SANS Institute
Information Security Reading Room

Gh0st in the Dshell: Decoding Undocumented Protocols

David Martin

Copyright SANS Institute 2020. Author Retains Full Rights.

This paper is from the SANS Institute Reading Room site. Reposting is not permitted without express written permission.

Gh0st in the Dshell: Decoding Undocumented Protocols

GIAC (GCIA) Gold Certification

Author: David M. Martin, dmartin@mastersprogram.sans.edu
Advisor: Manuel Humberto Santander Peláez
Accepted: 2016-05-08

Abstract

While many types of malware use well-documented protocols, such as HTTP, HTTPS, or IRC for command and control, any network traffic analyst will eventually encounter malware that uses an undocumented, custom protocol. This traffic is sometimes encrypted but often relies on simple obfuscation techniques or security through obscurity to avoid detection. These protocols must be decoded to understand what an attacker is doing on a victim system and develop signatures to detect it. The art of reverse-engineering undocumented network protocols can be a difficult and time-consuming process, but can be greatly simplified by using Dshell, a network traffic analysis framework developed by the US Army Research Lab and recently released open-source to the information security community. Dshell comes with a number of powerful built-in decoders, but also allows analysts to write custom decoder and parser modules for new network protocols. This powerful and extensible framework will prove a valuable tool for decoding many protocols not readable by other tools. This case study will demonstrate the process of reverse engineering a command and control protocol and writing a Dshell decoder for it, using the Gh0st remote access Trojan (RAT)'s proprietary network communication protocol as an example.

© 2016 The SANS Institute Author retains full rights.

1. Introduction

A 2015 study indicated that nearly 70 percent of traffic on the internet was made up of HTTP (57.39%) and HTTPS (9.53%) web traffic. Known and suspected Peer to Peer (P2P) traffic such as BitTorrent made up the next largest portion, at just over 20 percent. Other documented protocols, such as SMTP email, Secure Shell (SSH) and Domain Name Service (DNS) together make up only slightly more than 5 percent. Unidentified traffic that did not conform to any of the protocols documented by the Internet Assigned Numbers Authority (IANA), accounted for almost 6 percent of all internet traffic (Richter, P., Chatzis, N., Smaragdakis, G., Feldmann, A., & Willinger, W., 2015). While traffic composition will vary somewhat from network to network, sooner or later anyone who examines network traffic for a living will encounter suspicious, unknown network traffic. Sometimes there is a reasonable explanation for the traffic; perhaps a proprietary protocol used by an unfamiliar application or a fragment of encrypted data. Other times, the unknown traffic turns out to be the result of network scanning or operating system profiling tools such as NMap. Perhaps most interesting and dangerous of all is the presence of a malware command and control (C2) mechanism that is trying to avoid detection. Once such a protocol has been detected, the analyst must attempt to learn how the protocol works and whether it poses a threat to their network

The first step in decoding an unknown protocol is to determine enough of its general characteristics to be able to identify other occurrences of the protocol in live or captured network traffic. At this point it is often a good idea to write a Snort or similar IDS rule to detect further instances of the novel protocol. This allows the analyst to detect the source and destination of the traffic and obtain further samples to study. If the traffic is being produced by malware, it will also help to locate and isolate the infected system producing the traffic. As will be seen in the Gh0st example, having a copy of the malware to reverse engineer can be invaluable in deciphering a command and control protocol.

Understanding the principles of how network communication protocols operate is crucial to decoding an unknown protocol. All network protocols work on the principle of encapsulation, with the header of the encapsulated protocol at the beginning of the

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

encapsulating protocol. Malware command and control protocols are no exception. Many malware variants encapsulate their traffic in an application layer protocol, frequently HTTP or IRC, while others, like the Gh0st C2 protocol examined in this paper, ride directly over TCP or UDP. Custom protocols will often have a protocol header containing readable information, even when the message is obfuscated or encrypted. In some cases, the header fields even contain the keys to decrypt the rest of the message. Certain header components such as length fields, sequence numbers or checksums are present in most protocol headers and can help to identify them. Many kinds of malware C2 traffic share unique characteristics that can aid in their identification. Malware will often send repeated beacons that contain information about the victim's system to a command and control server and will frequently employ obfuscation techniques, such as XOR or RC4 encoding (Grosfelt, 2014). An example of both these behaviors is the PlugX malware, which sends encoded beacons to the attacker's C2 server at a regular interval. These beacons contain, among other information, the infected computer's IP address, hostname and the username under which the malware is running (Stewart, 2014).

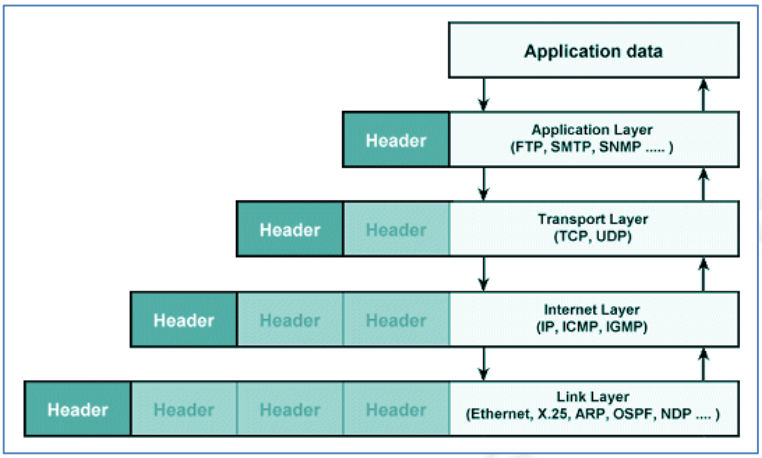


Figure 1: Encapsulation of data in the TCP/IP protocol stack (Wells)

Another suspicious behavior is a reverse protocol in which the where the presentation layer data flows in the opposite direction of the transport layer connection. An example of this situation is a reverse shell, in which malware on the victim system initiates an outbound TCP connection to the hacker's computer. The hacker then initiates a shell connection to send commands back to the victim over the inbound TCP connection. This differs from an ordinary shell connection like Telnet or SSH, the user initiates both the TCP and shell connections. Reverse shells are often used to allow an attacker to connect to a system that is not directly accessible from the Internet, because it

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

is behind NAT or a corporate firewall (Hammer, 2006). This type of traffic can also be detected by finding flows in which the client sends more data than the server.

Once a command and control protocol has been identified and the decoding process begins, the next logical step is to write a script to decode future instances of the same traffic. A tool known as Dshell can greatly simplify this process, allowing the analyst to write a decoder module that will identify and parse the newly characterized traffic. Because Dshell is written in Python, new decoders can be developed by anyone with a reasonable amount of scripting skill. The decoder can be iteratively improved as the analyst's understanding of the protocol improves and each improvement in the decoder can improve the understanding of the protocol. Analyzing a sample of the malware generating the command and control traffic can also provide valuable information about how the malware communicates.

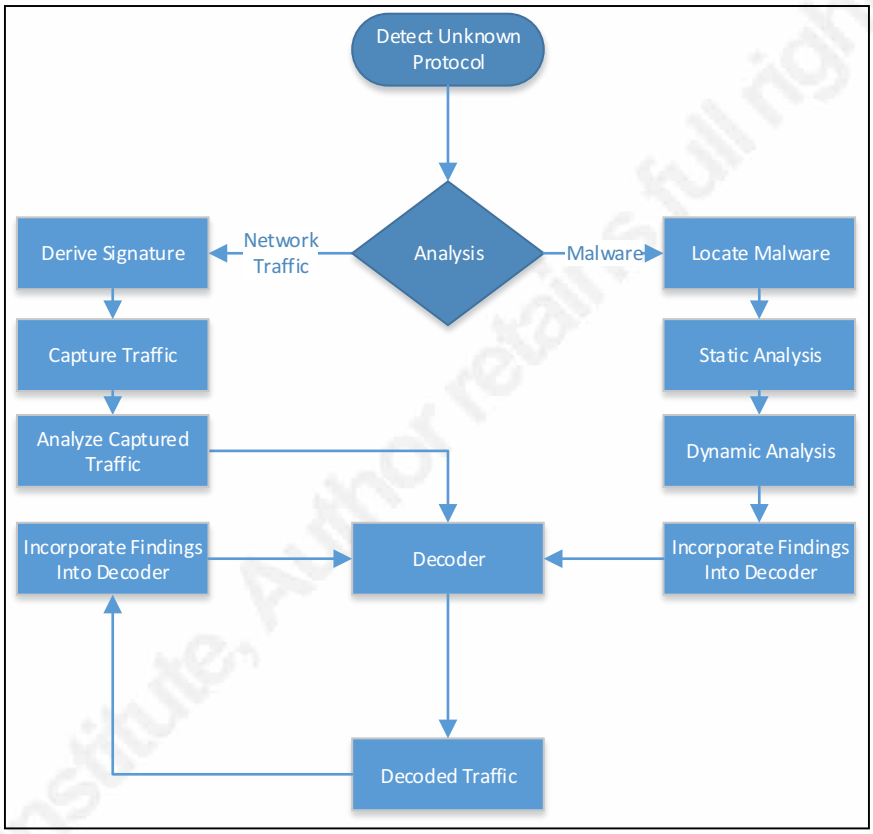


Figure 2. Decoder Development Process

To demonstrate the process of decoding an unknown protocol and writing a Dshell decoder for it, I have chosen the Gh0st RAT command and control protocol as an example. Gh0st, which is discussed in greater detail later in this paper, is a well-known Remote Access Trojan (RAT) that has been used by several different hacker groups and is easily available for download today. It uses a proprietary communication protocol that

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

has been well documented by security researchers. It does not employ encryption, so decoding is possible without the need to obtain keys from either side of the intrusion.

