Characterizing the Remote Control Behavior of Bots

Elizabeth Stinson and John C. Mitchell

Department of Computer Science, Stanford University, Stanford, CA 94305
{stinson, mitchell}@cs.stanford.edu

Abstract. A botnet is a collection of bots, each generally running on a compromised system and responding to commands over a "command-and-control" overlay network. We investigate observable differences in the behavior of bots and benign programs, focusing on the way that bots respond to data received over the network. Our experimental platform monitors program behavior, considering data received over the network to be tainted, applying library-call-level taint propagation, and checking for tainted arguments to selected system calls. As a way of distinguishing locally-initiated from remotely-initiated actions, we capture and propagate "cleanliness" of local user input (as received via the keyboard or mouse). Testing indicates behavioral separation of major bot families (agobot, DSNXbot, evilbot, G-SySbot, sdbot, Spybot) from benign programs with low error rate. Our prototype implementation monitors execution of an arbitrary Win32 binary.

1 Introduction

Botnets have been instrumental in distributed denial of service attacks, click fraud, phishing, malware distribution, manipulation of online polls and games, and identity theft [2, 24, 30, 40, 49]. As much as 70% of all spam may be transmitted through botnets [4] and as many as ¼ of all computers may be participants in a botnet [53]. A bot master (or "botherder") directs the activities of a botnet by issuing commands that are transmitted over a command-and-control (C&C) overlay network. Some previous network-based botnet detection efforts have attempted to exploit this ongoing C&C behavior or its side effects [3, 6, 40]. Our work investigates the potential for host-based behavioral bot detection. In particular, we test the hypothesis that the behavior of installed bots can be characterized in a way that a way that distinguishes bots from innocuous processes. We are not aware of any prior studies of this topic.

Each participating bot independently executes each command received over the C&C network. A bot command takes some number of parameters (possibly zero) – each of a particular type – in some fixed order. For example, many bots provide a web-download command, which commonly takes two parameters; the first is a URL that identifies a remote resource (typically a file) that should be downloaded and the second is the file path on the host system at which to store the downloaded data. A botnet, therefore, constitutes a remotely programmable platform with the set of commands it supports forming its API.

Many parameterized bot commands are implemented by invoking operating system services on the host system. For example, the web-download command connects to a target over the network, requests some data from that target, and creates a file on the host system; all of these actions (connect, network send and receive, and file creation) are performed via execution of system calls. Typically, a command's parameters provide information used in the system call invocation. For example, the `connect` system call takes an IP address argument, which identifies the target host with which a connection should be established. Implementations of the web-download command obtain that target host IP from the given URL parameter. Thus, execution of many parameterized commands causes system call invocations on arguments obtained from those parameters.

In this paper, we test the experimental hypothesis that the external control of bots through parameterized commands separates bot behavior from normal execution of innocuous programs. We postulate that a process exhibits external control when it uses data received from the network (an untrusted source) in a system call argument (a trusted sink). We test our hypothesis via a prototype implementation, BotSwat, designed for the environment in which the vast majority of bots operate: home users' PCs running Windows XP or 2000 [40]. BotSwat can monitor execution of an arbitrary Win32 binary and interposes on the run-time library calls (including system calls) made by a process. We consider data received over the network to be tainted and track tainted data as it propagates via dynamic library calls to other memory regions. We identify execution of parameterized bot commands when tainted arguments are supplied to select *gate functions*, which are system calls used in malicious bot activity.

Our experimental results suggest that the presence of network packet contents in selected system call arguments is an effective indicator for malicious Win32 bots, including tested variants of agobot, DSNXbot, evilbot, G-SySbot, sdbot, and Spybot. Bots from these families constitute 98.2% of malicious bots seen in the wild [24]. While these bots may implement commands in significantly different ways, similarities in the way they respond to external control allow a single approach to identify them. Additionally, the thousands of variants of each such family generally differ in ways that will not affect our ability to detect them; this is in contrast to traditional anti-malware signature scanners which may require a distinct signature for each variant [58]. Moreover, our generic approach does not rely on a particular command-and-control communication protocol (e.g., IRC) or botnet structure (e.g., centralized or peer-to-peer).

Since our prototype implementation only has visibility into memory-copying calls made via a DLL, we introduce strategies to counteract the effects of *"out-of-band"* memory copies – those which occur outside of the interposition mechanism. In particular, we perform *content-based tainting*, which considers a memory region tainted if its contents are identical to a known tainted string. We also introduce *substring-based tainting*, whereby a region will be considered tainted if its contents are a substring of any data received by the monitored process over the network. Use of these strategies allows us to effectively identify bot behavior even when all of the bot's calls to memory-copying functions occur out-of-band, which may be the case if the bot is built to statically link in memory copying functions. A consequence of BotSwat's use of library-level taint propagation is that bots could apply out-of-band

encryption functions (e.g., XOR) to network data and consequently defeat detection by the prototype implementation. This is a limitation of our current testing platform rather than a deficiency in the characterization of bot remote-control behavior. Our testing of versions of agobot, which encrypt C&C communications via dynamic calls to the OpenSSL library, indicates that remote control behavior can still be identified (even when communications are encrypted), given visibility into the cryptographic function calls. Current botnet C&C communications tend to be unencrypted [30].

While many bot actions, such as establishing network connections, creating files, and executing programs, may also be performed by benign applications, we are able to separate bot behavior from the behavior of benign programs by distinguishing between actions that are initiated locally and those which are initiated remotely by a botnet controller. We tested benign programs that exhibit extensive network interaction and that are typical to the target environment (home-user PCs). Early testing of benign programs revealed that a benign program may use some tainted value in a system call argument as a result of local user input. To account for this phenomenon in our experimental assessment, we designed and implemented a user-input module that identifies data values resulting from local user input as received via the keyboard or mouse. These *"clean strings"* are used to identify instances of local control. Our testing of eight benign programs over a variety of activities common to those applications resulted in eight total flagged behaviors (five different) whereas testing six bots resulted in a total of 202 flagged behaviors (18 different).

