Characterizing Bots' Remote Control Behavior

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 execution of an arbitrary Win32 binary, 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 further 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.

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 [2,18,19,25,33,30]. As much as 70% of all spam may be transmitted through botnets [4] and as many as $\frac{1}{4}$ of all computers may be participants in a botnet [37]. 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 [3,6,25]. 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 distinguishes malicious 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 remote control of bots through parameterized commands separates bot behavior from normal execution of innocuous programs. We postulate that a process exhibits external or remote 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 [25]. 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 [18]. 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 [38]. 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 Dynamically Linked Library (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. These strategies are applied upon calls by a monitored process into *taint propagation functions*, which are DLL functions used to copy or convert the contents of memory. Applying 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 statically links in C library functions.

A consequence of BotSwat's use of library-call-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 [19].

While both bots and benign programs may create files, interact with the network, and execute programs, we are able to separate bot behavior from that of benign programs by distinguishing between remotely-initiated and locally-initiated actions. We tested applications typical to the target environment (home-user PCs) which exhibit extensive network interaction. Early testing revealed that a benign program may use some tainted value in a system call argument as a result of local user input. For example, when a user downloads a webpage via a browser then clicks on a hypertext link therein, the browser will consequently request the content stored at the linked URL. In so doing, the browser will invoke system calls (e.g., `connect`, `send`) on tainted arguments (the URL). If user input were not tracked, this sequence of events would look similar to bot execution of the web download command. 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 Sect. 2, we provide background information on bots. Section 3 describes our experimental method, and Sect. 4 details our prototype implementation. Our experimental results are given in Sect. 5. We discuss the potential for and challenges to applying our findings for real-time host-based bot detection in Sect. 6. Section 7 describes related work and Sect. 8 provides concluding remarks.

2 Bots and Botnets

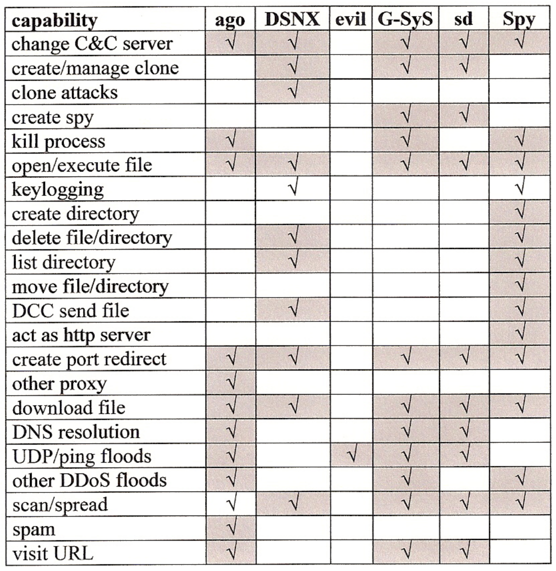
A botnet is a network of compromised machines that can be remotely controlled by a bot master over a command-and-control (C&C) network. Individual bots connect to a rendezvous point – commonly an IRC server and channel, access to which may require password authentication – and await commands.

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 [25]. 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" [36]. Variants may be created by augmenting the functionality of a bot (e.g., adding a new exploit for use in spreading) or by applying "packing transformations" (such as compression and encryption) to a bot binary [36,38]. 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. Data from McAfee suggests that bots from these tested families collectively constitute 98.2% of known variants (as of June 2005) [18]. Since bots in the wild may link in C library functions statically or dynamically, we tested bots under both conditions.

Fig. 1. Bot capabilities



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	√			√	√	

2.2 Bot Capabilities and Commands

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 Sect. 5. Note that, of the 22 different bot activities listed, 21 are implemented as parameterized commands by each of the bots that provides that 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.

Candidate Commands Since our characterization of bot behavior exploits the fact that command parameters are often used in system call arguments, we identify a bot's *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. Our method considers *non-candidate commands*, those which take no parameters or parameters with only "local meaning" to the bot, out-of-scope.