2. Dshell

In December 2014, the United States Army Research Labs (ARL) released the Dshell project on their GitHub page. This powerful network traffic analysis framework had been in development and use by ARL and other law enforcement and intelligence agencies for several years. With its open source release, Dshell became available for use and improvement by security professionals and researchers throughout the community. It is a Linux command line utility that has been described as "Metasploit for network traffic" or "TCPDump on steroids." Dshell is written in Python and depends on several Python libraries, including *pypcap*, *dpkt*, *PyCrypto* and *IPy*. Dshell was developed on Ubuntu, but is compatible with most Linux distributions and Macintosh OS X. While it is theoretically possible to compile Dshell for Windows, doing so is currently non-trivial, as Dshell's build process involves setting a number of Linux-specific environment variables and relies on the *make* utility that is not present in Windows. With the forthcoming introduction of an Ubuntu-based Linux subsystem and Bash shell in Windows 10, it will soon be possible to run Dshell in Windows as easily as Linux.

2.1. Capabilities

The power of Dshell is its modularity and extensibility through custom decoder modules that can be written to extract information from network traffic. This capability is particularly useful when analyzing new or unknown protocols. Some decoders can be chained together, feeding the output of one decoder into another. It is able to decode live network traffic or read from pcap or pcap-ng files. It includes a number of output options, including *stdout*, CSV, JSON, XML, and HTML. It can also write its results directly to one of several databases, such as SQLite, MySQL, and Elasticsearch. Another key benefit of the Dshell framework is that it provides implementations of many utility functions such as parsing command line parameters, reading network traffic and formatting output. This can save analysts a great deal of time and effort, allowing them to focus on network traffic analysis rather than coding.

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

2.2. Usage

Dshell installation is relatively straightforward. After installing the dependencies in the included README file, the user can build the application by issuing the **make** command from the dshell root directory. Once **make** finishes, Dshell can be launched by running the **dshell** command from the Dshell root directory. The 'Dshell>' prompt indicates the user is in the Dshell environment, which is actually a Bash instance with custom environment variables and a custom path. The command **decode** will then launch the actual decoding framework. If no option is specified, Dshell will print a simple usage

```
dave@dave-VirtualBox ~/Dshell ./dshell
dave@dave-VirtualBox:~/Dshell Dshell> decode
Usage: decode [options] [decoder options] file1 file2 ... filen [-- [decoder ar

Options:
  --version             show program's version number and exit
  -h, -?, --help        Print common command-line flags and exit

  Input options:
    -i INTERFACE, --interface=INTERFACE
                        listen live on INTERFACE
    -c COUNT, --count=COUNT
                        number of packets to process
    -f BPF, --bpf=BPF   replace default decoder filter (use carefully)
    --nofilterfn        Set filterfn to pass-thru
    -F FILEFILTER       Use filefilter as input for the filter expression.  An
                        additional expression given on the command line is
                        ignored.
    --ebpf=EBPF         BPF filter to exclude traffic, extends other filters
    --no-vlan           do not examine traffic which has VLAN headers present
    --layer2=LAYER2     select the layer-2 protocol module
    --strip=STRIPLAYERS
                        extra data-link layers to strip

  Logging options:
    -L LOGFILE, --logfile=LOGFILE
                        log to file
    --debug             debug logging (debug may also affect decoding
                        behavior)
    -v, --verbose       verbose logging
    -q, --quiet         practically zero logging
```

Figure 3 Running Dshell

screen and exit. The basic format for the command is: **decode** *-d [decoder name]+[chained decoder name] [options] file1 file2 … filen [decoder args].*

2.2.1. General Options

In order to actually decode network traffic, the user must specify a decoder to process the traffic using the *-d* or *–decoder* option, followed by the name of the decoder to run. The *-l* option will provide a list all the installed decoders and some information about what they do, which is useful for choosing an appropriate decoder. A good starting

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

point can be the *netflow* decoder that will display basic netflow information (Timestamp, Source IP, Destination IP, Source Port, Destination Port, Protocol, Packets Sent, Packets Received, Bytes Sent, Bytes Received and Duration). The *followstream* decoder is another good all-purpose decoder for displaying the content of network traffic, though it can produce an overwhelming amount of output if not carefully filtered. Some decoders are designed to decode a particular type of network traffic, like web, FTP or DNS. Often, there are several options to decode the same kind of traffic. For example, the *web, httpdump* and *rip-http* decoders all extract different information from HTTP web traffic. While the *web* decoder displays http headers and content, *httpdump* parses useful information about HTTP sessions such as user-agent strings, cookies and URL parameters from HTTP sessions and *rip-http* rips files from HTTP traffic. Other decoders like *grep* and *snort* are designed to search for traffic that matches a given search expression, or snort rule. Decoders whose names are followed by the plus sign are "chainable," meaning that their output can be passed directly to other decoders. For example, the command *'***decode** *-d grep+followstream'* would search for the supplied expression and pass any matching traffic to the *followstream* decoder for viewing.

Users can view the a more detailed list of options with the *-h* flag. A configuration file specifying a set of options can be supplied with the *-C* or *–config* option. There is no documentation regarding the format of the configuration file, but an examination of the source code reveals that it uses the Python *ConfigParser* module for parsing the standard config.ini format. Command line options can be specified in a config file using *option = value* pairs. The user can also choose to run multiple decoders in parallel with the *-p* switch or process multiple files in parallel with the *-t* switch. In either case, the user can limit the number of concurrent processes with the *-n* flag. The *–usag'* and *–status* flags, while listed in the usage instructions, do not appear to be implemented yet.

2.2.2. Input Options

Dshell can listen on a network interface specified with the *-i [interface]* flag, though it must be run as *root* to do so. If this flag is not present, it reads from one or more pcap files provided on the command line. Other input options include handling VLAN headers with the *–vlan* flag, handling layer-2 protocols other than the default Ethernet

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

with the *–layer2=[protocol]* flag and stripping extra data-link layers with the *--strip* flag. By far, the most frequent input options are the *–bpf* and *–ebpf* flags. Both use the standard Berkley Packet Filter (BPF) language to filter the input provided to the decoder, which can be valuable in keeping Dshell from outputting an overwhelming amount of data. All decoders have a standard BPF filter defined in the module that can be added to with the *–ebpf* option or overridden with the *–bpf* option. For example, if the default filter for a decoder were *tcp and port 80,* applying the option *–ebpf='src port 1234'* would result in a filter of *tcp and port 80 and src port 1234*, where *—bpf='src port 1234'* would completely replace the default filter. This option should be used with case, as it can potentially result in a much larger amount of traffic being processed. More complicated filters can be included from a file, using the *-F* option and providing the filename.

2.2.3. Output Options

Dshell provides a very flexible set of output options that can be selected on the command line. To understand the output options, it is necessary to first understand the two types of output generated by Dshell decoders. Alerts are somewhat analogous to snort alerts, as they typically contain a message that is displayed on the screen indicating why the alert was triggered, along with metadata about the traffic that generated the alert, organized into named fields. These alerts and the information contained within are the building blocks of all structured output formats, such as CSV, JSON, and XML. The second kind of output is produced by write statements, which are typically used to format content for human consumption and are suited for protocols requiring interpretation or more verbose output. While most decoders will employ a combination of alerts and writes, others like the followstream decoder only use writes.

The default output mode displays both alerts and writes in plain text to the console or to a text file specified with the *-o* option. Different output modes can be specified with the *-O* or *–output* switch. Many decoders can take advantage of Dshell's *colorout* option to color client-server traffic red and server-client traffic green, much like Wireshark's *followstream* format. When the *colorout* option is selected and output is written to a file, Dshell will generate an HTML file that preserves the color coding of the output. The *xmlout* and *jsonout* formats are well suited for displaying structured

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Default

```
Dshell> decode -d grep --grep_expression=Gh0st gh0st_beacon.pcapng
grep 2016-03-08 19:37:39   192.168.61.129:1043  <-   192.168.61.128:80    ** Gh0stx[illegible]c **
```

CSVOut

```
Dshell> decode -d grep --grep_expression=Gh0st gh0st_beacon.pcapng -O csvout
#decoder,datetime,sip,sport,dip,dport
grep,2016-03-08 19:37:39,192.168.61.129,1043,192.168.61.128,80
```

JSONOut

```
Dshell> decode -d grep --grep_expression=Gh0st gh0st_beacon.pcapng -O jsonout
{"dmac": "00:0c:29:78:4c:38", "sip": "192.168.61.129", "clientbytes": 148, "endtime": "2016-
03-08 19:37:39", "proto": "TCP", "serverpackets": 1, "smac": "00:0c:29:12:b2:67",
"serverbytes": 22, "state": "closed", "starttime": "2016-03-08 19:37:39", "decoder": "grep",
"dport": 80, "pkttype": 2048, "sport": 1043, "dip": "192.168.61.128", "match": "Gh0st",
"clientpackets": 1}
```

XMLOut

```
Dshell> decode -d grep --grep_expression=Gh0st gh0st_beacon.pcapng -O xmlout
<dshell><alert addr="(('192.168.61.129', 1043), ('192.168.61.128', 80))" clientasn="None"
clientbytes="148" clientcountrycode="None" clientip="192.168.61.129" clientpackets="1"
clientport="1043" decoder="grep" dip="192.168.61.128" dipint="3232251264" direction="sc"
dmac="00:0c:29:78:4c:38" dport="80" endtime="1457483859.85" match="Gh0st" pkttype="2048"
proto="TCP" serverasn="None" serverbytes="22" servercountrycode="None"
serverip="192.168.61.128" serverpackets="1" serverport="80" sip="192.168.61.129"
sipint="3232251265" smac="00:0c:29:12:b2:67" sport="1043" starttime="1457483859.74"
state="closed" ts="1457483859.74">Gh0stxKc``</alert></dshell>
```

Figure 4 Comparison of Output Options

connection data all will output all metadata fields encoded into alerts to the selected format. These formats are ideal for importing into other tools for further sorting, correlation, and analysis. It should, however, be noted that the *csvout* option will only display the decoder name and 5-tuple about the connection. Dshell can also output directly to a database, and supports SQLite, MySQL and ElasticSearch by default. In order to work with other database systems, like Oracle or PostgreSQL, some degree of addition or modification to the Dshell source code is required. If the *pcap* output option is specified, any traffic generating an alert can also be written to another *pcap* file. This option is generally used in filter-type decoders, like the *asn-filter* decoder that will output the traffic from a specified Autonomous System Number (ASN).