In section 2, we provide background information on bots. Section 3 describes our experimental method and section 4 details our prototype implementation. Our experimental results are given in section 5. We discuss the potential for and challenges to applying our findings for real-time host-based bot detection in section 6. Section 7 describes related work and section 8 provides concluding remarks.

2 Bots and Botnets

A botnet is a network of compromised machines that can be remotely controlled by a bot master over a command-and-control (C&C) network. Currently, IRC is the most common C&C communication protocol [6]. Individual bots connect to a rendezvous point – commonly an IRC server and channel, access to which may require password authentication – and await commands. Malicious bots are mostly useful in the aggregate – as a general-purpose computing resource – and proportionally so.

2.1 Bot Families and Variants

The Honeynet Project identifies four main Win32 bot families: (1) the agobot family – the most well known; (2) the sdbot family – the most common; (3) DSNXbot; and (4) mIRC-based bots [40]. A family is "a new, distinct sample of malicious code", whereas a variant is "a new iteration of the same family, one that has minor differences but that is still based on the original" [52]. Variants may be created by augmenting the functionality of a bot (e.g., adding a new exploit for the bot to use in spreading) or by applying "packing transformations" (such as compression and

Figure 1: Capabilities of tested bots and success of our testing method

capability	ago	DSNX	evil	G-SyS	sd	Spy
change C&C server	√	√		√	√	√
create/manage clone		√		√	√	
clone attacks		√				
create spy				√	√	
kill process	√			√		√
open/execute file	√	√		√	√	√
keylogging		√				√
create directory						√
delete file/directory		√				√
list directory		√				√
move file/directory						√
DCC send file		√				√
act as http server						√
create port redirect	√	√		√	√	√
other proxy	√					
download file	√	√		√	√	√
DNS resolution	√			√	√	
UDP/ping floods	√		√	√	√	
other DDoS floods	√			√		√
scan/spread	√	√		√	√	√
spam	√					
visit URL	√			√	√	

Shaded cells represent activities detected by BotSwat.

encryption) to a bot binary [52, 58]. We tested at least one variant from each of the first three major Win32 bot families (agobot, sdbot, and DSNXbot) as well as evilbot and Spybot. Since bots in the wild may link in C library functions statically or dynamically, we tested bots under both conditions. Data from McAfee suggests that bots from the agobot, sdbot, and Spybot families collectively constitute 98.2% of known variants (as of June 2005). Signature-based methods may require analysis and generation of a distinct signature for each such variant [58]. Moreover, evidence suggests that malware writers may be able to infer the byte sequences used to identify malware variants and thus evade detection via applying targeted transformations [50].

2.2 Bot Capabilities and Commands

In general, bot commands can be categorized as performing one or more of: process management, file management, bot management, network interaction, and local spying. Figure 1 provides a summary of some of the functionality exported by the tested bots. The shaded cells represent activities that are detected by BotSwat as described in section 5. Note that, of the 22 different bot activities listed, 21 are implemented as parameterized commands by each of the bots that provides that

Figure 2: The number of system calls invoked during successful execution of commands

	ago	DSNX	evil	G-SyS	sd	Spy
Total # syscalls invoked by all commands	591	145	5	187	173	202
Total … invoked by candidate commands	417	114	5	122	110	145

capability. The exception is keylogging, which – for both of the bots that perform it – logs the captured keystrokes to a file whose name is statically configured. This chart reflects the bot versions we tested; different variants from each of these families may export more or less functionality.

2.2.1 Candidate Commands

Since our characterization of bot behavior exploits the fact that command parameters are often used in system call arguments, we identify *candidate commands* as those which take at least one parameter that is subsequently used (in whole or in part) in an argument to a critical system function. *Non-candidate commands* take no parameters or their parameters have only "local meaning" to the bot and thus are not subsequently used in any system-wide function.

Naturally, we would like to understand how much of a bot's remote-control behavior can be attributed to candidate commands. We therefore counted the number of distinct system calls invoked during a successful execution of each bot command and summed these values across all commands and across candidate commands. These values are provided in figure 2. We obtained the number of system calls invoked during successful execution of a bot command through inspection of bot source code. We did not include in these tallies memory- or string-copying functions or functions which convert between endianness formats (e.g., `htonl`).

The number of system calls invoked by a bot's candidate commands accounts for around 64 to 79% of the system calls invoked by all of the bot's commands. Interestingly enough, the non-candidate commands that cause the highest number of system call invocations generally perform beneficial tasks (from the perspective of the compromised host). For example, for agobot, of the 174 system call invocations over execution of all non-candidate commands, 26 of these occur as a result of executing commands to uninstall the bot (`bot.removeall` and `bot.removeallbut`) and 25 of these 174 occur as a result of executing the `bot.secure` command, which patches up the compromised host's vulnerabilities. Thus, approximately 30% of system call invocations over all of ago's non-candidate commands occur as a result of executing commands beneficial to the compromised host. Similar patterns also held for G-SyS and sd.

3 Experimental Method

We developed a host-based method that identifies instances of external control, whereby a process uses data it received from an untrusted source in a system call argument without having received intervening (local) user input implicitly or explicitly agreeing to this use. We describe the required components generally in

order to underscore the notion that BotSwat is merely one implementation of this method; alternative approaches to tracking taintedness, e.g., may be substituted.

Tainting Component This component identifies when untrusted data is received by the system (taint instantiation) and tracks that data as it propagates to other memory regions (taint propagation). For our method, taint instantiation occurs upon network receive, and taint propagation keeps track of memory regions to which tainted data is written. This component exports an interface that enables querying whether a particular memory region is considered tainted.

User Input Component This purpose of this component is to identify actions that are initiated by the local application user. A primary challenge in designing this component is to identify the data values corresponding to mouse input events where this mapping (from event to value) is heavily application-dependent and not typically exposed (i.e., available via a library call). This component exports an interface that enables learning whether a data value or memory region is considered clean or whether a syscall invocation is likely the result of user input.

