

THE DARK CLOUD: UNDERSTANDING AND DEFENDING AGAINST BOTNETS AND STEALTHY MALWARE

Contributors

Jaideep Chandrashekar

Intel Corporation

Steve Orrin

Intel Corporation

Carl Livadas

Intel Corporation

Eve M. Schooler

Intel Corporation

Index Words

Malware

Botnets

Network Defense

Cyber-Security

Stealth

Rootkits

Abstract

The proliferation of botnets reveals a worrisome trend in the spread and sophistication of computer viruses and worms in the Internet today. (A *botnet* is essentially a collection of compromised distributed computers or systems, known as *bots* because of their zombie-like nature, under the control of a *bot-herder*, by virtue of the use of command and control servers.) Botnets are the latest scourge to hit the Internet, each one revealing a new level of technologic expertise and the use of quality software processes that undermine, if not downright prohibit, the ability of current anti-malware and other intrusion detection systems (IDSs) to deal with them. Most IDSs focus on detecting known threats, or on detecting the volume of traffic generated by a bot-host after it has been activated. Most bots, however, are polymorphic: they change with every instantiation so appear as something new every time. Furthermore, most bots generate only low-volume, periodic communication back to a bot-herder, and this volume is generally within the thresholds used by IDSs. In this article, we present an overview of the state of the art of botnets and stealthy malware, then develop and present several promising anti-botnet defense strategies that specifically target current and emerging trends in botnet development.

Introduction: Current and Emerging Trends in Botnets

With estimates of botnet infections continuing to gain in momentum, botnets are the latest scourge to hit the Internet and are the latest challenge for IT personnel. Each new botnet discovered reveals the use of more advanced technology and the use of quality software processes that are challenging the defense strategies of current intrusion detection systems (IDS). Thus, we begin this article with an overview of the state of the art of botnets and stealthy malware. We first describe the botnet lifecycle and highlight the advanced capabilities and stealth techniques in use today by botnets; we also examine and strategize about future advances in this area. We then go on to present several promising anti-botnet defense strategies, notably a collection of real traces to calibrate normalcy, the development of techniques that analyze communication with remote nodes with the goal of identifying botnet command-and-control (C&C) channels, and the application of various forms of correlation to amplify accuracy of detection and to root out stealthiness.

“Botnets are the latest scourge to hit the Internet and are the latest challenge for IT personnel.”

Botnets Defined

A botnet is a collection of distributed computers or systems that has been compromised, that is, taken over by rogue software. As a result, these machines are often called *zombies* or *bots*. Bots are controlled or directed by a *bot-herder* by means of one or more C&C servers. Most commonly, the bot-herder controls the botnet with C&C servers, delivered via protocols such as internet relay chat (IRC) or peer-to-peer (P2P) networking communications. Bots typically become installed on our devices via malware, worms, trojan horses, or other back-door channels. Further information on botnets can be found in [1].

The statistics for the size and growth of botnets differ widely, based on the reporting organization. According to Symantec's "Threat Horizon Report" [2], 55,000 new botnet nodes are detected every day, while a 2008 Report from *USA Today* states that "...on an average day, 40 per cent of the 800 million computers connected to the Internet are bots used to send out spam, viruses and to mine for sensitive personal data" [3]. *USA Today* also reports a tenfold increase in 2008 in the code threats reported over the same period in 2007, signifying the increase in threat surface area for botnet-style infections [3]. Various sources estimate that the best-known botnets—Storm, Kraken, and Conficker—have infected staggering numbers of machines. These numbers range from 85,000 machines infected by Storm, to 495,000 infected by Kraken [4], to a staggering 9 million nodes infected by Conficker [5].

The Underground Economy and Advances in Botnet Development

Like any money-driven market, botnet developers operate like a legitimate business: they take advantage of the economic benefits of cooperation, trade, and development processes, and quality. Recently, botnets have begun to use common software quality practices such as lifecycle management tools, peer reviews, object orientation, and modularity. Botnet developers are selling their software and infection vectors, providing documentation and support, as well as collecting feedback and requirements from customers.

Common economic goals are driving innovation, collaboration, and risk reduction in the Botnet communities. On-line barter and marketplace sites have sprung up to service this underground community with barter and trade forums, on-line support, and rent and lease options for bot-herders. This cooperation has led to a fairly mature economy where botnet nodes or groups are bought and sold, or where several bot-herders can cooperate when targeting an entity for attack. Botnets can be rented for the distribution of spam. Stolen identities and accounts are traded and sold among the participants.

"A botnet is a collection of distributed computers or systems that has been compromised."

"Bots typically become installed on our devices via malware, worms, trojan horses, or other back-door channels."

"Botnet developers are selling their software and infection vectors, providing documentation and support, as well as collecting feedback and requirements from customers."

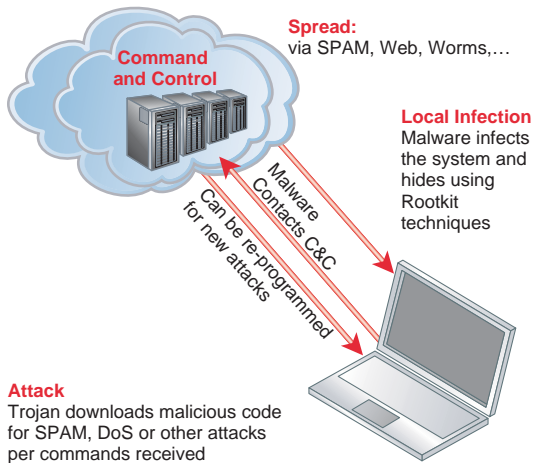


Figure 1: The Botnet Lifecycle
Source: Intel Corporation, 2009

“By polymorphism, we mean that the malware code changes with every new infection.”