Each decoder can specify additional options specific to that decoder. For example, the followstream decoder has a '*–followstream_hex'* option to display the output in two-column hexdump format. All decoder specific options begin with the name of the decoder followed by an underscore and the specific option name. Not all decoders have specific options, but others have several. Decoder-specific options and extended information

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

about a decoder can be displayed by calling *"decode -d decoder_name -?"*. All TCP based decoders offer a default option of *'–(decoder_name)_ignore_handshake'* to accommodate protocols that abuse the normal behavior of the TCP protocol. This option can be especially useful for decoding reverse protocols. A complete list of decoders and their decoder-specific options is contained in Appendix B.

2.3. Decoders

Dshell decoders are stored in a series of subdirectories, grouped by type, under the *dshell/decoders/* directory. Any user with an understanding of Python and the Dshell framework can write a decoder and place it in one of these directories, after which it can be run with the *'-d'* option. While there is no formal documentation about the procedure of writing a decoder, the Dshell source code is well commented and provides most of the information necessary to write a decoder. In particular, the scripts in the /lib/ and /lib/output/ folders contain many valuable comments that explain the workings of the Dshell framework. Perhaps the best way to begin writing a Dshell decoder is by examining the source code of the various decoders included with Dshell, then modifying and experimenting with them.

A unique term used in the Dshell code is the "blob", which is best defined as all traffic in a TCP stream sent in a single direction at one time. A blob may consist of one or more TCP segments, but is delimited by any traffic sent the opposite direction. As an example, an HTTP GET request from a client would be a blob. The corresponding HTTP response from the server would be another blob. Any subsequent server responses in the same stream would be part of the same blob, until further traffic in the same stream is received from the client or the stream ends.

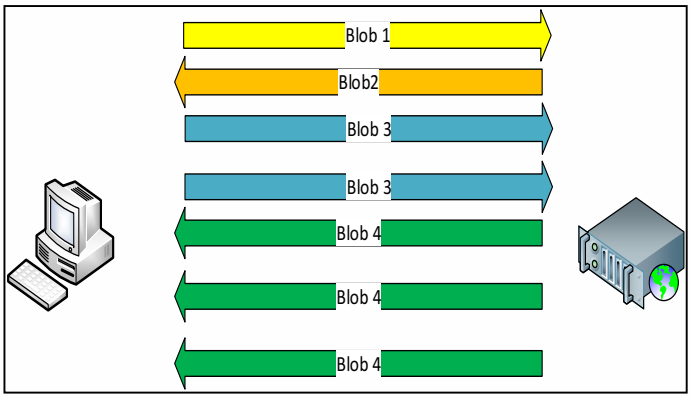


Figure 5 TCP Segments and Blobs

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Dshell supports two basic types of decoders: packet and session, and includes templates for writing both (see Appendix C). Packet decoders have a very simple structure, as they are designed to process each packet individually. It includes, by default, an *__init__* function and a *packetHandler* function that is called for each packet processed by the decoder.

Session decoders are significantly more complex, as they must keep track of the session state. The session decoder template includes an *__init__* function and five default functions for handling session data in different ways. The *packetHandler* function can still be used in a session decoder, generally for UDP data. Otherwise, UDP data will be converted into pseudo-sessions. The *connectionInitHandler* is called as soon as a session is established and before any data is transmitted. It can be useful for setting up tracking variables that will persist throughout a session. The *sessionHandler* waits until it has reassembled all the data from a session. It generally serves as the workhorse function for decoders in which the same processing logic can be applied to the entire stream. The *blobHandler* handles blobs one at a time, as soon as they can be reassembled. It is particularly useful in command and control protocols where the format of the command and response can vary, as will be seen in the Gh0st RAT decoder example. The *connectionCloseHandler* is called after all data has been handled and is often used to clean up data structures and summarize connection information. Decoder authors may use some or all of these functions in addition to any other custom functions they may wish to define.

3. Gh0st RAT

Gh0st RAT is one of several commoditized Remote Access Trojans (RAT's) that have been widely used among the hacker community. Created by a Chinese hacking team calling itself the C. Rufus Security Team, Gh0st has been used by many different attackers, from unsophisticated "script kiddies" to complex cyber espionage operators. Gh0st was employed in high profile intrusion like the 2008-2009 GhostNet intrusions against Tibetan interests throughout the world and Gh0st variants are still in use today (SecDev, 2009). It is relatively easy to locate a copy with nothing more than a Google

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

search and a willingness to download software from one of several suspicious websites. Gh0st was designed to run in Microsoft Windows XP, which was the most common version of Windows in use when it first appeared in 2008. It will run on Windows 7 and later versions, but will trigger the user account control warning as it must run as an administrator. It consists of controller or "client" that is installed on the hacker's computer and allows it to control the victim and an implant or "server" that is installed on the victim computer. At the time of writing, the most widely available version appeared to be Beta 3.6, which I located on what appeared to be a Chinese hacker website, www.15897.com, for use in my testing.

In 2012, McAfee Foundstone security researcher Michael Spohn published a comprehensive analysis of the Gh0st malware (Spohn, 2012). In his research, Spohn compiled Gh0st Beta 3.6 from source, patched a number of errors and added readable English labels before naming the new version the "Foundstone Friendly Gh0st 3.7". That version was visually identical to the test sample used for my testing and was used for reference for deciphering the default Gh0st menu options. Spohn's examination of the Gh0st source code revealed the internal data structures the malware used and how those data structures were encoded into its network command and control protocol. Spohn's research focused heavily on the structure of the Gh0st malware, but also revealed critical information about its communication protocol, including the structure of the header and the fact that the payload was Zlib compressed. This research provided a valuable starting point for the creation of a Gh0st decoder.

3.1. Capabilities

Gh0st is a full-featured RAT with an easy to use GUI. Its features include file upload/download, keylogging, remote terminal, remote process management, a remote desktop client and even audio and video recording of the remote system. The value of these capabilities for a computer intruder is evident. If a hacker succeeds in installing the malware implant on a victim system, they would have complete control and be able to perform nearly any function they wished on the victim system. The Gh0st implant is generated by the controller, making it configurable by the end user. When generating an implant, the user can configure the command and control IP/domain, port and key without having to edit the source code. The user can also configure the implant to reach

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

back to a URL for a configuration file containing the C2 domain and port. This allows an attacker the flexibility to modify their infrastructure without losing contact with their implants as they could if the C2 domain or IP were statically encoded into the implant when it was generated.

3.2. Usage

Gh0st can be obtained in one of several forms, the simplest of which is a single-file Windows executable. The full C++ source code is also available on GitHub for those who wish to tweak or modify the code or simply compile it for themselves. The single executable was used for testing. It launched once Antivirus protection was disabled, showing a windows with three tabs and a grid. The interface appeared to have been developed on a computer using a Chinese character set, as most of the labels and menu items either used Chinese characters or did not render properly. This is consistent with reports that Chinese hackers developed Gh0st.

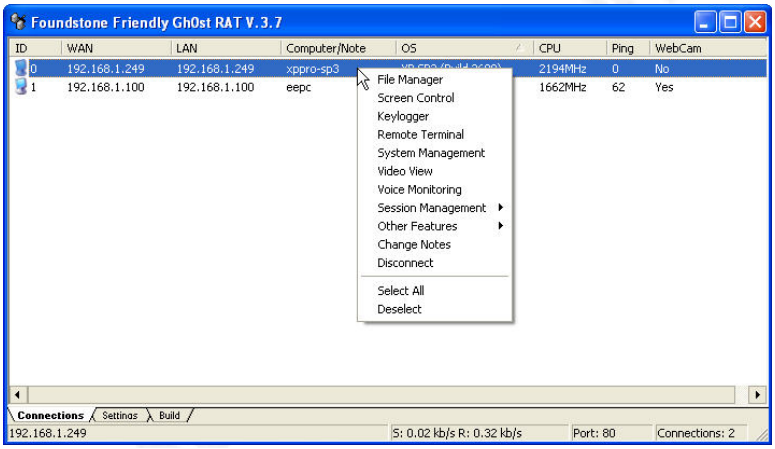


Figure 6. Foundstone Friendly Gh0st Main Grid (Spohn, 2012)

The process of generating a Gh0st implant from the controller is quite simple. In order to generate an implant with a hardcoded command and control address, simply copy the key string from the "Settings" tab into the proper field in the "Build" tab. Any changes to the controller configuration will result in changes to the key, so implants configured with the old key will no longer be able to connect. The user can also configure the service name the implant will disguise its self as. Once the key is entered, the user clicks on the

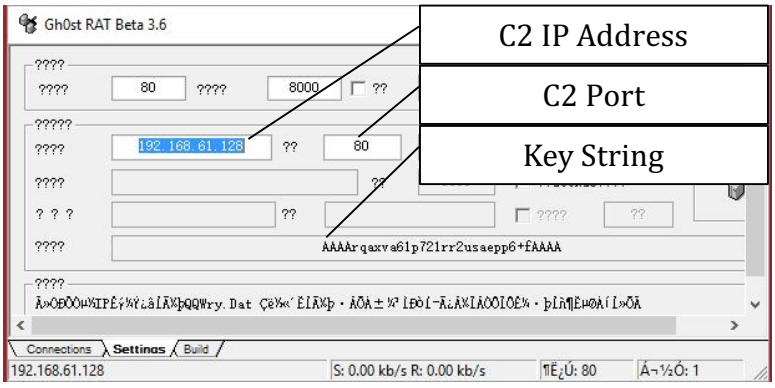


Figure 7. Gh0st Controller Settings Tab

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

"Generate" button and choses the location and filename where they wish to save the implant, which they can deliver to their victim through the method of their choice.