Behavior-Check Procedure Triggered by invocation of selected system calls, this procedure queries the tainting and user-input components to determine whether to flag the invocation as exhibiting external control. For example, this procedure might flag an invocation on an argument that contains more tainted than clean data.

4 Implementation

This section describes the interposition approach and the tainting, user-input, and behavior-check instrumentation we use to evaluate our hypothesis.

4.1 Library and System Call Interposition

We use the detours library provided by Microsoft Research for library- and system-call interposition [9, 10]. Our platform consists of a set of functions that we want to interpose upon, a replacement function for each, and a mechanism for performing interposition. The replacement functions contain the tainting, user-input, and behavior-check instrumentation. This platform is packaged as a DLL that can be injected into a target process upon creation of that process.

4.2 Tainting Module

Our tainting module defines taintedness as a property of certain memory addresses, strings, and numeric values. The module operates dynamically at the library call level and considers input received over the network tainted. Taint propagation functions include those which copy memory from a source to a destination buffer (e.g.,

`memcpy`), convert a buffer's contents to a numeric value (e.g., `atoi`), or convert one numeric value to another (e.g., `htons`).

As a result of out-of-band memory copies, our mechanism may possess one of two flawed views regarding a particular memory region. In particular, if a destination region D is written to with tainted data via an out-of-band operation, we will not know that D should be considered tainted. Our belief that D does not contain tainted data is a *false negative*. Similarly, a tainted region T may be written to via an out-of-band operation with untainted data; in this case, our belief that T is tainted is a *false positive*. We introduce techniques to reduce false negatives (content-based and substring-based tainting) and false positives (content-matching).

There are three conditions under which a region may be considered tainted: address-based, content-based, and substring-based. Under *address-based tainting*, a memory region is considered tainted only if its address range overlaps with that of a known tainted region. With *content-based tainting*, a memory region M will be considered tainted if M's contents are identical to a known tainted string. Under *substring-based tainting*, a memory region will be considered tainted if its contents are a substring of any data received over the network by this process.

To reduce false positives, we perform *content-matching*: for a believed-to-be-tainted memory region M, before taking any action on the basis of M's supposed taintedness, we confirm that M's contents match the relevant portion of the network receive buffer from which M allegedly descended. The information needed to perform such a comparison (an identifier of the ancestor network receive buffer, the offset into that buffer from which this tainted data descended, the number of bytes of tainted data, etc.) is contained in the data structure describing a tainted memory region.

The tainting module may run in one of two modes, which differ in the conditions used to determine taintedness. Under *cause-and-effect propagation*, a memory region is considered tainted if the address-based or content-based conditions hold. Under *correlative propagation*, a memory region will be considered if any of the three conditions holds. Consequently, these modes differ in the amount of resilience provided against out-of-band copies. Cause-and-effect propagation was designed for the case where the majority of memory-copies made by a monitored program are visible to the interposition mechanism. We refer to this as cause-and-effect propagation since, in applying it, there is a tight causal relationship between receipt of some data over the network and use of that data in a system call argument. That is, we can point to a sequence of memory copies from a network receive buffer to a system call argument buffer. Correlative propagation, on the other hand, was designed for the case where all memory copies occur out of band – such as can occur when a bot statically links in C library functions. This mode is referred to as correlative propagation since, in applying it, we are ultimately identifying when data received over the network correlates to that used in system call arguments.

Once we determine that a source argument is tainted – via applying the appropriate conditions, given the mode, and performing the content-matching checks – taint propagation proceeds in the following way. We ensure the buffer's contents exist as a tainted string and also that its address range is a known tainted region. If the taint propagation function copies some portion of this source buffer to a destination buffer (e.g., `strncpy`), the corresponding portion of the destination region is transitively marked tainted. If, on the other hand, the taint propagation function converts the

source buffer to a numeric value (e.g., `inet_addr`), we add the numeric result to our collection of tainted numbers. Finally, if the taint propagation function converts one number to another (e.g., `htons`), if the source number is tainted, we add the destination value to our set of tainted numbers as well.

4.3 User Input Module

Our implementation tracks local user input as received via the keyboard or mouse and considers subsequent use of such clean data, such as in a system call argument, innocuous. Obtaining the data value corresponding to a keystroke is generally straightforward as the system generates a message in response to keyboard input for the target application identifying the key or character. Our implementation monitors such messages and creates, for each line of keyboard input, a clean string consisting of the previously input characters.

Obtaining the data values or actions corresponding to a mouse input event is more challenging as the system will generate, upon receipt of such an event, a message which merely identifies the target window, type of event (e.g. left button down), and coordinate pair within that window at which the event occurred. Thus, the actual data value corresponding to such an event is application-defined and not available via a library call. Our implementation addresses this opacity via exploiting locality of reference.

For a Windows input event E, the application that received E calls `DispatchMessage` to pass E to the application window procedure responsible for handling it. This window procedure must process E prior to returning from `DispatchMessage` [51] and may invoke system calls in its processing. We find that – because of the semantics of `DispatchMessage` – data values relating to an input event are generally referenced by interposed-upon functions during execution of this library call. Thus, upon entry to `DispatchMessage` and until return from it, we add any string referenced by any interposed-upon function to our collection of clean strings. Therefore, while we may not be able to determine the specific data value or action that corresponds to a mouse input event, we may be able to obtain a set of data values that in practice contains the correct one.