Any bot execution of a received command is an instance in which that bot is being remotely controlled. The remote control behavior associated with a particular command consists of all the actions taken by the bot as a direct consequence of receipt of that command. Not all commands result in an equal amount of remote control behavior; e.g., a command that asks a bot to return its ID (some statically-configured value) to the bot controller entails fewer actions than the described web-download command. We approximate a command's remote control behavior by identifying the number of distinct system calls invoked during a successful execution of that command; these values were obtained through bot source code inspection. A bot's *total potential remote control behavior*, then, is the sum of the remote-control behavior of each of that bot's commands (Table 1, Row 1). Our coverage of that potential can be measured by summing the remote-control behavior of each of a bot's candidate commands (Table 1, Row 2). The complete list of system calls used in the tallies can be found in [52]. The number of system calls invoked by a bot's candidate commands accounts for around 64 to 79% of the system calls invoked over 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); specific examples of this can be found in [52].

Table 1. The number of system calls invoked during successful execution of commands.

	ago	**DSNX**	**evil**	**GSyS**	**sd**	**Spy**
# syscalls invoked over all cmds	591	145	5	187	173	202
# syscalls invoked over candidate cmds	417	114	5	122	110	145

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.

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. Invocations on arguments that contain more bytes of tainted than clean data are flagged.

4 Implementation

This section describes the interposition approach and the tainting, user-input, and behavior-check instrumentation used 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]. 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 its creation. Our implementation consists of approximately 70,000 lines of C++ code and, for the purpose of conducting thorough experiments, may intercept up to 2,200 API functions.

4.2 Tainting Module

Our tainting module operates dynamically at the library-call level and considers data received over the network to be tainted; consequently, network receive functions (e.g., `recv`, `WSARecv`) are instrumented as taint instantiators. 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`). Taintedness can be a property of memory addresses, strings, or numeric values. A total of 172 different functions (enumerated in [52]) were instrumented as taint propagators.

As a result of out-of-band memory copies, our mechanism may possess one of two flawed views regarding a particular memory region. 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 perform content-matching to reduce false positives and content-based and substring-based tainting to reduce false negatives.

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 (where actions include propagating taint or flagging a system call invocation), we confirm that M's contents match the relevant portion of the network receive buffer N from which M allegedly descended. The information needed to perform such a comparison (an identifier of N, the offset into N from which this tainted data descended, the number of bytes of tainted data, etc.) is stored in the data structure describing a tainted memory region.

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 if its address range overlaps with that of a known tainted region. With *content-based tainting*, a memory region will be considered tainted if its 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.

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 process 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 most or 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.

Upon a call to a taint propagation function f, that function's relevant arguments are checked for taintedness via applying the appropriate conditions, given the mode, and performing content-matching. Given a tainted source argument, taint propagation proceeds in the following way. For source buffers, we ensure that the tainted portion of that buffer is a known tainted string and its address range is a known tainted region. If f copies some portion of this source buffer to a destination buffer, the corresponding portion of the destination region is transitively marked tainted. If, on the other hand, f converts the source buffer to a numeric value, we add the numeric result to our collection of tainted numbers. Finally, if the tainted source argument is a number which f converts to another number, we add the destination value to our set of tainted numbers.

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 value corresponding to a mouse input event is more challenging as the system generates, 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. 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; in particular, our goal was to identify when an application was executing code to handle a user-input event. We posited that any data values referenced during execution of such code could be considered clean and that in this way we could infer a set of data values corresponding to a user input event.

For a Windows user input event `E`, an application calls `DispatchMessage` in order to invoke that application's predefined handler for `E`. The handler must process `E` prior to returning from `DispatchMessage` [35] and may invoke system calls in its processing. 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.

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 (Fig. 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, if the argument value is tainted, we flag the invocation; otherwise, we pass control to the system call. While a numeric value will either be considered tainted or not, buffer arguments may contain some number of bytes of tainted and/or clean data. If the sink type is a data buffer which contains no tainted data, control is passed to the system call. Otherwise, we query the user-input module to determine whether that buffer also contains clean data. If not, the invocation is flagged; if so, this procedure will flag the invocation only if the argument contains more bytes of tainted than clean data.

A *behavior* is a general description of an action that may be detected via checking particular arguments for one or more system calls. The same gate function may be instrumented to detect multiple different behaviors. Conversely, multiple library functions may be instrumented to check for a single behavior. Table 2 contains the complete list of behaviors and the gate functions for each behavior. In general, we favored instrumenting lower-level API functions as gates; e.g., instrumenting `NtOpenFile` as a gate enables us to detect all behaviors that entail listing a directory, deleting a file, or replacing a file since the higher-level API functions for these tasks ultimately call into `NtOpenFile`.