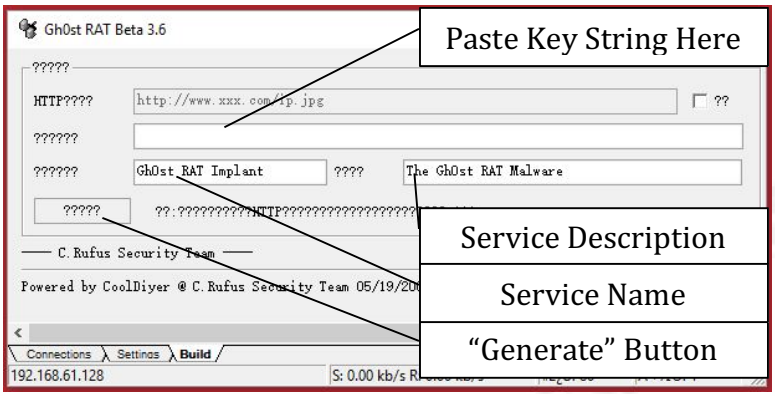
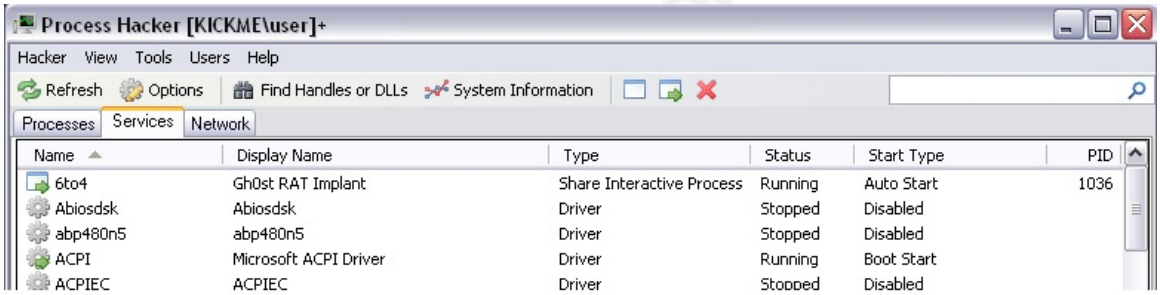


Figure 8. Gh0st Controller Implant Generation Tab

When executed on the victim system, Gh0st writes 6to4ex.dll to %SystemRoot%\system32\ and installs it as a service set to start automatically. The service appears under the name "6to4" and lists the description entered by the attacker when generating the implant. Once a victim has been implanted with the Gh0st "server",



it will begin emitting beacons containing system information to the controller via the configured IP and port. When the Gh0st controller receives a beacon, the victim's information will appear as a row in the table in the Gh0st controller's main window. This will list the victim's LAN and WAN IP address, hostname, operating system, CPU speed, ping and whether it has a webcam connected. Right-clicking on a victim will open a menu listing actions that can be performed on the victim system.

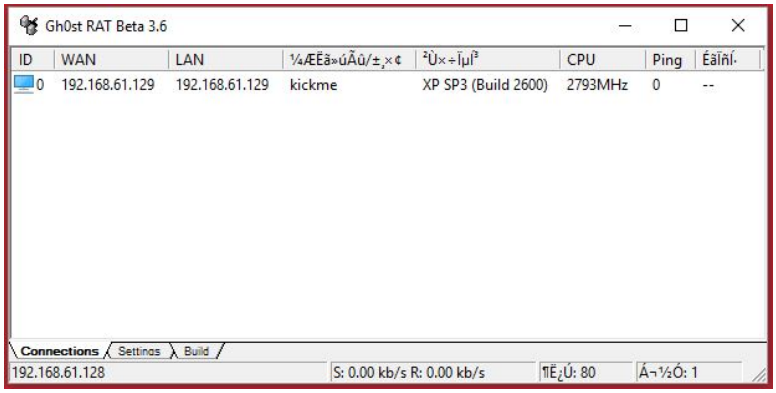


Figure 10. Gh0st Controller Main Grid

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Gh0st provides all the features an enterprising intrusion actor would need: a file manager, remote desktop, remote shell, a key logger, audio and video recording and a system management tool. When selected, each tool opens a new window. The file manager looks quite similar to the native Windows XP file manager and provides a drag and drop interface to upload and download files. The screen control feature operates similarly to remote desktop, though with added malicious features like the ability to block other users' input. Unlike the Windows Remote Desktop, a user who is logged into the victim system will be able to see the attacker's activity on their screen. The remote terminal opens a fully functional interactive Windows command shell that appears nearly indistinguishable from a default Windows command prompt. The video view and voice monitoring options allow the hacker to activate and monitor the victim's webcam and microphone. The keylogger opens a window that displays text and user interactions in real time. The system management tool opens a task manager-like window and allows the user to terminate or restart processes or windows.

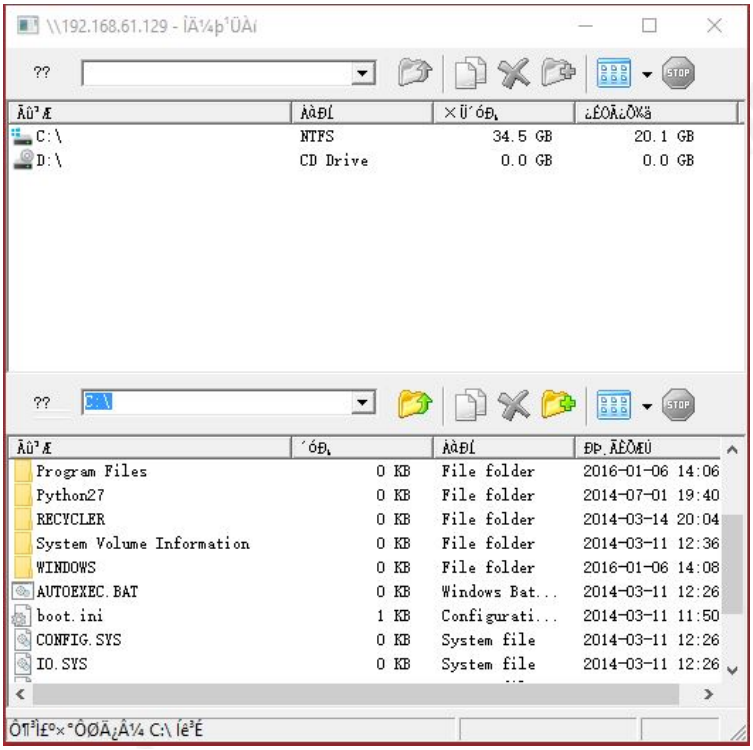


Figure 11. Gh0st File Manager

Figure 12. Gh0st Remote Terminal

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

There are also two sub-menus containing further options for interacting with the implant: session management and other features. The session management menu allows the actor to restart the implant, change the controller location or uninstall the implant. The actor can also restart or shut down the system remotely. The attacker also has the option to change a client's display name for easier management of numerous compromised hosts.

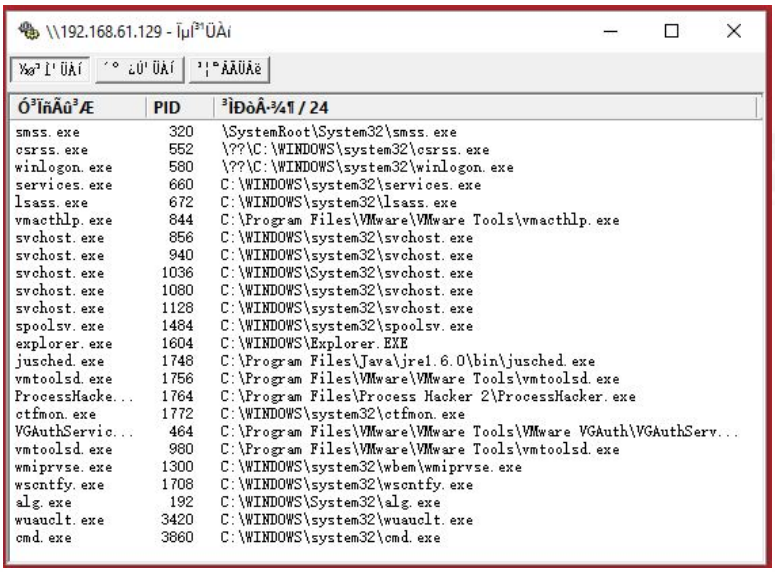


Figure 13. Gh0st System Management Window

3.3. Communication Protocol

While Spohn went into a great deal of detail about the structure and functionality of the Gh0st malware, my analysis focused on its network communication protocol. Gh0st uses a fairly simple, but proprietary protocol for communication between the implants and their controller. Network communication is conducted over port 80 by default, but can be configured to use any port the user desires. A Gh0st packet consists of three header fields and a variable-length payload. The first 5 bytes contain the initialization string "Gh0st". While this is the default string, numerous other initialization strings have been reported, so this string should not be relied upon for identification (Spohn, 2012). The next 4 bytes contain a Little-Endian integer representing the packet length. The 4 bytes after that contain a Little-Endian integer representing the packet length once the payload is uncompressed. This header is followed by a variable length payload, compressed with the Zlib compression algorithm (Spohn, 2012).

The first byte of the payload is always an "OpCode" that contains one of three types of data: Command Codes that tell the implant what to do, Token Codes that identify the type of payload returned by the implant or Mode Codes that are used by both the

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

implant and the controller (Spohn, 2012). By examining the Gh0st source code, Spohn identified the functions that generated the opcodes and their descriptions. He compiled a full list of OpCodes, which I have included in Appendix D.

Command Code	Hex
COMMAND_LIST_DRIVE	0x01
COMMAND_LIST_FILES	0x02
COMMAND_DOWN_FILES	0x03
COMMAND_UPLOAD_FILE	0x04
COMMAND_FILE_DATA	0x05
COMMAND_SYSTEM	0x35
COMMAND_PSLIST	0x36
COMMAND_KILLPROCESS	0x39
COMMAND_SHELL	0x40
IMPLANT_LOGIN	0x66
IMPLANT_DRIVE_LIST	0x67
IMPLANT_FILE_LIST	0x68
IMPLANT_FILE_DATA	0x6A
IMPLANT_SHELL_START	0x80

Figure 14. Significant OpCodes

The remainder of the packet contains the information being passed between the controller and implant. The majority of command codes correspond with menu items, though multiple commands are associated with most menu items. For example, COMMAND_LIST_DRIVES, COMMAND_LIST_FILES, and COMMAND_DOWN_FILES are all associated with the file manager tool.