4.4 Behavior-Check Procedure

Our ability to identify bot behavior relies in part on our selection of appropriate system calls and their arguments to check for taintedness and cleanliness. The collection of bot capabilities (figure 1) informed our selection of system calls (gates) and their particular arguments (sinks); these are described below. The algorithm is as follows. If the sink type is numeric, we query the tainting module to determine whether that argument value is tainted; if so, we flag the invocation; if not, we pass control to the system call. If the sink type is a data buffer, we query the tainting module to determine whether some portion of that buffer is tainted. If not, we cease processing this invocation, passing control to the system call. If so, we query the user-input module about this same argument. If the user-input module indicates that no

Figure 3: Detected behaviors; gate functions for each behavior; sink arguments for gate

	Behavior	API function (gate)	Sink argument [argument #]
B1	tainted open file	NtOpenFile	filename [2]
B2	tainted create file	NtCreateFile	filename [2]
B3	tainted program execution	CreateProcess{A,W}; WinExec	prog name [0], cmd line [1]; prog name [0]
B4	tainted process termination	NtTerminateProcess	process name, PID [0]
B5	bind tainted IP	NtDeviceIoControlFile	IP [6]
B6	bind tainted port	NtDeviceIoControlFile	port [6]
B7	connect to tainted IP	connect; WSAConnect	IP [1]; IP [1]
B8	connect to tainted port	connect; WSAConnect	port[1]; port [1]
B9	tainted send	NtDeviceIoControlFile; SSL_write	send buffer [6]; send buffer [1]
B10	derived send	NtDeviceIoControlFile; SSL_write	send buffer [6]; send buffer [1]
B11	sendto tainted IP	sendto; WSASendTo	IP [4]; IP [5]
B12	sendto tainted port	sendto; WSASendTo	port [4]; port [5]
B13	tainted set registry key	NtSetValueKey	value name [1]
B14	tainted delete registry key	NtDeleteValueKey	value name [1]
B15	tainted create service	CreateService{A,W}	service name [1]; binary path name [7]
B16	tainted open service	OpenService{A,W}	service name [1]
B17	tainted HttpSendRequest	HttpSendRequest{A,W}	host, path, and referrer [0, 1]
B18	tainted IcmpSendEcho	IcmpSendEcho	IP [1]

portion of the argument is clean, then the invocation is flagged. If, on the other hand, that module indicates that some portion of the argument is clean, this procedure will flag the invocation only if the argument contains more tainted than clean data. For a character buffer, the portion tainted is the number of bytes of tainted data in that buffer and the portion clean – the number of bytes of clean data in the buffer.

A *behavior* is a general description of an action that may be detected via checking particular arguments for one or more system calls. In some cases, the same gate function may be instrumented to detect multiple behaviors. For example, `NtDeviceIoControlFile` is invoked by both `bind` and `send` and thus behaviors pertaining to port binding and network send entail checking arguments to this system call. Conversely, in some cases, multiple library functions are instrumented to check for a single behavior. Figure 3 contains the complete list of behaviors, the gate functions for each behavior, and the sink arguments for each identified gate function.

Two behaviors (tainted send and derived send) require a bit more explanation. *Tainted send* occurs when data received over one connection (or socket) is sent out on another. Most commonly, tainted send was exhibited by two command types: redirects and those that generate network messages. When the bot is acting as a proxy (port redirect), it echoes out on a second socket what it heard on the first. For commands that generate network messages, the data heard on the first socket specifies the parameters to send on the second (e.g., the received URL identifies the file path to request in an outgoing HTTP GET message). Since an application may commonly

receive and send certain fixed strings over a variety of connections, we do perform content-based or substring-based tainting for such strings. The set of such strings is small, application-specific, and generally consists of protocol header fields; e.g., a browser's set includes "HTTP/1.1" and "Accept-Range". Consequently, the tainted send behavior is not flagged for transmission of routine messages that do not otherwise contain tainted data. *Derived send* occurs when we obtain from some system call *f* on some tainted input `X` some value `Y` (where `X` != `Y`) then subsequently send `Y` out on any socket. Various data leaking commands match derived send, such as those that take a parameter identifying a registry key and return its value.

5 Experimental Evaluation

This section provides the results of testing our experimental hypothesis that the remote control behavior of bots can be detected via checking selected system calls for tainted arguments. In order to determine the usefulness of this characterization of external control, we compare the effects of detected commands to those of all commands. Finally, we measure whether benign programs exhibit external control.

5.1 Terminology

For each malicious bot, we tested execution of its candidate commands, as in 2.2.1. When BotSwat *flags* a system call invocation, we say that a behavior is *detected*. If flagging this invocation is incorrect, we refer to this as a *false positive*. Any behavior flagged for a benign program is considered a false positive. If BotSwat fails to flag a system call invocation on an argument that contains data received over the network (most likely because BotSwat does not know that this argument should be considered tainted), we say a behavior is *exhibited* but not detected and refer to this as a *false negative*. A command is detected when BotSwat correctly flags at least one behavior exhibited by that command.

5.2 Bot Results

In summary, we found that the external or remote control behavior of bots can be measured by identifying system call invocations which use tainted parameters. Moreover, the effects of a bot's detected commands account for the majority of the effects of all of a bot's commands (where effects are measured via number of system call invocations). Also, bots in general exhibit a great volume and diversity of behaviors. Figure 4 provides a summary of our test results. Row 1 identifies the total number of commands provided by each of the tested bots. The number of those commands that take at least one parameter that is subsequently used (in whole or in part) in a critical system function is provided in row 2. The 3rd row gives the number of candidate commands that were detected using cause-and-effect propagation (C&E) for bots built with C library functions dynamically linked in (DYN). The last row shows the number of candidate commands detected using correlative propagation

Figure 4: Summary of bot-command detection

	ago	DSNX	evil	G-SyS	sd	Spy
# commands	88	28	5	56	50	36
# candidate commands	36	14	5	26	20	15
# commands detected (DYN, C&E)	33	14	N/A	26	20	15
# commands detected (STAT, CORR)	31	10	5	12	12	15

(CORR) on bots built with statically linked in C library functions (STAT). We did not have a version of evilbot which dynamically linked in C library functions. All tested bots either primarily or exclusively use C library functions for memory copying.