“By rootkitting, we mean the stealthy installation of malicious software—called a rootkit—that is activated each time a system boots up.”

The Botnet Lifecycle

The lifecycle of a botnet typically includes four phases: spread, infection, command and control (C&C), and attack, as shown in Figure 1. We describe each phase.

Spread Phase

In the spread phase in many botnets, the bots propagate and infect systems. Bots can spread through a variety of means, including SPAM e-mails, web worms, and through web downloads of malware that occur unbeknownst to users. Since the goal of the spread phase is to infect a system for the first time, bot-herders attempt to either trick the user into installing the malware payload or exploit vulnerabilities on the user system via applications or browsers, thereby delivering the malware payload.

Infection Phase

The malware payload, once on the system, uses a variety of techniques to infect the machine and obfuscate its presence. Advances in bot infection capabilities include techniques for hiding the infection and for extending the life of the infection by targeting the anti-malware tools and services that would normally detect and remove the infection. Botnets employ many of the standard malware techniques in use by viruses today. Polymorphism and rootkitting are two of the most common techniques in use.

- By polymorphism, we mean that the malware code changes with every new infection, thus making it harder for anti-virus products to detect the code. Further, the use of code-hardening techniques often employed by SW developers to protect from SW piracy and reverse engineering, are in turn used by botnet developers. These techniques include code obfuscation, encryption, and encoding that further hide the true nature of the malware code as well as making it harder for anti-virus vendors to analyze it. There are indications that malware and botnet developers are beginning to look into advanced rootkitting techniques to further hide the malware.
- By rootkitting, we mean the stealthy installation of malicious software—called a rootkit—that is activated each time a system boots up. Rootkits are difficult to detect because they are activated before the system’s operating system (OS) has completely booted up. Advances in rootkit techniques include hyperjacking and virtualization-based rootkits as well as identifying and using new targets for code insertion such as firmware and BIOS.

A virtual machine monitor (VMM) or hypervisor runs underneath an OS, making it a particularly useful means for botnet and malware developers to gain control of computer systems. Hyperjacking involves installing a rogue hypervisor that can take complete control of a system. Regular security measures are ineffective against this hypervisor, because the OS is unaware that the machine has been compromised, and software anti-virus and local firewalls are unable to detect them.

Another technique that is currently used by botnet developers is to actively target the anti-virus, local firewall and intrusion prevention and detection software (IPS/IDS) and services. Some of the techniques employed by botnets have included attacking the anti-virus and firewall software by killing its process or blocking its ability to get updates. Two examples that we know of show how botnets blocked the security software from getting updates:

- A botnet changed the local DNS settings of the infected system to disable the anti-virus software from reaching its update site.
- A botnet was actively detecting connection attempts to the update site and blocking them.

These update-blocking techniques prevent the security software from getting potential updated signatures from the vendor that identify the newer version of the botnet or from being able to communicate with a central vendor server for anomaly correlation and update.

Timing the infection to strike between malware detection services scan times is another infection technique employed by botnet developers. The bot slowly infects a system without generating alarms in the intrusion detection software services.

Other advanced bots spoof the local and remote scans performed by the IDS/IPS and anti-virus software. In this case, the botnet's malware presents a false image of memory or hard disk to the anti-virus software to scan, or the malware disrupts vulnerability scans by dropping packets, spoofing the network response, or redirecting traffic coming from vulnerability scanners.

Command and Control

Botnet C&C servers use one of several protocols to communicate, the most common of which up to this point has been IRC. Recently, however, a trend towards the use of protected or hardened protocols has begun to emerge. For example, the Storm botnet uses an encrypted P2P protocol (eDonkey/Overnet). Advances in C&C techniques are crucial for bot-herders to keep their Botnets from being detected and shut down. To this end, botnets have begun to leverage protocols such as HTTP and P2P that are common across networks, thus making the botnet harder to detect. HTTP is particularly advantageous to botnets because of the sheer volume and diversity of HTTP traffic coming from systems today. Also, botnet software can take advantage of the local browser software for much of its functionality and communications stack, leveraging HTTP's ability to transit firewalls. Other techniques on the horizon include the use of VoIP, web services, and the use of scripting within the HTTP communications stack. Another advanced technique uses a *blind drop*, a site on the Internet such as a forum, BBS, or a newsgroup, where users can leave anonymous messages. Botnet nodes can post messages to these sites, and bot-herders can anonymously check for messages from their nodes and post instructions. The botnet nodes can then poll the site for new instructions and other communications as part of a messaging-based C&C. Social networking sites are a prime target for this kind of C&C.

“Timing the infection to strike between malware detection services scan times is another infection technique employed by botnet developers.”

“Botnet software can take advantage of the local browser software for much of its functionality and communications stack, leveraging HTTP's ability to transit firewalls.”

“Steganographic techniques are the next method by which botnet developers plan to evade detection.”

“Phishers, hackers, spammers, and virus writers use the botnet to sell information and services.”

“Most bots are polymorphic: they change with every instantiation so always appear as new.”