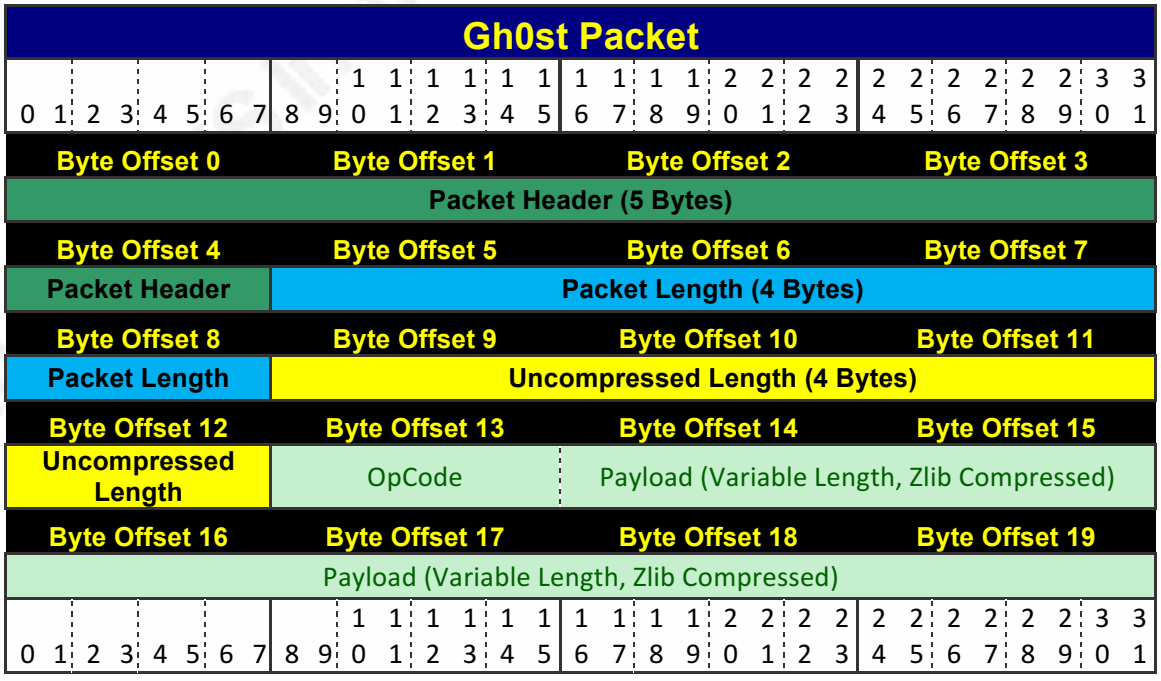


Figure 15. Gh0st Packet Structure

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

As will be seen in section 4.2, Each OpCode is followed by a unique series of data structures containing parameters, such as the name of a file to transfer or the names of the files in the directory the actor is browsing. These appear to be consistent for each command or response and can often be identified by plain-text artifacts that are obvious once the payload is decompressed. In this case, I had the advantage of being able to examine the source code for both the implant and controller, observe how messages were displayed by the controller and consult Spohn's previous analysis of the malware. This allowed me to quickly and accurately identify fields and their encoding by being able to check my answers against outside sources.

4. The Gh0st Decoder

My goal was to create a decoder that would convert the encoded command and control traffic into an easily readable transcript of the actions the hacker was performing on the implanted system. I installed the Gh0st controller in one virtual machine with IP 192.168.61.128 and generated an implant set to beacon to that IP address. I transferred the implant to a Windows XP malware VM (192.168.61.129), which was running on the same host-only network, and detonated it. As expected, less than two minutes later, an entry for the victim VM appeared in the Gh0st controller window, as can be seen in Figure 8. I worked through the menu options in order, generating traffic representative of the most common intruder activities on a victim system. First I opened a command prompt and ran several commands typical of network reconnaissance, including *dir*, *netstat* and *net use*. I

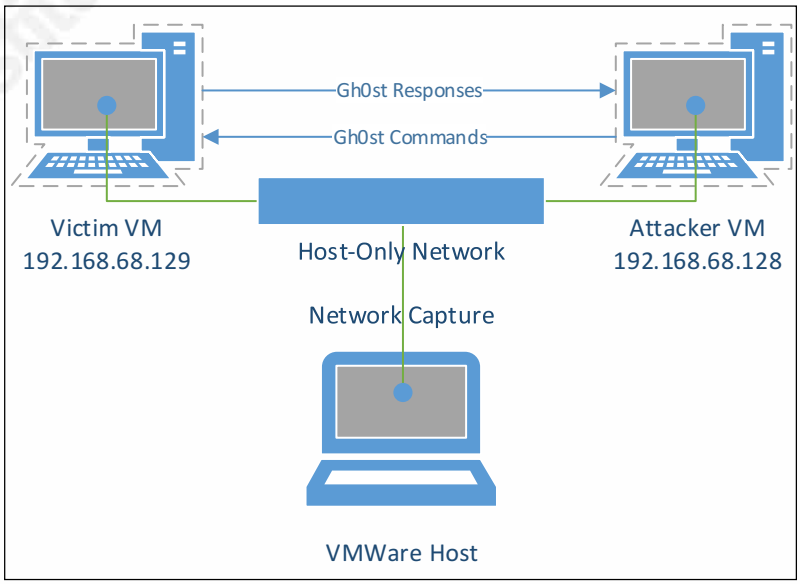


Figure 16. Test VM Network Setup

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

then used Gh0st's keylogger function to capture a username and password from the victim's web browser. I used the file manager to browse the user's documents, downloaded one and uploaded the implant again. I concluded by using the process manager to view and terminate processes. I used Wireshark on my OS X host to capture the traffic from the host-only network I would use to develop and test the decoder.

4.1. Parsing the Protocol Header and Decompressing

As Gh0st uses a connection-oriented TCP protocol for communications, I chose the session decoder template as the starting point for the Gh0st decoder due to its ability to track the connection state. Due to the fact that Spohn's research indicated that the Gh0st payload was Zlib compressed, the first step in creating a decoder was to identify the Gh0st protocol header and decompress the payload. I first created a simple proto-decoder that parsed out the init string and the compressed and uncompressed message sizes from the header, then decompressed the message body for further inspection using the Python *zlib.decompress()* function. Even this minimal decoder produced a large amount of somewhat human-readable output that allowed for further reverse-engineering and the eventual creation of the full-featured decoder. Since the Gh0st traffic contained distinct messages and responses within each TCP stream, I chose the *blobHandler* for the majority of the decoding. Inside this function, I iterated through the TCP segments in the blob, checked for the init string, and decompressed the remainder of the message. I set the output of the decoder to display in two-column hexdump-style hex/ASCII output, so that I could easily examine

```
gh0st connection from 192.168.61.129

Start: 2016-03-09 00:37:39.739821 UTC   End: 2016-03-09 00:37:39.849615 UTC
192.168.61.129:1043 -> 192.168.61.128:80 (148 bytes)
192.168.61.128:80 -> 192.168.61.129:1043 (22 bytes)
00000000   47 68 30 73 74 94 00 00 00 E0 00 00 00 00 00 00     Gh0st..........
00000010   9C 00 00 00 05 00 00 00 01 00 00 00 28 0A 00 00     ............(...
00000020   02 00 00 00 53 65 72 76 69 63 65 20 50 61 63 6B     ....Service Pack
00000030   20 33 00 00 01 00 00 00 E9 03 00 00 66 00 02 00      3.........f...
00000040   1C 00 38 00 02 00 00 00 04 04 00 00 40 02 00 00     ..8.........@...
00000050   DD C7 00 00 00 00 00 00 00 00 00 00 01 00 01 00     ................
00000060   00 00 00 00 00 00 00 00 88 1C 00 00 04 04 00 00     ................
00000070   14 08 00 00 02 00 04 13 00 00 00 00 00 00 00 00     ................
00000080   60 5E 06 02 02 00 00 50 78 DC 37 00 58 E8 D2 05     `^.....Px.7.X...
00000090   F8 6F 15 00 DC 10 00 00 00 00 00 00 A3 5E 06 02     .o...........^..
000000A0   00 20 00 00 03 00 00 00 00 01 01 00 E9 0A 00 00     . ..............
000000B0   C0 A8 3D 81 6B 69 63 6B 6D 65 00 00 00 00 00 00     ..=.kickme......
000000C0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000000D0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00     ................
000000E0   00 00 00 00 00 00 00 00 00 00 00 00                 ............
00000000   47 68 30 73 74 16 00 00 00 01 00 00 00             Gh0st........
```

Figure 17. Output of Gh0st Proto-Decoder

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

both the raw hex for numerical fields and ASCII representation for text fields. The code for the proto-decoder is included in Appendix E for reference.

With each new field I located, I built additional functionality into the Gh0st decoder to parse the message structure for the various Gh0st functions until I was able to create a full protocol decoder. By default, the 5-byte init string, contained in bytes 0-4, is "Gh0st". The sample I examined used the default init string, but for the sake of reusability, I included the complete list of known Gh0st init strings as a dictionary data structure that the decoder attempts to match against the first 5 bytes (Meta, 2015). If a sample using an unrecognized init string needs to be decoded, the init string can be added to the decoder's dictionary by editing the decoder source code. Alternately, the *'–gh0st_ignoreinit'* option will cause the decoder to attempt to decode all traffic matching its filters as gh0st, regardless of their init string.

The compressed message size is stored in bytes 5-8 and the uncompressed message size in bytes 9-12. I observed that all numerical fields in the Gh0st protocol used little-endian encoding, and in almost all cases used 4-byte fields. This was extremely useful in identifying the locations of fields containing information like sizes and offsets. Decoding these values required me to import the Python *struct* library for its unpack function that would convert them to a string containing a decimal number that could easily be cast to an integer as needed. These particular values only had to be printed, so were left as strings.

After parsing the header, I used the Python *zlib.decompress* function to decompress the remainder of the message, which began at byte offset 13. After the message payload is decompressed, the first byte is an OpCode, identifying either a command or response, as detailed in the Foundstone report. I programmed all of these OpCodes into a dictionary in the decoder that was used to look up descriptions of the codes. I mostly preserved Spohn's naming conventions, which were, themselves derived from function names in the Gh0st RAT source code, but renamed the "TOKEN" prefix to "IMPLANT" to clarify the fact that these OpCodes represented responses from the implant.