5.2.1 Detection of Commands on Dynamically-Linked Bots

The best detection occurs under cause-and-effect propagation on dynamically-linked bots, since these conditions provide the best visibility into the bot's use of data received over the network. Only three total candidate commands were not detected in this mode: agobot's `harvest.registry` and scanning commands. Agobot's scanning commands use a transformation of a received parameter in a system call argument. Taintedness was not propagated across this transformation operation and thus the `scan.start` and `scan.startall` commands were not detected. In particular, the scan commands take an IP prefix range; ago then generates and scans random IP addresses within that range. Additionally, the same set of commands were detected (and the same behaviors flagged for each command) for agobot whether that bot encrypted command-and-control messages via dynamic calls to the OpenSSL library or not. This supports our claim that detection of remote control is resilient to command encryption, given visibility into the cryptographic function calls.

5.2.2 Detection of Commands on Statically-Linked Bots

Although static linking severely hinders visibility into a bot's use of received data, we were still able to detect many of the bots' candidate commands by correlating received network data to system call arguments. Section 5.2.3 explores the effects of detected vs. undetected commands and gives some evidence that these undetected commands are significantly less harmful than are the detected commands. We also note that that many of the undetected commands rely on the previous execution of a command this *is* detected under these conditions. In particular, three of DSNX's four undetected commands (75%), seven of sdbot's eight (87.5%), and seven of G-SySbot's fourteen (50%) perform clone management; this functionality only makes sense when a clone exists to be managed. The command that creates a clone – for each of these three bots – was detected under STAT, CORR. There were three false positives under this mode; in all cases, the incorrectly flagged behavior was in fact malicious but was not an example of external control.

The candidate commands that were not detected share a common property that could be used to produce even better detection results. Specifically, 24 of the 28 undetected commands use `sprintf` to format the argument buffers passed to system

Figure 5: The number of system calls invoked during successful execution of commands

	ago	DSNX	evil	G-SyS	sd	Spy
Total # syscalls invoked by all commands	591	145	5	187	173	202
Total … by candidate commands	417	114	5	122	110	145
Total … by detected commands (DYN, C&E)	393	114	N/A	122	110	145
Total … by detected commands (STAT, CORR)	386	110	5	99	99	145

calls. The call to this buffer-formatting function was not visible to BotSwat and thus it was not able to infer that the resulting argument buffers contained (among other data) strings received over the network. However, statistical tests that measure how similar an argument buffer is to data received over the network are likely to provide significant gains here.

5.2.3 The Effects of Candidate Commands Relative to All Commands

In addition to measuring success by comparing the number of detected to the number of candidate commands, we can also compare the number of system calls invoked by detected commands to the number of system calls invoked by candidate commands. This gives us better perspective on our results. For example, under correlative propagation on a statically-linked version of G-SySbot, we detect only 12 out of 26 candidate commands (46%), as in figure 4, row 4. However, when we compare the number of system calls invoked by detected (99) vs. candidate (122) commands under these same conditions, as in figure 5, row 4, the results are more encouraging (81%). The same also holds for DSNX and sdbot. This is a consequence of the relative severity of commands we are able to detect under these conditions.

5.2.4 Bots Exhibit Volume and Diversity of Behaviors

For each bot command, we counted the number of distinct behaviors correctly detected in a successful execution of that command. Then we tallied these values across commands, giving us the number of times each behavior was detected for each bot (figure 6). Note that in practice the raw number of detected bot behaviors might be much larger since execution of certain commands may cause the same behavior to be repeatedly flagged. This is the case with execution of denial-of-service commands, which often cause a particular behavior to be flagged with transmission of each denial-of-service packet. We note that the distribution of detected bot behaviors across families is not uniform; e.g., behavior B11 (*sendto tainted IP*) is frequently flagged in agobot but never in DSNXbot and only rarely in G-Sys, sd, and Spybots. Such differences may be leveraged to perform classification of an encountered bot as more likely to be a variant of a particular family.

5.3 Benign Program Results

We tested eight benign applications that exhibit extensive network interaction across a variety of activities typical to these programs. False positives in this context are any instances in which a system call invocation is flagged. This could arise from

Figure 6: Over execution of all of a bot's commands, the number of times each behavior was detected

	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15	B16	B17	B18
ago	5	6	7	2	1	5	14	2	14	1	7	3	1	1	1	1	0	0
DSNX	4	4	2	0	0	1	6	4	8	0	0	0	0	0	0	0	0	0
evil	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G-SyS	1	1	8	0	0	1	8	4	10	1	1	1	0	0	0	0	3	1
sd	1	1	2	0	0	1	8	4	10	1	1	1	0	0	0	0	3	1
Spy	4	5	1	1	0	2	4	3	1	0	1	1	0	0	0	0	0	0
Total	15	17	25	3	1	10	40	17	43	3	10	6	1	1	1	1	6	2

imperfections in our user-input module implementation, which may not be able to infer that a system call invocation is the result of local user input. Alternatively, a benign program may genuinely exhibit external or remote control. There were eight false positives; two for the browser, three for the email client, two for the IRC client, and one for the IRC server. The programs, activities across which their behavior was traced, and results are described below.

5.3.1 Benign Program Testing

We chose applications typical to our target environment that exhibit significant network interaction. We tested a browser (Mozilla firefox), an email client (Eudora), an IRC client (mIRC), an ssh client (putty), FTP clients (WS_FTP and SecureFX), an anti-virus signature updater (Symantec's LiveUpdate, LuComServer_3_0.exe), and an IRC server (Unreal IRCd). Since the majority of systems infected with bots are those of home users (who do not typically run server programs) [48], we tested against only one server program. We note, however, that server programs may, at an abstract level, be designed to respond to certain types of external control (that exerted by the client).

We used the browser to visit a variety of sites, some of which contained linked-in images. Once at a site, we clicked on hypertext links, downloaded files specified by links, saved the web page's contents to a file, executed downloaded programs from within the browser, etc. With the email client, we received, composed, replied to, forwarded, and sent email, including and excluding attachments, and including and excluding HTML. We also saved and executed received attachments from within the email client. We exercised the IRC client over a range of its capabilities: connecting to a server and channel, messaging, DCC file transfer, DCC chat, etc. We used the ssh client to connect to and execute commands on a remote host. Using FTP clients, we connected to and browsed various FTP sites, navigated across directories (alternatively using the mouse and keyboard), and downloaded files. We tested the anti-virus signature updater via establishing a base state with stale virus definitions files then instructing the updater to get the latest AV signatures. Finally, the IRC server was networked to other servers and serviced clients.

