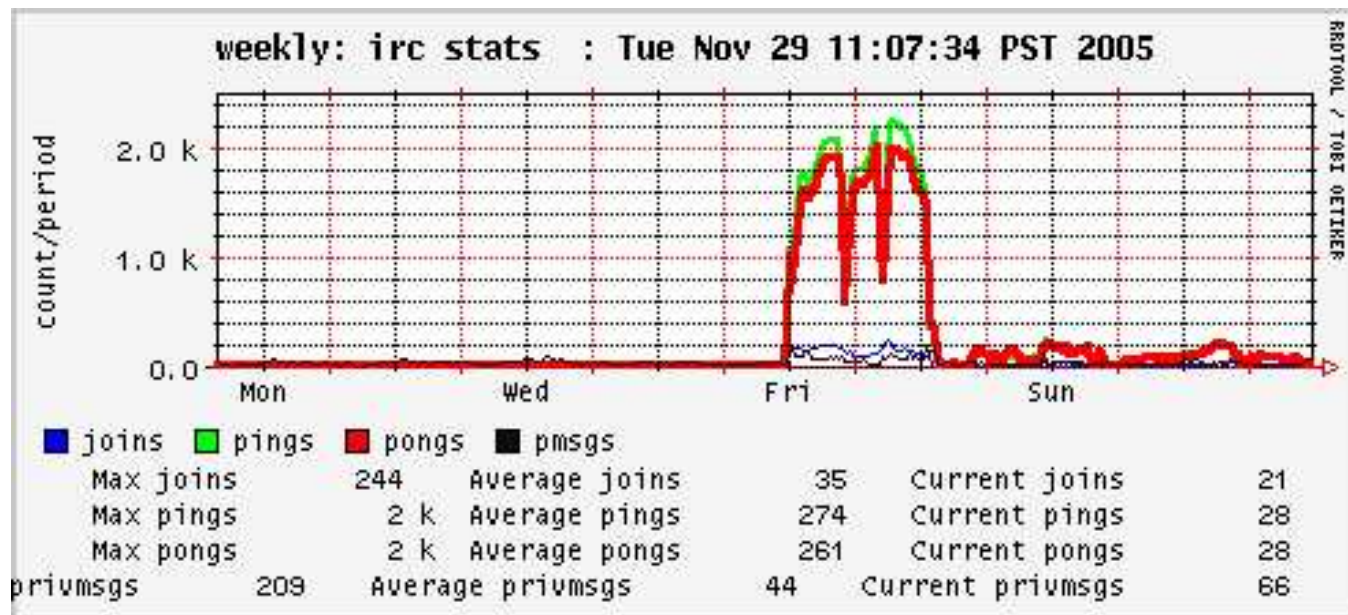


Figure 1: Thanksgiving Botserver IRC Message Counts

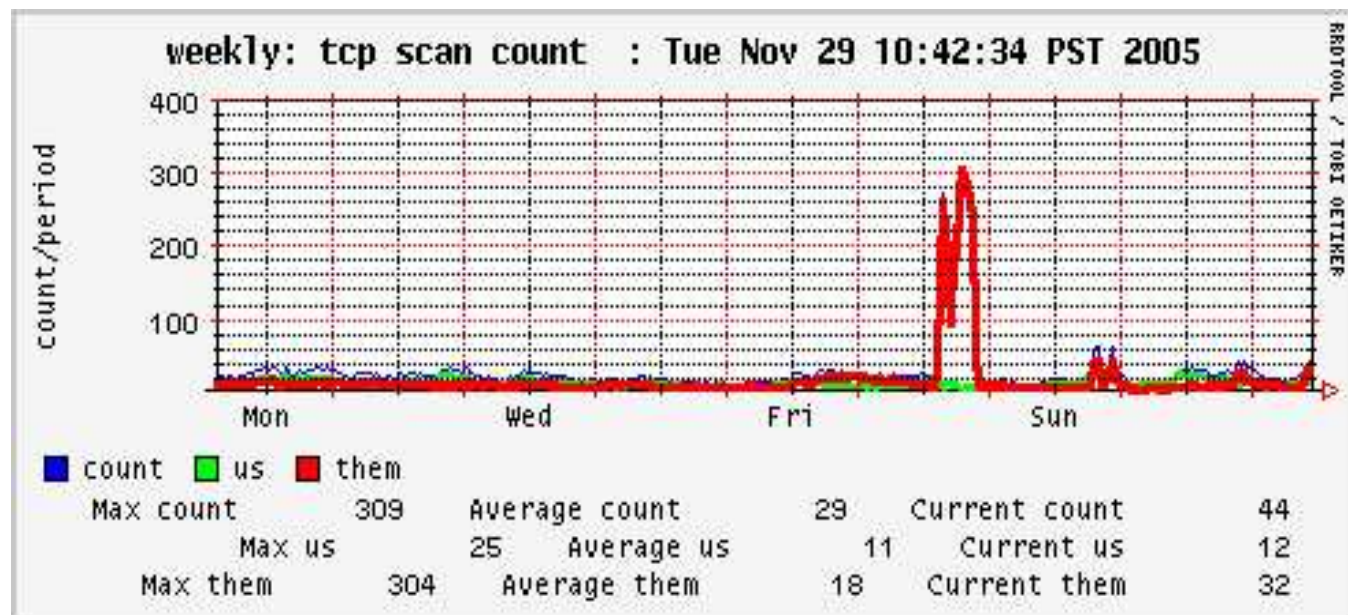


Figure 2: TCP Scanners During Bot Attack