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

4.2. Parsing the Payload

The remainder of the payload, following the OpCode varied somewhat depending on the command or response and necessitated numerous special handling cases for parsing them in the *blobHandler* and formatting their output in *connectionCloseHandler*. I did not write handlers for all Gh0st RAT functions. In particular, I did not parse the screen capture, audio capture or webcam recording functions because I was unable to determine how they were encoded or extract any useful information from them. By examining the output of my proto-decoder, I was able to determine the arrangement of data structures for most of the Gh0st OpCodes. In order to keep the segments in a connection in sequential order, loaded them into the extremely useful *OrderedDict* data structure provided by the Python *collections* library.

When a Gh0st connection is initiated, it attempts to connect to its command and control address on port 80 at approximately 2-minute intervals until it establishes a connection. This initial login (OpCode 0x66), contains the Windows version information obtained by querying the *OsVersionInfoEx* Win32 API function (Spohn, p26). The major version is stored between offset 04-07, the minor version between offset 08-0A and the build number in a 5-byte field from 0C-10. When converted to decimal, this example decodes to version 5.1, build 2600, which corresponds to the Windows XP Service Pack 3 on the infected VM. It also contains the processor speed in MHz between 9F-A0, the

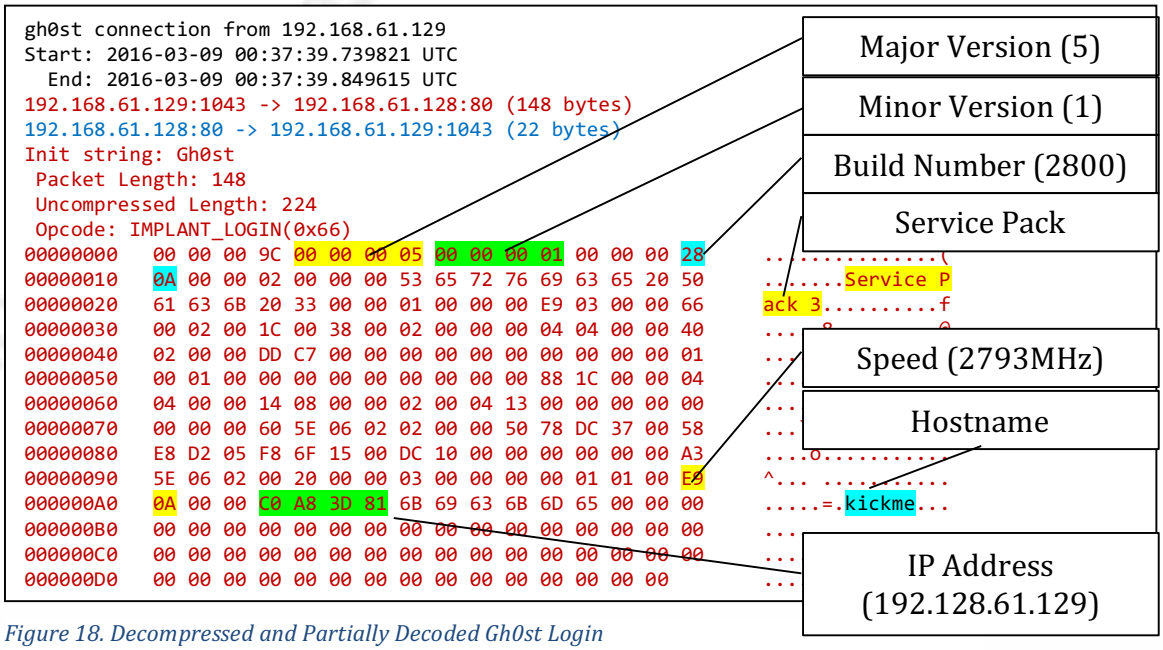


Figure 18. Decompressed and Partially Decoded Gh0st Login

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

address between offset A4-A6 (which requires the Python socket library to convert to dotted decimal notation), and the hostname between A7-D8. As can be seen below, there is more data available in the login message than I was able to identify. This TCP stream remains alive for the duration of the command and control session, while other functions launch within different TCP streams. This appears to correspond with windows in the Gh0st controller, each of which seem to create their own TCP stream.

After determining the structure of the login message, I created a handler case that looked for the IMPLANT_LOGIN OpCode (0x66) and parsed the data from the correct offsets and added to the *msg* dictionary that contains the message. Using a similar process, I created decoding routines for the file manager, the command prompt and the keylogger functions of the implant. Each of these cases presented unique challenges in parsing and formatting their data. I also discovered that there were some edge cases where Zlib decompression was failing. As it turned out, these consisted of cases where a message spanned multiple segments. In these instances, the subsequent segments would simply be continuations of the previous message without their own '*gh0st'* header. I provided for message reassembly by concatenating the payload portion that was failing to decompress with as many subsequent segments as necessary for the message to decompress properly.

```
gh0st connection from 192.168.61.129
Start: 2016-03-09 00:37:39.739821 UTC
  End: 2016-03-09 00:37:39.849615 UTC
192.168.61.129:1043 -> 192.168.61.128:80 (148 bytes)
192.168.61.128:80 -> 192.168.61.129:1043 (22 bytes)

Init string: Gh0st
 Packet Length: 148
 Uncompressed Length: 224
 Opcode: IMPLANT_LOGIN(0x66)

   Windows Version: 5.1 Build 2600 (Service Pack 3)
   Host IP: 192.168.61.129
   Hostname: desktop
   Processor Speed: 2793 MHz
```

Figure 19. Fully Decoded Gh0st Login

The file manager used several different OpCodes depending on the functions it performed. As soon as the file manager was started, the implant transmitted the IMPLANT_DRIVE_LIST (0x67) OpCode and a list of all drives on the compromised

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

system. The message contained the drive letter at offset 0, the drive size at offset 2-5, the drive free space at offset 6-9. These static fields were followed by two variable length strings containing the drive type and file system, delimited by null bytes. As the number of drives and the size of the aforementioned fields were all variable, I used a while loop and an offset variable to iterate through the payload until the offset was equal to the uncompressed size. A nested for loop started from offset 10 and appended characters to the *drive_type* variable until it reached a null byte. I repeated the process for the filesystem field, then used that ending offset as the offset

```
00000000   43 03 F4 27 00 00 16 0E 00 00 4C 6F 63 61 6C 20     C..'......Local
00000010   44 69 73 6B 00 4E 54 46 53 00 44 05 00 00 00 00     Disk.NTFS.D.....
00000020   00 00 00 00 43 44 20 44 72 69 76 65 00 00 45 02     ....CD Drive..E.
00000030   ED 3A 00 00 02 27 00 00 52 65 6D 6F 76 61 62 6C     .:...'..Removabl
00000040   65 20 44 69 73 6B 00 46 41 54 33 32 00 46 02 57     e Disk.FAT32.F.W
00000050   0F 00 00 27 03 00 00 52 65 6D 6F 76 61 62 6C 65     ...'...Removable
00000060   20 44 69 73 6B 00 46 41 54 33 32 00                   Disk.FAT32.
```

Figure 20. Implant Drive List

```
Init string: Gh0st
 Packet Length: 62
 Uncompressed Length: 47
 Opcode: IMPLANT_DRIVE_LIST(0x67)

Drive   Type          FS      Size    Free
C       Local Disk    NTFS    10228   3636
D       CD Drive      0       0       0
E       Removable Disk FAT32  15085   9986
F       Removable Disk FAT32  3927    807
```

Figure 21. Decoded Implant Drive List

```
Init string: Gh0st
 Packet Length: 26
 Uncompressed Length: 5
 Opcode: COMMAND_LIST_FILES(0x02)

C:\.

Init string: Gh0st
 Packet Length: 363
 Uncompressed Length: 497
 Opcode: IMPLANT_FILE_LIST(0x68)

File Name                       Size            Time Stamp
Config.Msi                      0               2016-01-06 14:07:11.905917
Documents and Settings          0               2014-03-11 12:37:33.765625
iDefense                        0               2014-04-05 18:05:59.993013
PDFStreamDumper                 0               2014-07-11 14:52:59.668726
Program Files                   0               2016-01-06 14:06:44.405213
Python27                        0               2014-07-01 19:40:35.095000
RECYCLER                        0               2014-03-14 20:04:07.126738
System Volume Information       0               2014-03-11 12:36:36.343750
WINDOWS                         0               2016-01-06 14:08:32.406250
AUTOEXEC.BAT                    0               2014-03-11 12:26:48.890625
boot.ini                        211             2014-03-11 11:50:32.437500
CONFIG.SYS                      0               2014-03-11 12:26:48.890625
IO.SYS                          0               2014-03-11 12:26:48.890625
MSDOS.SYS                       0               2014-03-11 12:26:48.890625
NTDETECT.COM                    47564           2006-02-28 12:00:00
ntldr                           250048          2014-03-11 16:50:57
pagefile.sys                    1610612736      2016-03-09 00:27:46.687500
```

Figure 22. Decoded Implant File List

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

base offset for the following drive. I used a similar approach to parse the directory listing messages produced by OpCode 0x68 (IMPLANT_FILE_LIST).

The other major functions of the file manager that required special handling were file transfers. Whether the transfers were uploads or downloads, the transfer procedure was essentially identical. When the transfer OpCode was issued (0x03 for uploads or 0x04 for downloads), the message size in bytes was contained between offset 4-7 and the name of the file to be transferred was stored in a null delimited, variable size field beginning at offset 8. I stored both of these values in variables for later use when the transfer was decoded. Opcodes 0x05 and 0x6a indicated the actual files transfers initiated by the previous upload/download commands. As all but the smallest file transfers tend to be split across multiple segments, I used the previously saved size value and the segment offset value contained between offset 4-7 to keep track of how much of the file had been received and display the progress. I continued appending each part of the file to the *file_transfer* variable until the number of bytes transferred equaled the total number of bytes for the file. The reassembled file was stored as a string in the data field of the message array.

The output of the system management interface's process list is similar in structure to those of the drive and directory listing commands. It contains the process ID in the first 4 bytes, followed by two variable-length, null-delimited fields containing the process name and program file name. I again used a pair of for loops to iterate through the characters in the two variable length fields. The program listing output was nearly identical to the process list, but only contained the PID and program name, so it only needed a single *for* loop.