A key feature of modern botnet development is the ability to re-program or update the botnet node software after it has infected a system. The C&C directs the node either to download the update directly or to go to a specific infected site hosting the update. Botnets with this reprogrammability have a higher value in the underground economy, as they can be augmented to perform new and advanced attack and stealth missions as they are developed.

As mentioned previously, stealth is a key feature of botnet technology. Kracken and Conficker Botnets both target and disable anti-virus software resident on the system. Other botnets deliberately try to hide from threshold-detection software by customizing the timing of infections and the frequency of communications to hide activities from both local and network security products. Steganographic techniques are the next method by which botnet developers plan to evade detection. They include the use of covert channels for communications and steganography-based messaging, such as mimicry and stegged content (i.e., embedding messages in content such as images, streaming media, VoIP, and so on).

Attack Phase

The final phase of the botnet lifecycle is the attack phase. In many cases the attack is simply the distribution of the SPAM that is carrying the infection, and when the attack is successful, the size of the botnet itself increases. Botnets also often have been used to send SPAM as part of barter and rental deals, whereby phishers, hackers, spammers, and virus writers use the botnet to sell information and services. Botnets also have been used to perform massive distributed denial-of-service (DoS) attacks against a variety of targets including government, corporate systems, and even other botnets. Some of the newer botnets can be upgraded to use various hacker tools, fault injectors (fuzzers), and so on, to further attack the networks they have infiltrated. For example, the Asprox botnet included an SQL injection attack tool, and another botnet included a Brute Force SSH attack engine. In addition to performing remote attacks, botnets can engage in persistent local attacks to phish for identities and accounts from the infected system and its users.

The Evolution of Anti-Botnet Strategies

Given the proliferation and sophistication of malware, it is not hard to see why traditional anti-malware techniques don't work against botnets. Most IDS focus on detecting known threats, or on detecting the volume of traffic generated by a bot host, after it has been activated. However, most bots are polymorphic: they change with every instantiation so always appear as new. Furthermore, most botnets generate only low-volume periodic communication back to a bot master, and this volume is generally within the thresholds used by IDS.

In the remainder of this article, we describe the Canary detector that targets early botnet detection. The Canary detector encompasses three promising anti-botnet strategies. The first strategy employed is the analysis of real enterprise network traces that reveal how the network is actually used; this analysis, in turn, reveals how certain user-driven traffic properties differ from botnet traffic. Our second strategy is an end-host detection algorithm that is able to root out the botnet C&C channel. Our approach is based on the computation of a single persistence value, a measure of how regularly remote destinations are contacted. The strength of this method is that it requires no *a priori* knowledge of the botnets that are to be detected, nor does it require inspection of traffic payloads. Although the botnet detection capability may be carried out solely at an individual end-host, we show that detection is further improved by correlating across a population of systems, either at a network operation center (NOC) or in a completely de-centralized fashion, to identify the *commonality* in persistent destinations across multiple systems. This is our third strategy.

The Design of the Canary Detector

The Canary detector takes a novel approach to detecting stealthy, end-host malware, such as botnets. Here we use the term *stealthy* to mean not generating a noticeable level of traffic. The central idea in our detection scheme is to track the usage of *destination atoms*, the logical collections of destination addresses that describe services. Specifically, we measure the correlation of destination atoms—temporally for individual users, and spatially across sets of users—and scrutinize those destination atoms that become significant. In the case of botnets, for example, the recruited end-hosts typically call home periodically. By tracking this destination atom over time at a coarse level, we can flag it when it becomes significantly persistent.

Preliminaries

Destination Atoms in Intel Enterprise Traces

Interested in studying correlations between user activity and network traffic patterns, we launched an enterprise data collection effort from inside Intel's corporate network. We collected traces (over a 5-week period from approximately 400 end-hosts) that we and others subsequently data-mined for interesting phenomena, statistics, and contradictions of long-held assumptions [6].

“The Canary detector encompasses three promising anti-botnet strategies.”

*“Here we use the term **stealthy** to mean not generating a noticeable level of traffic.”*

Looking at real enterprise traces, we can see that there are substantial efficiencies to be gained when correlating destination usage. Thus, our Canary algorithms rely on a level of abstraction we call *destination atoms*, that is, logical representations of network services. This level of summarization leads to a significant reduction in the number of destination *entities* that are tracked, and thus, tracking atoms requires less overhead. The base definition for a destination corresponding to a connection is the tuple (destIP, destPort, proto), which is simply the end-point for the connection consisting of the destination address, the destination port, and the transport protocol that is used. Often, in the case of well-known services, multiple physical hosts provide the same, indistinguishable application service. Thus, we can group the set into a single atom (dstService, dstPort, proto). Here, the *service* is simply the domain name to which the underlying addresses resolve. Examples of atoms include (www.google.com, 80, tcp), (akamaitech.com, 80, tcp), and (mail.cisco.com, 135, tcp).

“Our Canary algorithms rely on a level of abstraction we call destination atoms, that is, logical representations of network services.”

Further summarization is also possible by applying heuristics on how ports are used by applications. Consider an FTP server, connected in PASV mode. The initial connection is over port 21, but a separate server-negotiated ephemeral port is used for data transfer. Thus, a single FTP session has two atoms, (ftp.service.com, 21, tcp) and (ftp.service.com, k, tcp), where k is a port number beyond 1024, which can be viewed as offering the same service. By considering FTP semantics, we can add the entire range of ports larger than 1024 to the associated atom (ftp.service.com, 21:>1024, tcp). This means that, when we see a connection on port 21, we can expect an ephemeral port to be used in the near future.