We tested each application over all of its activities under four scenarios; each combination of cause-and-effect or correlative propagation and with the user-input

module enabled or disabled. We present the results of running under correlative propagation (which has the most relaxed requirements for taintedness) with the user-input module enabled though note briefly that absent the user-input module our ability to distinguish execution of benign programs from that of malicious bots suffers.

5.3.2 Benign Program Results

Four of the eight false positives occur as a result of the automatic downloading of linked-in images performed in rendering an HTML document. Two of these were exhibited by the browser and two by the email client, both upon receipt of an HTML document containing an <IMG SRC="…"> element. Receipt of such an element causes the browser or email client to request the content specified in the URL, which is the value of the SRC attribute. Also, when the user receives an email with an attachment, Eudora automatically creates a file of the same name (as the received file) in its attachments directory. This causes the tainted open file behavior (B1).

The mIRC client generated two false positives as a result of performing Direct Client Protocol (DCC) file receipt. These false positives reveal limitations in our user-input module implementation. In preparation for DCC file transfer, the file sender provides an IP and port to the recipient via a network message. The recipient then creates a TCP connection to the sender using the specified IP and port. Therefore, behaviors B7 (connecting to a tainted IP) and B8 (connecting to a tainted port) were flagged. Prior to the chat client creating such a connection, however, the client asks the user whether he wishes to perform this operation and will only proceed if the user responds affirmatively. Our user-input module was not able to infer the connection between the user input agreeing to this behavior (via a dialog box) and the values used to create the network connection.

The IRC server repeatedly exhibited the tainted send behavior (B9) – which identifies when data heard over one socket is sent out on another. Clearly this behavior is expected, since the overriding purpose of an IRC server is to participate in a chat network, which entails receiving messages and sharing those with its clients and/or other servers.

5.3.3 Benign Results Discussion

We note that there exists a configuration for each of the browser, email client, and IRC client that would suppress exhibition of six of the eight total false positives. In particular, the browser can be configured to not automatically download linked-in images as can the email client (four behaviors total). Also, the IRC client can be configured to never participate in DCC file receipt (two behaviors). We find it interesting that most of the detected behaviors of benign programs may be known to carry a risk and thus our flagging of these behaviors may not be totally inappropriate. Figure 7 summarizes the exhibition of behaviors across all tested programs. We note that bots exhibit high volume (202 across all bots and all commands, as in figure 6) and great diversity (18 different) of behaviors. By contrast, only eight behaviors total (five different) were flagged over execution of all benign programs even when testing under the most liberal taint propagation mode, correlative. We discuss how one might handle these false positives – if a bot detection system were built based upon our findings – in section 6.

Figure 7: For each tested bot and benign program, the number of distinct behaviors detected (out of 18)

	# distinct behaviors	which behaviors
agobot	16	B1 – B16
G-SySbot	12	B1 – B3, B6 – B12, B17, B18
sdbot	12	B1 – B3, B6 – B12, B17, B18
Spybot	10	B1 – B4, B6 – B9, B11, B12
DSNXbot	7	B1 – B3, B6 – B9
evilbot	1	B3
Eudora	3	B1, B7, B17
Firefox	2	B7, B9
mIRC IRC client	2	B7, B8
Unreal IRCd	1	B9
Putty	0	None
SecureFX	0	None
Symantec AV updater	0	None
WS_FTP	0	None

5.4 Performance Results

Function interception via the detours library imposes an overhead of fewer than 400 nanoseconds per invocation [10]. We measured the overall performance impact of BotSwat's instrumentation via scripting a bot to receive then execute various commands then measured the bot's performance natively and under each of our two propagation modes. These tests indicate that the overall performance overhead is 2.81% when using cause-and-effect propagation and 3.87% under correlative.

6 Potential for Host-Based Bot Detection and Future Directions

The goal of our study is to determine whether the remote control behavior of bots can be identified by checking arguments to selected system calls for taintedness. We looked at bots from families whose variants collectively constitute 98.2% of all bots [52] and were able to identify the vast majority of their remote-control behavior. Benign programs exhibited far fewer and a much less diverse set of behaviors. These results are particularly encouraging in light of the challenges facing traditional signature-based malware detection mechanisms and the need for an effective method to identify bots. In particular, our approach is immune to the implementation differences that commonly distinguish bot variants of a particular family (as in [58]) and even differences that distinguish between bots of different families. Signature-based approaches, by contrast, may require *a priori* analysis of each malware variant (and certainly each family) before they can protect against it [58].

Consequently, our work raises questions about whether behavioral monitoring could provide host-based bot detection and, if so, how. Adequately answering this question is a research problem unto itself. We touch briefly below on some of the challenges to building a bot detection mechanism based on our findings and note that generally the challenges may differ depending upon the assumed attacker power (i.e., user-space or kernel-space access). There are some interesting issues surrounding when to label a process a bot; figures 6 and 7 provide some starting insight.

System Call Interposition Mechanism The challenges here are general, well studied [54], and affect the tainting module, the user-input module, and the behavior-check procedure. Each must be implemented in a way that provides visibility into the actions taken by a monitored process. As a starting point, we note that sophisticated versions of the bots we tested ship with anti-debugger, anti-emulation, anti-virtual machine mechanisms [30]. Also, bots can detect inline function hooks (such as used in our prototype implementation) via examining the first five bytes of a detoured function [8]. Moreover, there are a variety of ways to perform an end-run around these hooks [13]. One solution might be to use kernel-land rather than user-land hooks but even these can be detected and subverted [38]. Moving the IDS to a Virtual Machine Monitor as in [39] is another approach which may be especially promising given hardware-implemented virtualization.