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; e.g., when a bot is acting as a proxy, it echoes out on a second socket the data heard on the first. Since an application may commonly receive and send certain fixed strings over a variety of connections, we do not 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 a system call is invoked on some tainted input to obtain a result that is then sent on the network. Various data leaking commands match derived send, such as those which take a parameter identifying a registry key and return its value.

Table 2. Detected behaviors and the gate functions for each behavior.

	Behavior	**gate function**
B1	tainted open file	NtOpenFile
B2	tainted create file	NtCreateFile
B3	tainted program execution	CreateProcess{A,W}
B4	tainted process termination	NtTerminateProcess
B5	bind tainted IP	NtDeviceIoControlFile
B6	bind tainted port	NtDeviceIoControlFile
B7	connect to tainted IP	connect; WSAConnect
B8	connect to tainted port	connect; WSAConnect
B9	tainted send	NtDeviceIoControlFile; SSL_write
B10	derived send	NtDeviceIoControlFile; SSL_write
B11	sendto tainted IP	sendto; WSASendTo
B12	sendto tainted port	sendto; WSASendTo
B13	tainted set registry key	NtSetValueKey
B14	tainted delete registry key	NtDeleteValueKey
B15	tainted create service	CreateService{A,W}
B16	tainted delete service	OpenService{A,W}
B17	tainted HttpSendRequest	HttpSendRequest{A,W}
B18	tainted IcmpSendEcho	IcmpSendEcho{A,W}

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. To determine the utility of this characterization of remote control, we compare the effects of detected commands to those of all commands. Finally, we measure whether benign programs exhibit remote control.

5.1 Bot Experiment Setup

We edited the source code of each bot by altering its C&C parameters such that, when executed, that bot would connect to a C&C server under our control. We then built two versions of each bot: one which dynamically linked in C library functions (DYN) and a second which statically linked these in (STAT). We then executed each bot binary, injecting our DLL into the newly-spawned bot process so as to intercept its API calls (as described in 4.1). We were then able to exercise each bot over its set of commands and monitor the effects of each such command.

5.2 Terminology

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*. We say that a command is detected when BotSwat correctly flags at least one behavior exhibited by that command. Note that many commands exhibit more than one behavior; therefore, a particular command may exhibit a false negative but still be detected.

5.3 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). Bots in general exhibit a great volume and diversity of behaviors. Table 3 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 (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.

Table 3. Summary of bot command detection.

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

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; thus, `scan.start` and `scan.startall` were not detected. Also, the same set of commands was detected (and the same behaviors flagged for each command) for agobot whether that bot encrypted C&C messages via dynamic calls to the OpenSSL library or not. Thus, detection of remote control is resilient to command encryption, given visibility into the cryptographic function calls.

Detection of Commands on Statically-Linked Bots Since all tested bots either primarily or exclusively use C library functions for memory copying, static linking severely hinders visibility into a bot's use of received data. We were still, however, able to detect execution of many of the bots' candidate commands by correlating received network data to system call arguments. We explore below the effects of detected vs. undetected commands and provide some evidence that these undetected commands are significantly less harmful than are the detected commands. 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 not an example of external control.

The candidate commands that were not detected under STAT, CORR 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 calls. The call to this buffer-formatting function was not visible to BotSwat (under STAT) and thus it was not able to infer that the resulting argument buffers contained (among other data) strings received over the network. Statistical tests that measure how similar an argument buffer is to data received over the network may provide significant gains here.

The Effects of Detected Commands Relative to All Commands As discussed in Sect. 2.2, not all commands result in an equal amount of remote control behavior. [1] We find that the commands we are able to detect for each bot – even under STAT, CORR – account for the majority of that bot's total potential remote control behavior. For Spybot, e.g., under STAT, CORR, the number of system calls invoked during execution of detected commands is 145 (Table 4) and during execution of all commands is 202 (Table 1). The same pattern held for all tested bots and is a consequence of the relative severity of commands we are able to detect even under these conditions.

Table 4. The number of system calls invoked during successful execution of candidate and detected commands.