In the real enterprise traces, we had many occasions to perform this level of summarization, most notably on the Microsoft* RPC ports between 135 and 139. We then arrive at the full definition of destination atom, the triple (addr set, port set, proto). Here, addr set is a set of destination addresses: these addresses are identical with respect to the applications provided; port set is a set of individual ports or port ranges; and finally, proto is the transport protocol the service uses. Table 1 enumerates some atoms extracted from the enterprise traces.

Destination Atom	Description
(google.com, 80, tcp)	HTTP sessions to any of the Google servers
(ftp.nai.com, 21:>1024, tcp)	Updates for Norton antivirus delivered via PASV FTP from the Norton Web site
(mail.cisco.com,135:>1024,tcp)	Microsoft RPC-based services use ephemeral ports after the session is negotiated over port 135

Table 1: Atoms Extracted from Enterprise Traces

Source: Intel Corporation, 2009

Note that a single destination host can provide a number of distinct services, and in this case, the port is sufficient to disambiguate the services from each other, even though they may have similar service names, which are obtained by (reverse) DNS lookup. Finally, note that in cases where the addresses cannot be mapped to names, no summarization is possible, and the conventional destination address is the final descriptor.

Persistence

The key anti-botnet technique we propose is to identify *temporal heavy hitters* without regard to their level of traffic; that is, identify services that get used with a degree of *regularity*. Again, this strategy was validated by the analysis of real enterprise traces from a diverse group of end users in varied geographic regions with disparate usage patterns. We believe that the set of significant atoms for an end-host is small and stable, and that when a host is infected with malware, it will connect periodically to a home server, and the latter will stand out. To perform this detection, we must first assign a numeric value to the somewhat nebulous concept of *regularity*, which we refer to as the persistence of an atom. We want to track the regularity of *usage*, rather than the connections themselves. Consider the act of using your newsreader to download the news headlines. Each time the newsreader application is launched, it makes a large number of connections. To track the long(er)-term communication with the end-host, we concentrate on tracking high-level *sessions*, rather than individual connection frequencies.

To track high-level sessions, we bin connections to the atom by using a small *tracking window*, w , and we assign a 1 or a 0 to that window (the atom was seen 1 or more times, or not). Clearly, the tracking window length should cover *sessions*. When we plot the inter-arrival time for individual atoms across a large number of users, we see that 59 percent of the connections to atoms are made within a minute of each other, and 87 percent of connections to the same atom are separated by at least an hour. We therefore select an hour as the tracking window length to compute persistence.

The other step needed to assign a numeric value to persistence is the construction of an observation window, W ; that is, we look at how long an atom should be regularly observed before it is classified as significant. Based on experience with the data, we defined the observation window, $W = 10w$, which roughly covers the average work day. Having defined w and $W = (w_1, w_2, \dots, w_{10})$, we quantify persistence for an atom a , as observed at host h , over the observation window W , $p(a, h, W)$, as the number of individual windows w_1, w_2, \dots, w_n where the atom was observed.

If we denote p^* as a threshold for an atom to be significantly regular, then if $p(a, h, W) > p^*$, the destination a is considered persistent for host h . Note that the definition of persistence has an inherent timescale dictated by W . Suppose that $w = 1$ hour and $W = 1$ day. When computed at this scale, persistence captures the day-to-day behavior of the atom. However, it fails to capture longer-term trends that may exist. Consider two different atoms: a_1 , seen every hour, and a_2 , observed once a day. We have $p(a_1) = 24/24$ and $p(a_2) = 1/24$.

“We believe that when a host is infected with malware, it will connect periodically to a home server, and the latter will stand out.”

“We use commonality to quantify how correlated a destination atom is across the users in a network.”

“Persistence requires a means for the system to collect and correlate information across end-hosts.”

Intuitively, however, they are both quite *regular* and thus both should be termed persistent. In fact, because we are trying to detect stealthy malware about which we have no *a priori* timescale information, the one timescale we pick may be the exact one that misses the malware activity. Thus, instead of relying on a single timescale W , we consider five different timescales, W_1, W_2, \dots, W_5 . Therefore, for every atom, we compute $p(a, h, W_i)$ for $i = 1, 2, \dots, 5$ and say that it is persistent if $\max_i p(a, h, W_i) > p^*$

Commonality

While persistence is defined as a property of the individual end-user, we use commonality to quantify how correlated a destination atom is across the users in a network. Thus, a destination atom is significant in this dimension if a large fraction of the users are communicating with it. Since these atoms are created because of many users in a network, we expect them to be quite stable among the population. The commonality metric is defined quite simply: let $N(a)$ be the number of users in the population that see the atom a , at least in some observation window. Thus, the commonality of atom a , $c(a) = N(a)/N$, where N is the total number of hosts in the network. Additionally, we could require a minimum persistence for the atom across the set of hosts that report connections to it; doing so would counter the effect of temporary transients such as flash crowds.

Unlike persistence, this commonality metric cannot be computed in isolation at an individual end-host. Persistence requires a means for the system to collect and correlate information across end-hosts. One solution is to assume the existence of a central IT operations center (ITOC) that can collect periodic reports of atoms observed from all the end-hosts, and that can determine the significant *common* atoms in the set. Alternatively, peer systems can share persistence information periodically with like-minded subsets of the population (e.g., proximate peers, those running a similar OS or patch level, those deemed trusted via the social network of users at the application layer, and so on).