Tainting Challenges and Tradeoffs While lightweight from a performance perspective, library-call level taint propagation may not detect bot behaviors when private (out-of-band) cryptographic functions are used. An assembly-code-level tainting module (such as via an emulator, as in [16]) would eliminate this threat, though currently poses a significant performance penalty. Additionally, tainting implementations commonly provide either an explicit or implicit way to launder tainted data. For example, if taintedness is not propagated across writes to persistent storage or communications between processes, e.g. via pipes, these are avenues for laundering. Designating additional sources as untrusted would require reevaluation of our hypothesis against benign programs to ensure the behavioral distinction holds.

User Input Module Challenges There are two considerations here: functionality and security. Since the meaning of local user input is often heavily application-defined, user input modules that incorporate application-specific information are likely to more precisely and accurately identify data values associated with local user input and possibly even the intended or legitimate uses of such values. This may be particularly helpful if bots migrate from being standalone applications to running as part of some known benign application, such as a bot implemented as a browser extension. Regarding security, there are two types of threats to the user-input module: (1) fooling the module into believing that user input has been received by the process; (2) genuinely obtaining local user input. Regarding the first, depending upon the module's implementation, it may be useful to incorporate a kernel-level component that identifies when local user input events are received. This would prevent user-space spoofing – by the bot – of local user input events as via system calls [55, 56] or calls to `DispatchMessage`. As for (2), the bot writer could create a window and trick the user into providing input, exploiting the module's consequent generation of clean strings.

Whitelisting Benign Program Behaviors Since it is not uncommon for applications to be increasingly extensible, we may wish to monitor known benign programs for bot behavior. For example, a bot implemented as a browser extension could be detected by monitoring execution of the browser. However, we have seen that a browser may generate two behaviors (in its automatic downloading of linked-in images) during legitimate operation. There are several possible responses to this. First, we may refrain from identifying a process as a bot until that process generates some volume and diversity of behaviors. Requiring a process to generate five different behaviors before identifying this process as a bot could be a reasonable tradeoff. An alternative approach would be to whitelist certain behaviors for certain programs. For example, we might suppress flagging the two behaviors generated by the browser when monitoring its execution.

Future Directions Other research questions include the feasibility of: correlating component behaviors to identify execution of high-level commands; designing a more fine-grained user-input module; identifying a bot's likely ancestry (i.e., which family this instance is a variant of); combining our host-based, behavioral bot detection with network-based approaches; and applying this same characterization to other malware which exhibit remote-control behavior, such as backdoor programs.

7 Related Work

Applications of Tainting Analysis Tainting has been applied statically, dynamically, at a language level, via an interpreter, an emulator, compiler extensions, etc. [1, 14, 16, 19, 20, 32, 43]. Most commonly, security-motivated tainting has been used to identify vulnerabilities in or exploitations of non-malicious programs [1, 16, 19, 20, 32, 43].

Host-Based Intrusion Detection The problem of distinguishing execution of an installed malicious bot from that of innocuous processes differs from that explored by much previous run-time, host-based, anti-malware research, which has focused on identifying when a host program (generally assumed to be non-malicious) has been exploited [5, 16, 17, 34, 44]. While a bot may be spread via leveraging such exploits, monitoring execution of an installed bot using one of these mechanisms will generally not result in the bot being identified as malicious since no exploit of a local host program is entailed in normal bot execution. Other behavior-based research has been done to identify rootkits and spyware [7, 37, 39, 47].

Botnet Detection We are not aware of any existing approaches to behavioral host-based bot detection. Some existing network-based approaches to botnet detection have exploited the ongoing C&C behavior [40] with others focusing on detecting secondary effects of botnets [6], such as increased rate of Dynamic DNS queries [3], or mitigating the effects of a botnet at a DDoS victim [35]. Other approaches include content-based network intrusion detection signatures [21] and using heuristics to identify certain IRC channels as likely rendezvous points. Clearly, content-based signature approaches are fragile since bot writers have total control over the content being communicated to and from bots. Similarly, approaches based on identifying a botnet at the rendezvous point are only as effective as the coverage of rendezvous

points is complete. Finally, while there is clearly value in mitigating the effects of botnets at a DDoS target, botnets pose a significant threat even excepting this functionality [2, 24, 30, 40, 49].

8 Conclusions

Botnets present a serious and increasing threat, as launching points for attacks including spam, distributed denial of service, sniffing, keylogging, and malware distribution [2, 28]. Our work explores whether the execution of malicious bots can be distinguished from that of innocuous programs. We provided a characterization of the remote control behavior of bots, identified the fraction of current bot remote-control behavior covered by this characterization, built a prototype implementation, and evaluated our hypothesis against six bots from five different families and a variety of benign applications typical to the target environment. We introduce techniques, such as content-based and substring-based tainting, that enable us to effectively identify a bot's remote control behavior even when visibility into the memory-copying calls made by a bot is severely limited.

Experimental evaluation suggests that the external or remote control behavior of bots can be detected by identifying system call invocations which use tainted parameters. We see that the effects of a bot's candidate commands (as measured via number of system call invocations) constitute the vast majority of the effects of all of a bot's commands. We also see that bots in general exhibit a great volume and diversity of behaviors. Finally, we note that, when we track local user input and sanitize subsequent uses of it, benign programs relatively rarely exhibit the external control behavior that we're measuring.

Our prototype implementation detects execution of a range of parameterized bot commands, taking advantage of the fact that bots are programmable platforms responding to commands received from a botnet controller over a command-and-control network. Since most known bot activity is associated with commerce in programmable distributed bot platforms, our method works because it detects precisely the functionality that makes bots most useful to their installers.

Acknowledgements. Thanks to the detours team at MSR and Galen Hunt in particular for helpful insights into detours. We are also grateful to David Dagon at Georgia Tech, who provided versions of agobot, and Andrew Sakai for testing assistance. Thanks to Tal Garfinkel and Adam Barth for helpful feedback. We thank Wenke Lee for extensive and valuable feedback on our work and on its presentation.