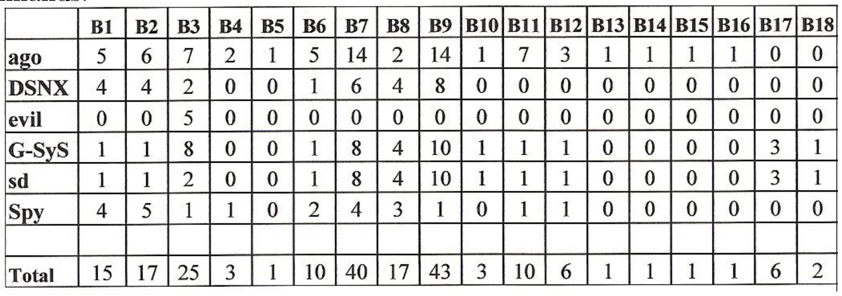
	ago	**DSNX**	**evil**	**GSyS**	**sd**	**Spy**
# syscalls invoked by candidate cmds	417	114	5	122	110	145
# syscalls … detected cmds (DYN, C&E)	393	114	N/A	122	110	145
# syscalls … detected cmds (STAT, CORR)	386	110	5	99	99	145

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 (Fig. 2). It is not uncommon for execution of a single command to result in detection of multiple behaviors. Executing a port redirect command, e.g., generally results in four detected behaviors: binding a tainted port (B6), connecting to a tainted IP (B7), connecting to a tainted port (B8), and tainted send (B9). 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. Such is the case with denial-of-service (DoS) commands, which often

[1] We approximate the remote control behavior associated with a particular command via tabulating the number of distinct system calls invoked in a successful execution of that command. Then the bot's total potential remote control behavior is the sum of these values across all of that bot's commands.

cause a particular behavior to be flagged with transmission of each DoS packet. We note that the distribution of detected behaviors across bot 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.

Fig. 2. The number of times each behavior was detected, over all of a bot's commands.



	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

5.4 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 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.

Benign Program Testing We tested a browser (firefox), email client (Eudora), IRC client (mIRC), ssh client (putty), FTP clients (WS_FP and SecureFX), anti-virus (AV) signature updater (Symantec's LuComServer_3_0.exe), and IRC server (Unreal IRCd). Since the majority of systems infected with bots are those of home users (who do not typically run server programs) [32], 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 containing 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, 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 AV 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.

Benign Program Results We present the results of running under correlative propagation (which has the most relaxed requirements for taintedness) with the user-input module enabled. 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 application to request the content specified in the `SRC` URL. Also, when the user receives an email with an attachment, Eudora automatically creates a file of the same name (as the received file), which 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.

Benign Results Discussion 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. In particular, [53] recommends disabling DCC file receipt so as to avoid malware infection (2 behaviors); the automatic downloading of linked-in images performed by the email client and browser may be exploited to perform DoS attacks [50] (4 behaviors); and email attachments are a known malware propagation vector (1 behavior).

Table 5 summarizes the detection of behaviors across all tested programs. Note that a single run of any such program may exhibit fewer behaviors depending upon the inputs to that particular run-time instance. In general, bots exhibit high volume (202 across all bots and all commands, as in Fig. 2) 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 in Sect. 6. Finally, we acknowledge the limitations of black-box dynamic testing; that is, there may be other inputs to these benign programs that would result in flagging additional behaviors. Similarly, it may be the case that higher fidelity taint propagation (e.g., assembly-code-level tainting) reveals additional behaviors. That said, all programs (malicious and benign) were tested using the same system, and the demonstrated behavioral gap between bots and benign applications under these conditions is dramatic.

Table 5. For each tested program, the number of distinct behaviors detected.

	# distinct behaviors	**which behaviors**
agobot	16	B1 - B16
GSySbot	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	2	B7, B8
Unreal IRCd	1	B9
putty	0	N/A
SecureFX	0	N/A
Symantec AV updater	0	N/A
WS_FTP	0	N/A

5.5 Performance Results

Function interception via the `detours` library imposes an overhead of fewer than 400 nanoseconds per invocation [9]. We measured the overall performance impact of BotSwat's instrumentation via scripting a bot to receive then execute various commands; the bot's performance was measured natively and under each of the two propagation modes. The overall measured performance overhead is 2.81% when using cause-and-effect propagation and 3.87% under correlative.