Command shell traffic initially appeared to be standard gh0st traffic with a plain text data section containing all commands entered by the actor in one direction and the output of those commands in the other. On closer inspection it turned out that only the first message in the command shell stream had an OpCode, and all subsequent messages contained only Zlib compressed plain text data beginning at offset 13. With this in mind, I created a variable in the connection *init* function to track whether blobs being processed contained command shell traffic. When OpCode 0x80 was detected, the *command_shell*

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

variable would be set to TRUE. Any subsequent traffic in the connection would be decoded without an OpCode, to produce a transcript of the shell session.

4.3. Handling Output

Once all the blobs in a connection have been decoded, they will be contained in list of message dictionaries that will be passed to the *connectionCloseHandler* for formatting and output. The function also has access to the metadata for the entire connection, which I print on the first several lines, identifying the IP's and ports of the implant and controller and the start and end times for the session. Throughout the function, I used Dshell's custom *out.write* and *out.alert* functions to format and display output, rather than the usual Python *print* or *stdout* commands. These functions allow additional formatting information to be included, such as the direction of traffic to control color coding and HTML formatting tags to include when the output is written to a file.

After printing the connection header, the decoder loops through the message queue for the connection address. In this way, if the controller was conducting multiple concurrent command and control sessions, each would be printed separately. Next, the decoder looks up the

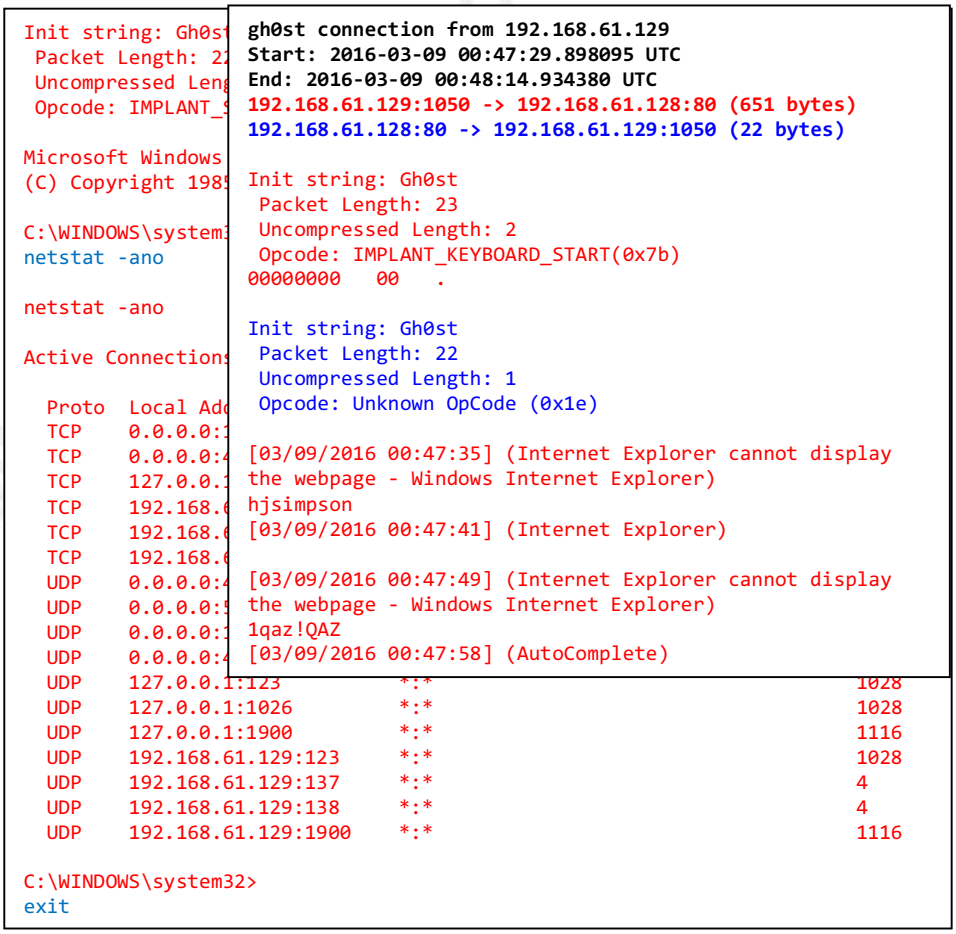


Figure 23. Command Shell Traffic

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

OpCode for each message and determines how to display the data format for that OpCode. If the OpCode indicates command shell traffic, the decoder simply prints the message data to recreate the shell session. The other special handling case was keylogger data, which is sent one character at a time and so is appended to a string variable to be printed once all keylogger messages have been processed.

If neither of these special cases apply, the decoder prints the Gh0st header information, consisting of the init string, compressed and uncompressed packet lengths and the OpCode with its description. Depending on the type of message, it is displayed in a logical format. The system information is printed in parameter: value pairs. Drive, file and directory listings as well as process information are presented in tabular format with headers. I was not able to decode Gh0st's screen capture, audio or video recording traffic, so these OpCodes print only the Gh0st header, but no content.

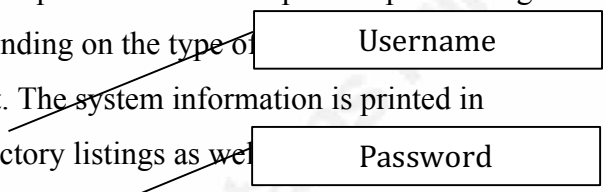


Figure 24. Keylogger Traffic

If an OpCode is not handled by any of the output handers, the data section is printed in hexdump format.

When a file transfer request is detected, the output module will print the file name and size. This will be followed by the actual file transfer, which will often be split across several messages. Each message containing a chunk of the file transfer will display its offset from the

```
Init string: Gh0st
 Packet Length: 77
 Uncompressed Length: 63
 Opcode: COMMAND_DOWN_FILES(0x03)

C:\Documents and Settings\user\My Documents\Trade_Secrets.txt.

Init string: Gh0st
 Packet Length: 82
 Uncompressed Length: 71
 Opcode: IMPLANT_FILE_SIZE(0x69)

C:\Documents and Settings\user\My Documents\Trade_Secrets.txt. (154 Bytes)

Init string: Gh0st
 Packet Length: 24
 Uncompressed Length: 9
 Opcode: COMMAND_CONTINUE(0x07)

00000000   00 00 00 00 00 00 00 00                          ........

Init string: Gh0st
 Packet Length: 153
 Uncompressed Length: 163
 Opcode: IMPLANT_FILE_DATA(0x6a)

Offset: 0 (154/154) Bytes transferred.
C:\Documents and Settings\user\My Documents\Trade_Secrets.txt Complete.
ASCII text, with CRLF line terminators
Writing content to file:
./192.168.61.129_1045_192.168.61.128_80___/C__Documents%20and%20Settings_u
```

Figure 25. Decoded File Transfer

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

start of the file and the number of bytes transferred compared to the total. When the message containing the final chunk of the file is displayed, the decoder will indicate that the transfer completed. Due to the way the file was reconstructed by the blobHandler, the entire file will be contained in the data section of this message. The decoder will not attempt to display the file by default, as many files are binary and do not display well in shell output. Instead, the decoder attempts to use the Python "magic" library to determine and print the file type, similar to the Linux command line "file" utility. The magic library is not installed by default and is not a Dshell dependency, so I wrapped this import in a try block that simply omits this line of output if the module is not available. If desired, the python-magic package can be installed using the Ubuntu apt-get package manager. If the user has supplied the –gh0st_dumpfiles option, the transferred file will be written to a subdirectory of the current working directory, named with the source and destination IPs and ports of the client and server. The files will be saved with their original name and full directory path, replacing slashes, backslashes and colons with underscores. This approach prevents files with the same name, but different directories from overwriting each other.

5. Conclusion

Scripting has long been a powerful tool in an analyst's toolkit and the Dshell framework allows for the rapid creation of scripts to analyze network traffic. While the standard decoders included with Dshell are extremely useful in decoding a variety of network traffic and standard protocols, the true power of Dshell is its extensibility. By leveraging Python libraries to do the heavy lifting of abstracting raw network traffic into easily useable data structures, Dshell allows an analyst to focus on the structure and content of the traffic instead of tedious and easily automated tasks like connection tracking and stream reassembly. It also allows a platform to incrementally develop decoders as more is learned from each step of the process. In the test case, writing a simple decoder to reverse the Zlib compression illuminated the contents of the payload and allowed further analysis of the OpCode and payload that would have previously been unreadable. With each section I added to the decoder, I ran it against the traffic samples and saw another piece of the puzzle come into focus. Dshell decoders can also be used

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

for creating analytics to answer questions, aggregate data or locate anomalies in the traffic being analyzed.

While writing the Gh0st decoder, I had the significant advantages of having access to Spohn's detailed malware analysis, a copy of the source code and perhaps most importantly, a working controller and implant I could test in a virtual network. This allowed me to see the output of each command I issued and match it to the network traffic I observed. This allowed me to identify and decode data structures in the traffic I might not have otherwise deduced had it been a truly new and undocumented protocol. Of course, analysts should make use of every resource at their disposal. If a malware sample is available, it can provide a great deal of insight into decoding its communication protocol. Information from strings can sometimes identify keys, imported libraries or function calls can help identify encryption algorithms or encoding schemes and a disassembler like Ida Pro can help identify the data structures encoded in messages. Oftentimes, there will be elements of a command and control protocol that are never fully identified or decoded, but even a partial decoder can provide insight into an attacker's activity that would not have otherwise been available. Analysts can share decoders they develop among members of their team and/or submit them to the GitHub repository for use by other members of the community. The source code of the completed Gh0st decoder is included in Appendix F and has been submitted to the GitHub Dshell repository.

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

References

Meta, Lohit. (2015). *Gh0st RAT: Complete Malware Analysis*. Washington, DC: Infosec Institute. Retrieved from http://resources.infosecinstitute.com/gh0st-rat-complete-malware-analysis-part-1/.

Spohn, Michael G. (2012). *Know Your Digital Enemy: Anatomy of a Gh0st RAT*. McAfee Foundstone. Retrieved from www.mcafee.com/us/resources/white-papers/foundstone/wp-know-your-digital-enemy.pdf.