In contrast to the ITOC approach, significantly common atoms are determined and maintained at the end-hosts, as in [7]. In either scheme an important point is that a sliding window is maintained over the entire observation window (the largest among the different timescales). While computing the commonality metric, only reports within this observation window are considered. Again, the test for significance is when the value of $c(a)$ is greater than a specific threshold c^* . When $c(a) > c^*$, we say that ‘ a ’ is common in the population.

Building Whitelists

We construct a whitelist for each user in two steps. First, the host observes its traffic for a training period, builds the set of atoms, and tracks their persistence; the length of this training period would vary with how stable the traffic patterns are, and we expect this to be defined by the network operator. We define p^* to be the persistence threshold; that is, if the persistence of a particular atom is larger than p^* , then the atom is added to the whitelist. In the detection phase, each end-host sends its set of observed atoms (all of them, not just the persistent ones) either to the central ITOC of the enterprise or to a subset of like-minded peers. At the ITOC, the commonality is calculated for each atom in the union. We define a threshold for commonality, c^* , and collect those atoms whose commonality exceeds c^* . These atoms are sent to every end-host, where they are incorporated into the whitelist. Thus, every host's whitelist has two components: an individual component capturing behaviors unique to that host, and a global component that corresponds to behavior that is common to the population. The global component can contain atoms that are not part of the individual host's regular behavior.

“Every host’s whitelist has two components: an individual component capturing behaviors unique to that host, and a global component that corresponds to behavior that is common to the population.”

Detection Algorithm

At a high level, our system generates alarms corresponding to two types of events. These are classified as (1) p-alarms, when a destination atom not contained in the host's whitelist becomes persistent and (2) c-alarms, when a destination atom is observed at a large number of end-hosts in the same window and is identified as common. Note that p-alarms are generated locally; the user is alerted and asked to acknowledge the alarm. In contrast, c-alarms are raised either at the central ITOC or locally, if full whitelists are distributed among peers. Note that when the alarm corresponds to an atom becoming significant, one of two things must happen: either the atom is classified as benign (by a user or operator) in which case it must be added to the appropriate whitelist, or else the alarm indicates malicious behavior, requiring remediation action. In this article, we do not address the remediation stage; we simply note that a number of possibilities have been suggested in the literature, such as throttling traffic, redirecting traffic through a scrubber, blocking traffic, and so on.

processPacket(pkt, t, wi)

```

1.   a <-- getDestAtom(pkt)
2.   if a in WHITELIST then
3.     return /* ignore atoms already in the whitelist */
4.   end if
5.   if a is a new connection initiation then
6.     DCT[a][currIdx] = 1 /*update persistence */
7.     sendReport(userID, a, t) /*report sent to central console*/
8.   end if

```

Code listing 1: Outgoing Packet Processing

Source: Intel Corporation, 2009

“Note that our system is not tied to any particular traffic feature or threshold definition.”

In the rest of this section, we briefly review the specific actions required to process outgoing packets (summarized in Code listing 1). When the (outgoing) packet corresponds to an atom already in the individual host whitelist, nothing further is done. If the outgoing packet does not correspond to an atom already in the host whitelist, then the following steps are taken:

- If the atom was not previously seen, a new entry is created in the data structure used to track persistence (DCT); this is indexed by the atom and points to a bitmap. Each bit corresponds to a particular tracking window.
- The data structure that tracks the observations of atoms (labeled DCT) is updated for the current tracking window.
- The atom, if new, is sent to the ITOC (possibly after being filtered through a minimum persistence criterion).

Note that our system is not tied to any particular traffic feature or threshold definition; for convenience, we assume connections per minute as the feature under consideration. To generate p-alarms, we track persistence at all the timescales by employing a sliding window. The data structure to do this is depicted in Figure 2. A dictionary (or hash table) is maintained, in which an atom is indexed, and this dictionary entry reveals the particular bitmap associated with the atom. When the atom is observed in a tracking window w_i , the i_{th} bit is set to 1 as described in Figure 2. As the sliding window is advanced, at the end of the last window, the persistence is computed for each atom observed in the last tracking window. It would seem that doing this for multiple timescales would be expensive. However, an interesting observation is that we do not need to replicate the structure at different timescales. Instead, we can exploit the overlapping nature of the timescales ($W_3 < W_4$); we can get away with this by using a single long bitmap that has enough bits to cover the longest observation window.