6 Potential for Host-Based, Behavioral Bot Detection

Signature-based anti-malware mechanisms suffer from several critical limitations, including the inability to detect novel malware instances or obfuscated variants and the need to continuously update their signature sets [34,38]. A recent study found that even the most effective anti-virus vendor failed to detect a significant percentage of malware samples found in the wild [42]. Behavior-based approaches to malware detection provide a powerful alternative: the ability to detect entire classes of malware including previously unseen instances. The primary challenge is to identify a useful behavioral characterization: one which identifies behavior fundamental to a class of malware but which is not generally exhibited by innocuous programs. The data presents a compelling argument that our characterization meets these criteria; the very behavior that makes bots most useful to their installers (their programmability) provides the basis for detection.

Our prototype implementation was designed to test the effectiveness of our behavioral characterization; a secure implementation of our method must be able to detect and differentiate such remote control behavior in a way that is difficult for malware to adaptively evade and subvert. Designing such a system is a research problem unto itself. We highlight some of the fundamental challenges and tradeoffs in building a bot detection mechanism based on our findings.

Process Monitoring Mechanism The mechanism that enables visibility into a process's actions may also be referred to as a *sandbox.* There are two primary design considerations: *visibility*, which refers to the type and granularity of events visible to the sandbox, and *isolation*, which refers to the difficulty of a monitored process to evade or subvert the sandbox. The (user-space) in-line function hooks [9] used in the prototype implementation provide high visibility (as the interposition code runs in the same address space as the monitored application) but very weak isolation [8,10]. Kernel-space system call interposition and Virtual Machine Introspection [24] are additional possibilities.

Tainting Challenges and Tradeoffs Since a malicious bot may evade detection via performing data movement (or data transformation) operations out-of-band, coverage is a critical aspect of the system's security. There appears at present to be a fundamental tradeoff in dynamic tainting modules between coverage and performance; i.e., tainting implementations that provide thorough coverage (as in [12]) exact significant performance penalties. Also, if there are operations across which taintedness is not propagated (e.g., writes to persistent storage or pipes), surely such avenues will be used to launder tainted data. Propagating taint more thoroughly may result in more flagged behaviors and false positives.

User Input Module Challenges There are two types of attacks specific to this component: spoofing user input events and genuinely obtaining user input. Exposure to user-input-spoofing attacks may be minimized by incorporating a kernel-level component that identifies receipt of user input events. The latter attack, however, highlights the fundamental challenge in this module's design. In particular, since the meaning of user input events is inherently application-defined, a user-input module must rely on the application that received a user-input event to implicitly or explicitly identify the semantics of that input. Consequently, if a malicious process is able to legitimately obtain *any* local user input, that process may be able to arbitrarily assign meaning to that input.

System Inputs and Outputs An interesting question is which processes to monitor using the detection mechanism. A reasonable decision may be to not monitor known benign programs. Such a decision would inevitably cause attackers to explore ways in which such known benign programs could be coopted to do the attackers' bidding (as in [50]). In either case, a general decision must be made about when to label something a bot. A reasonable tradeoff may be to require some volume and diversity of behaviors; then a lower threshold more narrowly constrains the attacker's arena but may also result in more false positives. Additionally, one could whitelist certain behaviors known to be generated by particular applications during their legitimate operation (as in Sect. 5.4). A final option may be to identify and flag execution of commands – sequences of correlated behaviors – rather than individual behaviors.

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. [1,11,12,14,15,20,27]. Most commonly, security-motivated tainting has been used to identify vulnerabilities in or exploitations of non-malicious programs.

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 [5,12,13,21,28]. 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 [23,49,7,31]. [31] identifies *extrusions*: stealthy outgoing network connection made by malicious processes. User-intended (legitimate) outgoing network connections include those preceded in time by receipt of user input. A difference between our work and theirs is that, for us, outgoing network connections are only one of 18 behaviors of interest; also, we are interested in the semantics of user-input events, not merely their occurrence at some point in time.