References

1. A. Turoff. Defensive CGI Programming with Taint Mode and CGI::UNTAINT
2. B. Schneier. How Bot Those Nets? In Wired Magazine, July 27, 2006.
3. D. Dagon. Botnet Detection and Response: The Network Is the Infection. In *Operations, Analysis, and Research Center Workshop (OARC)*, July, 2005.
4. D. Ilett. Most spam generated by botnets, says expert. ZDNet UK, Sept 22, 2004.

5. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May, 2001.
6. E. Cooke, F. Jahanian, and D. McPherson. The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In *Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, Cambridge, MA, July 2005.
7. E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer. Behavior-based Spyware Detection. In *Proc 15th USENIX Security Symposium*, August 2006.
8. G. Hoglund and J. Butler. Rootkits: Subverting the Windows Kernel. First Edition, Addison-Wesley, Upper Saddle River, NJ, 2006.
9. G. Hunt and D. Brubacher. README file, Microsoft Research Detours Package.
10. G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.
11. G. Nebbett. Windows NT/2000 Native API Reference. First edition, MTP, 2000.
12. I. Guilfanov. DataRescue: Interactive Disassembler Pro - FLIRT Technology.
13. J. Butler et al. Bypassing 3rd Party Windows Buffer Overflow Protection. In phrack Volume 0x0b, Issue 0x3e, Phile #0x0, 7/13/2004.
14. J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the USENIX 13th Security Symposium*, San Diego, CA, August 2004.
15. J. Gulbrandsen. How Do Windows NT System Calls REALLY Work?
16. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed Systems Symposium (NDSS)*, February, 2005.
17. J. Rabek, R. Khazan, S. Lewandowski, and R. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, October 2003.
18. J. Robbins. Bugslayer. In Microsoft Systems Journal, February, 1998.
19. K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc IEEE Security and Privacy Conference*, 2002.
20. Locking Ruby in the Safe http://www.rubycentral.com/book/taint.html
21. LURHQ. Phatbot Trojan Analysis. http://www.lurhq.com/phatbot.html
22. M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-Aware Malware Detection. In *IEEE Symposium on Security and Privacy*, May, 2005.
23. M. Handley and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *USENIX Sec. Symp*, 2001.
24. M. Overton. Bots and Botnets: Risks, Issues, and Prevention. In *Virus Bulletin Conference*, Dublin, Ireland, October, 2005. http://tinyurl.com/2xxbkf
25. M. Pietrek. An In-Depth Look into the Win32 Portable Executable File Format, Parts I and II. In *MSDN Magazine*, February, 2002 and March, 2002.
26. M. Russinovich and B. Cogswell. Examining VxD Service Hooking: Monitoring, alerting, or otherwise changing parts of Windows. Dr. Dobbs Journal, May 1996.
27. M. Russinovich and B. Cogswell. Windows NT System-Call Hooking. In Dr. Dobb's Journal, January 1997. http://www.ddj.com/184410109
28. M. Russinovich. Inside the Native API.
29. M. Russinovich. Sony, Rootkits and Digital Rights Management Gone Too Far.
30. N. Ianelli and A. Hackworth. Botnets as a Vehicle for Online Crime. CERT Coordination Center, December, 2005.

31. NtDeviceIoControlFile Function. http://tinyurl.com/2cj59y
32. perlsec http://perldoc.perl.org/perlsec.html
33. Princeton WordNet. http://wordnet.princeton.edu/perl/webwn?s=system%20call
34. S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on Security and Privacy*, May 1996.
35. S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. In *Network and Distributed System Security Symposium (NDSS)*, Boston, MA, May 2005.
36. S. Labaton. An Army of Soulless 1's and 0's. New York Times, June 24, 2005.
37. Strider GhostBuster Rootkit Detection http://research.microsoft.com/rootkit/
38. T. C. Keong. Defeating Kernel Native API Hookers by Direct KiServiceTable Restoration
39. T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distributed Systems Security Symposium (NDSS)*, San Diego, CA, February 2003.
40. Honeynet Project & Research Alliance. Know your Enemy: Tracking Botnets.
41. The majority of bot code was obtained from: http://tinyurl.com/3y4cfd
42. The Metasploit Project. Windows System Call Table (NT/2000/XP/2003/Vista).
43. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. *Proc. 10th USENIX Security Symp*, 2001.
44. V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc 11th USENIX Security Symposium*, August 2002.
45. E. Parizo. New bots, worm threaten AIM network. SearchSecurity, Dec, 2005.
46. R. Naraine. Money Bots: Hackers Cash In on Hijacked PCs. eWeek, Sept, 2006.
47. W. Cui, R. Katz, and W. Tan. BINDER: An Extrusion-based Break-in Detector for Personal Computers. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, Tuscon, AZ, December 2005.
48. K. Martin. Stop the bots. In The Register, April, 2006.
49. G. Keizer. Bot Networks Behind Big Boost In Phishing Attacks. TechWeb, Nov, 2004.
50. M. Christodorescu and S. Jha. Testing Malware Detectors. In the *Proc. of the International Symposium on Software Testing and Analysis* (ISSTA), July 2004.
51. MSDN Library. Using Messages and Message Queues. http://tinyurl.com/27hc37
52. Symantec Internet Security Threat Report, Trends for July 05–December 05. Volume IX, Published March 2006. http://tinyurl.com/2a5toa
53. W. Sturgeon. Net pioneer predicts overwhelming botnet surge. ZDNet News, January 29, 2007.
54. T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Network and Distributed Systems Symposium*, 2003.
55. LowLevelKeyboardProc Function. http://tinyurl.com/7790
56. LowLevelMouseProc Function. http://tinyurl.com/2favqs
57. SetWindowsHookEx Function. http://tinyurl.com/74f9t
58. Symantec Internet Security Threat Report, Trends for January 06–June 06, Volume X. Published September 2006.