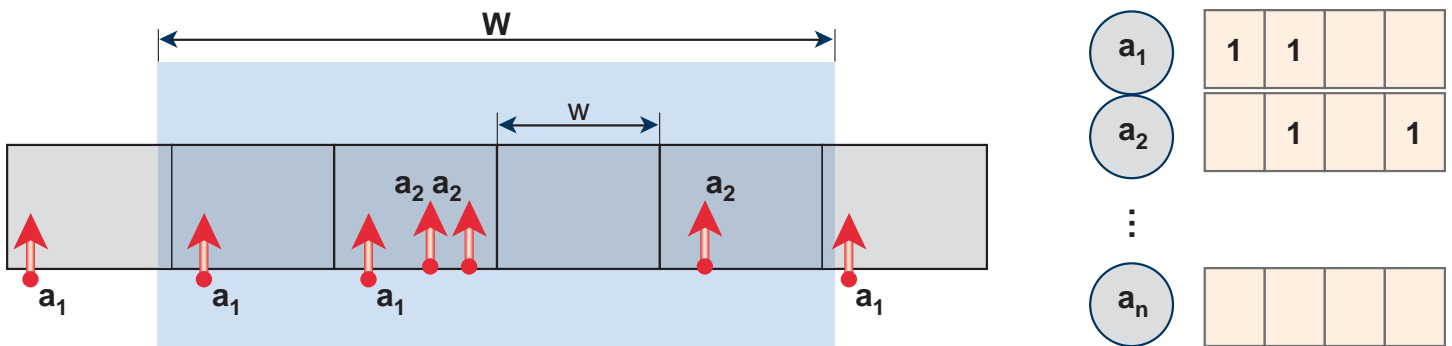


Figure 2: Data Structure Used to Track Atom Persistence

Source: Intel Corporation, 2009

If at any time, the persistence value of the atom exceeds the threshold p_* , an alarm is raised for the atom; at this time, the user is asked to attest whether the atom is valid and should be added to the whitelist. If the value is not significant even after sufficient tracking windows, the bitmap is cleared out and the atom is no longer tracked (a new bitmap is instantiated if it ever appears again).

To understand the overhead imposed by this procedure, we note that the length of the dictionary need not be large. If an outgoing packet is already in the whitelist (specifically, if its atom is in the whitelist), then no new dictionary entry is required. For everything else, we only need one entry per atom (even if the same atom has many connections or packets associated with it). With atoms that actually need to be tracked, the computation involved is simply the time it takes to index the dictionary and update the bitmap. However, we see in the traffic that most atoms that we track occur very infrequently (and that the most obviously persistent atoms are already in the whitelist and do not need to be tracked). Therefore, most entries in the bitmap are empty; an easy optimization would be to use sparse vectors in lieu of bitmaps. In our analysis, we found that the worst-case scenario over all users, and all observation windows W_{\max} had 1435 atoms requiring tracking. The average case was 485 atoms. This is almost negligible if one considers the computational power and memory associated with modern-day mobile systems.

We conclude this discussion by briefly discussing how the c-alarms are generated through tracking commonality—a very straightforward operation. The central console at the ITOC keeps track of atoms seen by different users over the largest observation window. When a report arrives from a host, the corresponding atom is updated. At the same time, old information is expunged (that is, sightings of an atom older than the observation window are discarded). When an atom's entry is updated, and the number of associated users (who have seen this atom recently) crosses the threshold c_* , a c-alarm is generated. The frequency with which a host sends reports to the central console determines how soon an anomaly will be detected. Dispatching the report immediately (as soon as the atom is first seen) helps with catching the anomaly early, but at the cost of communication. Batching updates reduces the communication cost, but increases the time to detection. While this is an interesting tradeoff to study, we do not explore it in this article.

“If an outgoing packet is already in the whitelist, then no new dictionary entry is required.”

“Batching updates reduces the communication cost, but increases the time to detection. This is an interesting tradeoff.”

“We detect three different types of anomalies: burst, persistent, and commonality.”

Testing with Malware Traces

We present the results from running our detection algorithm with traces collected from real botnets. Recall that we detect three different types of anomalies: burst anomalies, triggered by large changes in traffic distribution; persistence anomalies triggered when destinations are communicated with regularly, even with very little traffic (such as botnet C&C channels); and commonality anomalies, triggered when a number of network users begin to exhibit correlated behavior. These anomalies correspond to the three types of alarms output by our system. Table 2 lists some well-known malware types, indicating what types of alarms are likely to result from each.

	Burst alarm	p-alarm	c-alarm
(long) DDoS attack	◆	◆	
DDoS attack	◆		◆
Scanning worm	◆		
IRC botnet	◆	◆	◆
Stealthy botnet		◆	

Table 2: Well-known Malware Types and Their Alarms

Source: Intel Corporation, 2009

“One of our goals is to understand the detection of the attack behavior and the channel behavior.”

Botnet Traces

We collected traffic traces from three distinct botnet families. We executed bot code on a host and logged packet traces for a week, by using the same host over multiple weeks to run the three different bots. The host was wiped clean in between collections, and a pristine copy of Windows* XP* was installed. Also, we turned off the auto-update functionality and configured the firewall to drop all incoming connections. From each trace, we discarded all packets that did not have a source or destination address corresponding to the host. The packet traces were converted to flows by using Bro [8], and the rest of the analysis uses flows. One of our goals in this section is to understand the detection of the different behaviors; that is, the *attack* behavior and the *channel* behavior (when the malware *calls home*). In the traces we collected, we saw both. Because many bots in the wild do not generate much volume (and try to remain undetected), detecting the control channel is of critical importance. We briefly describe the three Botnets and how the flows were classified:

SDBot. An SDBot is a well-studied botnet that uses IRC as the channel but on a non-standard port. However, the IRC servers are easy to pick out from the domain names, for example irc.undernet.org. The traces revealed two distinct atoms in the control flows. The remaining flows consist of scans being run on a neighboring network prefix. We noticed a large number of scans on ports 135, 139, 445, and 2097 (a well-known commercial anti-virus product). In the traces, we see connections on the well-known IRC ports and use this knowledge to identify control traffic (the IRC traffic) and attack flows.

Zapchast. This botnet also uses IRC as the channel and uses the well-known IRC ports (6666 and 6667). We saw a total of five *IRC service atoms* (about 13 distinct IP addresses) in the traces. The attack traffic was predominantly netbios traffic.

Storm. This botnet is P2P-based and very different from the others. The traces are two orders of magnitude larger than the other botnets. Lacking a single destination server or a well-defined port, it was quite hard to identify the control channels and we had to rely on some heuristics to do this: the fact that Storm uses UDP to connect to the P2P is documented.