Botnet Detection Host-based approaches include scanning the contents of files and memory for certain byte sequences as well as content-based filtering, which identifies receipt of packets containing known bot-command keywords, as in Norton Intrusion Prevention. Network-based approaches to botnet detection include those which: (a) detect secondary effects of botnets [6,3]; (b) set up honeypots to obtain bot binaries then infiltrate those botnets [25,41,40]; (c) mitigate the effects of a botnet at a DDoS victim [22]; (d) apply content-based Network Intrusion Detection System (NIDS) signatures [16]; (e) apply heuristics to IRC channel traffic to identify likely C&C rendezvous points; (f) identify IRC

NICK messages likely to have been generated by bots [44]; (g) track and correlate various types of NIDS alarms to identify *bot-infection sequences* [43]; (h) perform analysis of flow data to identify suspected bots then likely conversations between such suspected bots and their C&C servers [45]. Challenges for these approaches include: changing the C&C protocol or botnet topology; encrypting or otherwise obfuscating C&C communications; altering the timing of bot-related events and port scanning activity so as to stay below detection thresholds; bots which employ non-worm-like spreading behavior; coverage of C&C rendezvous points; running a botnet entirely within a single administrative domain; etc.

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. 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. Significant challenges remain in the problem of building a secure and robust bot detection system based on these observed behavioral differences.

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 to 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. We are very grateful to our reviewers and to our shepherd for their insightful questions and comments.

References

1. Turoff, A.: Defensive CGI Programming with Taint Mode and CGI::UNTAINT
2. Schneier, B.: How Bot Those Nets? In Wired Magazine, July 27, 2006.

3. Dagon, D.: Botnet Detection and Response: The Network Is the Infection. In Operations, Analysis, and Research Center Workshop, July 2005.
4. Ilett, D.: Most spam generated by botnets, says expert. ZDNet UK, Sept. 22, 2004.
5. Wagner, D., Dean, D.: Intrusion Detection via Static Analysis. In IEEE Symposium on Security and Privacy, May 2001.
6. Cooke, E., Jahanian, F., McPherson, D.: The Zombie Roundup: Understanding, Detecting, and Disrupting Botnets. In Steps to Reducing Unwanted Traffic on the Internet, July 2005.
7. Kirda, E., Kruegel, C., Banks, G., Vigna, G., Kemmerer, R.: Behavior-based Spyware Detection. In Proc. 15th USENIX Security Symposium, August 2006.
8. Hoglund, G., Butler, J.: Rootkits: Subverting the Windows Kernel. First Edition, Addison-Wesley, Upper Saddle River, NJ, 2006.
9. Hunt, G., Brubacher, B.: Detours: Binary Interception of Win32 Functions. In 3rd USENIX Windows NT Symposium, July 1999.
10. Butler, J.: Bypassing 3rd Party Windows Buffer Overflow Protection. In phrack Volume 0x0b, Issue 0x3e, Phile #0x0, 7/13/2004.
11. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding Data Lifetime via Whole System Simulation. In Proc. of the USENIX 13th Security Symposium, August 2004.
12. Newsome, J., Song, D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In Network and Distributed Systems Symposium, February 2005.
13. Rabek, J., Khazan, R., Lewandowski, S., Cunningham, R.: Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In Proc. of the ACM Workshop on Rapid Malcode, October 2003.
14. Ashcraft, K., Engler, D.: Using programmer-written compiler extensions to catch security holes. In IEEE Symposium on Security and Privacy, May 2002.
15. Locking Ruby in the Safe http://www.rubycentral.com/book/taint.html
16. LURHQ. Phatbot Trojan Analysis. http://www.lurhq.com/phatbot.html
17. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-Aware Malware Detection. In IEEE Symposium on Security and Privacy, May 2005.
18. Overton, M.: Bots and Botnets: Risks, Issues, and Prevention. In Virus Bulletin Conference, Dublin, Ireland, October 2005.
19. Ianelli, N., Hackworth, A.: Botnets as a Vehicle for Online Crime. CERT Coordination Center, December 2005.
20. perlsec http://perldoc.perl.org/perlsec.html
21. Forrest, S., Hofmeyr, S., Somayaji, A., Longstaff, T.: A Sense of Self for Unix Processes. In IEEE Symposium on Security and Privacy, May 1996.
22. Kandula, S., Katabi, D., Jacob, M., Berger, A.: Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. In Network and Distributed System Security Symposium, May 2005.
23. Strider GhostBuster Rootkit Detection http://research.microsoft.com/rootkit/
24. Garfinkel, T., Rosenblum, M.: A Virtual Machine Introspection Based Architecture for Intrusion Detection. In Network & Distributed Systems Security, Feb. 2003.
25. Honeynet Project & Research Alliance. Know your Enemy: Tracking Botnets.
26. The majority of bot code was obtained from: http://tinyurl.com/3y4cfd
27. Shankar, U., Talwar, K., Foster, J., Wagner, D.: Detecting format string vulnerabilities with type qualifiers. In Proc. 10th USENIX Security Symp., Aug. 2001.
28. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In Proc. 11th USENIX Security Symposium, August 2002.