We looked at distributions of the UDP flows (flows with two-way traffic) and noticed a very large number of packets that were of a small, fixed size (the flows were on non-standard ports and unlikely to be attacks). We took these flows to be an indicator of maintenance traffic and isolated all the ports involved. UDP flows to this set of ports are assumed to be part of the control channel. We did see a much smaller number of HTTP and SSH flows that may also be control related; the volume of these flows is such that it does not affect our results. The attack traffic for Storm is overwhelmingly on TCP port 25 (SMTP).

Evaluation

In the rest of this section, we discuss the detection of persistence anomalies, and we defer the analysis of commonality anomalies due to space limitations.

Detecting stealthy behavior with p-alarms. To validate the detection of the control channel in each of the Botnets, we first identify the distinct atoms that can be extracted from the control traffic. For each of these atoms, we compute persistence over the lifetime of the (malware) trace. Recall that we compute this at five different timescales. For the purposes of detection, we consider the atom to be flagged as a p-alarm, if the value at any timescale exceeds the threshold $p^* = 0.6$. We found that this threshold is associated with the fewest false alarms per day and the best detection rate, where the rates were averaged over all the destination atoms for all the malware traces.

In Figure 3, we plot the maximum persistence value for each of the atoms. The Y axis indicates the value used for p^* . The scatter plot contains three distinct markers for each of the botnets, and each mark plots the persistence value for the corresponding atom. We plot a vertical line at $p=0.6$, which is the persistence threshold used by our detection system. Atoms that occur to the right of the vertical line are flagged by our system as possible C&C destinations. The particular threshold, i.e., $p=0.6$ was selected so as to achieve the best tradeoff between minimizing the number of false positives (i.e., normal, benign destinations flagged by our method as C&C destinations), and maximizing the detection rate (i.e., the fraction of C&C destinations that we correctly flag).

“Lacking a single destination server or a well-defined port, it was quite hard to identify the control channels.”

“For each of these atoms, we compute persistence over the lifetime of the (malware) trace.”

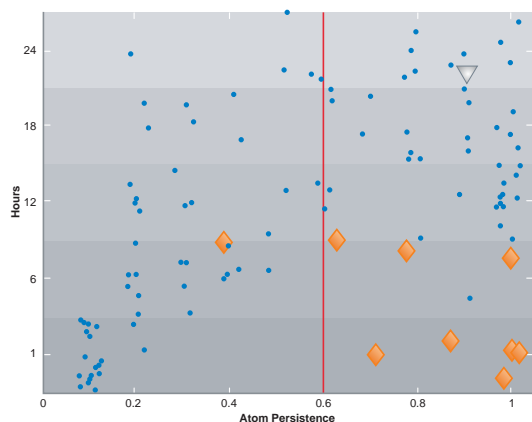


Figure 3: Detection by Persistence of Three Botnets

Source: Intel Corporation, 2009

“We are able to develop and present the Canary end-host detector, designed to root out the botnet command and control channel.”

The SDBot traces revealed exactly one atom, and this atom appears toward the top right of the plot. It is the largest marker and is shown as a triangle. The Zapchast traces contained exactly nine atoms, all but one of which appear to the right of the vertical line. Finally, the Storm traces contain approximately 82,000 atoms with persistence levels evenly distributed (for convenience, we only plot a sample of 100 atoms). While persistence is reflected on the x-axis, the vertical bands indicate different timescales. Thus, a point in the bottom band indicates the persistence value is associated with the 1-hr timescale.

We plot the maximum persistence for each destination atom, so the band indicates the timescale at which the persistence value maxed. Looking over the points, we see that the SDBot atom and eight of the nine Zapchast atoms are easily detected, appearing to the right of the threshold. For the single Zapchast atom to the left of the threshold, we noticed exactly two connections, close to each other, over the entire trace. We conclude that these connections do not really count as regular. We point out that these particular botnet instances are stealthy and generate very few connections. One of the atoms (to the right of the line) was associated with 30 connections over a whole week, with at most one connection in a window. This behavior qualifies as being close to indistinguishable. However, the persistence value for this atom is 0.7 and is above the threshold. This particular example drives home why a system such as ours is required to detect stealthy malware. With malware becoming more stealthy and with developers building in extraordinary measures to keep it from being detected, looking for volume-based anomalies is unlikely to have much success.

Conclusions

With the rapid evolution of botnets toward increasingly stealthy behavior and the staggering numbers of end-hosts already infected by such malware, there is a dire need to develop and deploy techniques to counteract these problems. In this article, we reviewed the latest in botnet behavior and trends to elucidate the shortcomings of traditional approaches that depend on rule-based and/or volume-based detection. Bots and botnets are able to evade anomaly detection in part because they are polymorphic in nature and thus are considered a new vulnerability with every new sighting; their communication behaviors deliberately mimic that of normal end-hosts, and thus they stay below detector threshold settings.

As a result, we analyze the behavior of real Intel enterprise end-host background traffic and contrast it to real botnet C&C channel activity. Consequently, we are able to develop and present the Canary end-host detector, designed to root out the botnet command and control channel by tracking the persistence of a node’s relationships with destination hosts, and the commonality of persistence across multiple peers—both fairly stable properties of non-botnet traffic. The strength of these methods requires no *a priori* knowledge of the botnets that are to be detected, nor do they require traffic payload inspection.