29. Parizo, E.:s New bots, worm threaten AIM network. SearchSecurity, Dec. 2005.
30. Naraine, R. Money Bots: Hackers Cash In on Hijacked PCs. eWeek, Sept. 2006.
31. Cui, W., Katz, R., Tan, W.: BINDER: An Extrusion-based Break-in Detector for Personal Computers. In Proc. of the 21st Annual Computer Security Applications Conference, December 2005.
32. Martin, K.: Stop the bots. In The Register, April, 2006.
33. Keizer, G.: Bot Networks Behind Big Boost In Phishing Attacks. TechWeb, Nov. 2004.
34. Christodorescu, M., Jha, S.: Testing Malware Detectors. In Proc. of the International Symposium on Software Testing and Analysis, July 2004.
35. MSDN Library. Using Messages and Message Queues. http://tinyurl.com/27hc37
36. Symantec Internet Security Threat Report, Trends for July 05-December 05. Volume IX, Published March 2006.
37. Sturgeon, W.: Net pioneer predicts overwhelming botnet surge. ZDNet News, January 29, 2007.
38. Symantec Internet Security Threat Report, Trends for January 06-June 06, Volume X. Published September 2006.
39. Barford, P., Yegneswaran, V.: An Inside Look at Botnets. Special Workshop on Malware Detection, Advances in Information Security, Springer Verlag, 2006.
40. Freiling, F., Holz, T., Wicherski, G.: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks. In European Symposium On Research In Computer Security, September 2006.
41. Rajab, M., Zarfoss, J., Monrose, F., Terzis, A.: A Multifaceted Approach to Understanding the Botnet Phenomenon. In Proc. of ACM SIGCOMM/USENIX Internet Measurement Conference, October 2006.
42. Jevans, D.: The Latest Trends in Phishing, Crimeware and Cash-Out Schemes. Private correspondence.
43. Gu, G., Porras, P., Yegneswaran, V., Fong, M., Lee, W.: BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation. Manuscript.
44. Goebel, J., Holz, T.: Rishi: Identify Bot-Contaminated Hosts by IRC Nickname Evaluation. 1st Workshop on Hot Topics in Understanding Botnets, April 2007.
45. Karasaridis, A., Rexroad, B., Hoeflin, D.: Wide-Scale Botnet Detection and Characterization. 1st Workshop on Hot Topics in Understanding Botnets, April 2007.
46. Kristoff, J.: Botnets. NANOG32, October 2004.
47. Ramachandran, A., Feamster, N., Dagon, D.: Revealing botnet membership using DNSBL counter-intelligence. In 2nd Workshop on Steps to Reducing Unwanted Traffic on the Internet, July 2006.
48. Grizzard, J., Sharma, V., Nunnery, C., Kang, B., Dagon, D.: Peer-to-Peer Botnets: Overview and Case Study. 1st Workshop on Hot Topics in Understanding Botnets, April 2007.
49. Wang, Y., Beck, D., Vo, B., Roussev, R., Verbowski, C.: Detecting Stealth Software with Strider GhostBuster. Microsoft Technical Report MSR-TR-2005-25.
50. Lam, V., Antonatos, S., Akritidis, P., Anagnostakis, K.: Puppetnets: Misusing Web Browsers as a Distributed Attack Infrastructure. In the 13th ACM Conference on Computer and Communications Security, October 2006.
51. Schneier, B.: Semantic Attacks: The Third Wave of Network Attacks. In the Cryptogram newsletter, October 15, 2000.
52. Stinson, E., Mitchell, J.: Characterizing the Remote Control Behavior of Bots. Manuscript. http://www.stanford.edu/~stinson/pub/botswat_long.pdf
53. mIRC Help, Viruses, Trojans, and Worms. http://www.mirc.co.uk/help/virus.html