References

- [1] “An Inside Look at Botnets.” Paul Barford and Vinod Yegneswaran. In *Series: Advances in Information Security*, Springer, 2006.
- [2] Symantec. “2H 07 Threat Horizon Report.”
- [3] *USA Today*. “Botnet scams are exploding.” March 17, 2008. At <http://www.usatoday.com/money>
- [4] Damballa. “Damballa announces discovery of Kraken BotArmy,” April 7, 2008. At <http://www.damballa.com>
- [5] F-Secure. “Calculating the Size of the Downadup Outbreak.” January 16, 2009. At <http://www.f-secure.com>
- [6] F. Giroire, J. Chandrashekar, G. Iannaccone, D. Papagiannaki, E. Schooler, and N. Taft. “The cubicle vs. the coffee shop: Behavioral modes in enterprise end-users.” In *Proceedings Passive and Active Measurement Conference (PAM’08)*, Springer Verlag Lecture Notes in Computer Science, pages 202-211, Volume 2979, April 2008.
- [7] D. Dash, B. Kveton, J. M. Agosta, E. Schooler, J. Chandrashekar, A. Bachrach, and A. Newman. “When gossip is good: distributed probabilistic inference for detection of slow network intrusions.” In *Proceedings of the 21st National Conference on Artificial Intelligence, (AAAI’06)*, pages 1115-1122, July 2006.
- [8] Bro. At <http://www.bro-ids.org>

Acknowledgments

The development of the Canary detector was a collaborative research effort with Frederic Giroire, Nina Taft, and Dina Papagiannaki.

Author Biographies

Jaideep Chandrashekar is a Research Scientist at Intel in Santa Clara, CA. His general area of interest is communication networks and distributed systems. In particular, he has worked on Internet and end-host security, traffic measurements and analysis, and Internet routing. His recent work has focused on building security solutions that adapt to individual traffic patterns and distributed anomaly detection mechanisms; and he has investigated the energy footprint associated with network traffic. He joined Intel research in 2006 after receiving a Ph.D. from the University of Minnesota. His e-mail is jaideep.chandrashekar at intel.com.

Carl Livadas is a Research Scientist at Intel Labs. He is currently working on the Distributed Detection and Inference (DDI) project; a cyber-security project focusing on collaborative techniques among overlay peers to promptly and accurately detect malicious behavior. His current research interests include peer-to-peer systems, content-based networking, and cyber security. Prior to joining Intel, Carl worked at BBN Technologies on several cyber-security projects, such as Zombiestones, IPSPOOR, Stingray, and STARLITE. Zombiestones involved the network-based detection and identification of IRC-based Botnets. IPSPOOR involved a simple, light-weight, and effective router-based solution to the problem of IP packet traceback. Stingray involved the design and implementation of a network-based insider threat detection and investigation system. Finally, STARLITE involved the development of novel stepping-stone detection techniques. Carl received his Ph.D. degree in Electrical Engineering and Computer Science from the Theory of Distributed Systems (TDS) group at the Laboratory for Computer Science at MIT. His Ph.D. work involved applying formal techniques to model, analyze, and design retransmission-based reliable multicast protocols. Prior to this work, Carl worked on formally modeling and verifying the correctness and safety of hybrid systems, such as collision avoidance systems for commercial aircraft and autonomous vehicles. His e-mail is clivadas at alum.mit.edu.

Steve Orrin is Director of Security Solutions for Software Pathfinding and Innovation, a part of the Software and Services Group at Intel Corporation, and is responsible for security platforms architecture and security strategy and product direction. Steve joined Intel as part of the acquisition of Sarvega, Inc. where he was their CSO. Steve was previously CTO of Sanctum, a pioneer in Web application security. Prior to joining Sanctum, Steve was CTO and co-founder of LockStar, Inc. and SynData Technologies, Inc. Steve was named one of InfoWorld's Top 25 CTOs of 2004 and is a recognized expert and frequent lecturer on enterprise security. He has also developed several patent-pending technologies covering user authentication, secure data access, and steganography, and he has one issued patent in steganography. Steve is a member of the Information Systems Audit and Control Association (ISACA), the Computer Security Institute (CSI), the International Association of Cryptographic Research (IACR), and he is also a co-founder of WASC (Web Application Security Consortium) and a co-founder of the SafeSOA task force. He participates in several OASIS, and AFEI working groups. His e-mail is steve.orrin at intel.com.

Eve Schooler joined Intel in 2005. She is a Principal Engineer at Intel Labs. Presently she leads the Distributed Detection and Inference (DDI) project, an effort that focuses on collaborative anomaly detection in large-scale networks and that, more broadly, promotes the adoption of an end-host correlation framework that leverages the idea of measurement everywhere. Eve obtained a B.S. degree from Yale University, an MS degree from UCLA, and a Ph.D. from Caltech, all in Computer Science. Her broad interests lie at the intersection of distributed systems, networking, and scalable group algorithm design. Interested in protocol standards, Eve served on the Transport Directorate of the IETF, co-founded and co-chaired the IETF MMUSIC working group for many years, and is a co-author of the SIP protocol that is widely used for Internet telephony. Prior to Intel, she held positions at Apollo Computer, Information Sciences Institute, AT&T Labs-Research, and Pollere LLC. Her e-mail is eve.m.schooler at intel.com.

Copyright

Copyright © 2009 Intel Corporation. All rights reserved.

Intel, the Intel logo, and Intel Atom are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.