SecDev Group. (2009, March 29). *Tracking GhostNet: Investigating a Cyber Espionage Network*. Information Warfare Monitor. Retrieved from https://www.nsi.org/pdf/reports/Cyber%20Espionage%20Network.pdf.

US Army Research Lab. (2016). *Dshell*. GitHub. Retrieved from https://github.com/USArmyResearchLab/Dshell.

Richter, P., Chatzis, N., Smaragdakis, G., Feldmann, A., & Willinger, W. (2015). *Distilling the Internet's Application Mix from Packet-Sampled Traffic*. NYU Polytechnic Institute Passive and Active Measurement Conference 2015. Retrieved from http://wan.poly.edu/pam2015/papers/50.pdf.

Wells, Christopher J. *Encapsulation of data in the TCP/IP protocol stack* [Diagram]. TechnologyUK. Retrieved from http://www.technologyuk.net/the_internet/internet/tcp_ip_stack.shtml.

Sincoder. (2013, May 8). *gh0st*. GitHub. Retrieved from https://github.com/sincoder/gh0st.

Grosfelt, Justin. (2104, January 15). *Command and Control Encryption*. RSA Blog. Retrieved from https://blogs.rsa.com/command-control-encryption-part-1/.

Stewart, Amanda. (2014, May 14). *PlugX: The Old Dog with a New Trick.* FireEye Blog. Retrieved from https://www.fireeye.com/blog/threat-research/2013/05/targeted-attack-trend-alert-plugx-the-old-dog-with-a-new-trick.html.

Hammer, Richard. (2006, May 25). *Inside-Out Vulnerabilities, Reverse Shells.* SANS Institute. Retrieved from https://www.sans.org/reading-room/whitepapers/covert/inside-out-vulnerabilities-reverse-shells-1663.

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Appendix A

Dshell Usage

```
Usage: decode [options] [decoder options] file1 file2 ... filen [-- [decoder args]+]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -d DECODER, --decoder=DECODER
                        Use a specific decoder module
  -C CONFIG, --config=CONFIG
                        specify config.ini file
  -l, --ls, --list      List all available decoders
  --usage               display usage examples
  --status              show status
  --tmpdir=TMPDIR       alternate temp directory (for use when processing
                        compressed pcap files)
  -r, --recursive       recursively process all PCAP files under input
                        directory

  Multiprocessing options:
    -p, --parallel      process multiple files in parallel
    -t, --threaded      run multiple decoders in parallel
    -n NUMPROCS, --nprocs=NUMPROCS
                        number of simultaneous processes

  Input options:
    -i INTERFACE, --interface=INTERFACE
                        listen live on INTERFACE
    -c COUNT, --count=COUNT
                        number of packets to process
    -f BPF, --bpf=BPF   replace default decoder filter (use carefully)
    --nofilterfn        Set filterfn to pass-thru
    -F FILEFILTER       Use filefilter as input for the filter expression.  An
                        additional expression given on the command line is
                        ignored.
    --ebpf=EBPF         BPF filter to exclude traffic, extends other filters
    --vlan              examine traffic which has VLAN headers present
    --layer2=LAYER2     select the layer-2 protocol module
    --strip=STRIPLAYERS
                        extra data-link layers to strip

  Output options:
    -o OUTFILE, --outfile=OUTFILE
                        write output to the file OUTFILE, add pcap=PCAPFILE to
                        write packets, session=SESSION to write session
                        text...
    -W PCAP, --pcap=PCAP
                        output decoded packets to PCAP (same as -o pcap=....)
    --db=DB             output to db. Supply "config=file" or
                        "param=...,param=..."
    --oformat=OFORMAT   define the output format
    -x, --extra         output a lot of extra information
    -w SESSION, --session=SESSION
                        write session file, same as -o session=
    -O OUTPUT, --output=OUTPUT
                        Use a custom output module. Supply
                        "modulename,option=value,..."

  Logging options:
    -L LOGFILE, --logfile=LOGFILE
                        log to file
    --debug             debug logging (debug may also affect decoding
                        behavior)
    -v, --verbose       verbose logging
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Appendix B

Dshell Decoder List

Name	Protocol	Description
dhcp	UDP	Extract client information from DHCP messages • Default filter: (udp and port 67)
dns	TCP	Extract and summarize DNS queries/responses (defaults: A,AAAA,CNAME,PTR records) • Default filter: (udp and port 53) • dns decoder options: --dns_only_noanswer report only unanswered queries --dns_show_noanswer report unanswered queries alongside other queries --dns_show_norequest report unsolicited responses alongside other responses --dns_only_norequest report only unsolicited responses --dns_showall show all answered queries/responses
dns-asn	TCP	identify AS of DNS A/AAAA record responses • Default filter: (port 53)
dns-cc	TCP	identify country code of DNS A/AAAA record responses • Default filter: (port 53) • dns-cc decoder options: --dns-cc_code=DNS-CC_CODE filter on a specific country code (ex. US) --dns-cc_foreign report responses in foreign countries
innuendo-dns	TCP	proof-of-concept detector for INNUENDO DNS channel • Default filter: (port 53)
reservedips	TCP	identify DNS resolutions that fall into reserved ip space • Default filter: (port 53)
asn-filter	TCP	filter connections on autonomous system number (ASN) • Chainable • Default filter: ip or ip6 • asn-filter decoder options: --asn-filter_alerts print alerts to screen --asn-filter_asn=ASN-FILTER_ASN asn for client or server
country	TCP	filter connections on geolocation (2-letter country code) • Chainable • Default filter: ip or ip6 • country decoder options: --country_both both (client/server) ARE in specified country --country_code=COUNTRY_CODE Either (client/server) IS in specified country --country_alerts Print alerts to screen --country_notboth

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

		specified country is not both client and server --country_neither neither (client/server) is in specified country
snort	RAW	filter packets by snort rule • Chainable • Default filter: ip or ip6 • snort decoder options: --snort_none pass if NO rules matched --snort_alerts alert if rule matched --snort_conf=SNORT_CONF snort.conf file to read --snort_rule=SNORT_RULE snort rule to filter packets --snort_all all rules must match to pass
track	TCP	tracked activity recorder • Chainable • Default filter: ip • track decoder options: --track_source=TRACK_SOURCE --track_alerts --track_target=TRACK_TARGET
large-flows	TCP	display netflows that have at least 1MB transferred • Default filter: tcp • large-flows decoder options: --large-flows_size=LARGE-FLOWS_SIZE number of megabytes transferred
long-flows	TCP	display netflows that have a duration of at least 5mins • Default filter: tcp or udp • long-flows decoder options: --long-flows_len=LONG-FLOWS_LEN set minimum connection time to alert on, in minutes [default: 5 mins]
netflow	TCP	generate netflow information from pcap • Default filter: tcp or udp • netflow decoder options: --netflow_group=NETFLOW_GROUP
reverse-flow	TCP	Generate an alert if the client transmits more data than the server • Default filter: tcp or udp • reverse-flow decoder options: --reverse-flow_threshold=REVERSE-FLOW_THRESHOLD Alerts if client transmits more than threshold times the data of the server --reverse-flow_zero alert if the server transmits zero bytes [default: false] --reverse-flow_minimum=REVERSE-FLOW_MINIMUM alert on client transmissions larger than min bytes [default: 0]
ftp	TCP	ftp • Default filter: tcp • ftp decoder options --ftp_port=FTP_PORT

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

		Port (or ports) to watch for control connections (Default: 21) --ftp_dump Dump files (Default: Off)
flash-detect	TCP	Detects successful Flash file download. • Chainable • Default filter: tcp and (port 80 or port 8080 or port 8000) • flash-detect decoder options: --flash-detect_ignore_handshake ignore TCP handshake --flash-detect_md5sum=FLASH-DETECT_MD5SUM Calculate and print the md5sum of the file. There are three options: 0: (default) No md5sum calculations or labeling 1: Calculate md5sum; Print out md5sum in alert; Name all dumped files by their md5sum (must be used with 'dump' option) 2: Calculate md5sum; Print out md5sum in alert; If found, a file's explicitly listed save name (found in 'content-disposition' HTTP header) will be used for file dump name instead of md5sum. Any other numbers will be ignored and the default action will be used. --flash-detect_dump Dump the flash file to a file based off its name, md5sum (if specified), or its URI. The file is dumped to the local directory "flashout". The file extension is ".flash" to prevent accidental execution.
httpdump	TCP	Dump useful information about HTTP sessions • Default filter: tcp and (port 80 or port 8080 or port 8000) • httpdump decoder options: --httpdump_showcontent Display response BODY. --httpdump_maxpost=HTTPDUMP_MAXPOST Truncate POST body longer than max chars. Set to 0 for no truncating. (default: 1000) --httpdump_urlfilter=HTTPDUMP_URLFILTER Filter to URLs matching this regex --httpdump_maxurilen=HTTPDUMP_MAXURILEN Truncate URLs longer than max len. Set to 0 for no truncating. (default: 30) --httpdump_showhtml Display response BODY only if HTML.
joomla-cve-2015-8562	TCP	detect and dissect malformed HTTP headers targeting Joomla CVE-2015-8562 • Default filter: tcp and (port 80 or port 8080 or port 8000) • joomla-cve-2015-8562 decoder options: --joomla-cve-2015-8562_raw_payload return the raw payload (do not attempt to decode chr encoding)
ms15-034	TCP	detect attempts to enumerate MS15-034 vulnerable IIS servers • Default filter: tcp and (port 80 or port 8080 or port 8000)
rip-http	TCP	rip files from HTTP traffic • Default filter: tcp and port 80 • rip-http decoder options: --rip-http_direction=RIP-HTTP_DIRECTION

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

		cs=only capture client POST sc=only capture server GET response --rip-http_content_filter=RIP-HTTP_CONTENT_FILTER regex MIME type filter for files to save --rip-http_name_filter=RIP-HTTP_NAME_FILTER regex filename filter for files to save --rip-http_append_conn append sourceip-destip to filename --rip-http_append_ts append timestamp to filename --rip-http_ignore_handshake ignore TCP handshake --rip-http_outdir=DIRECTORY directory to write output files (Default: current directory)
web	TCP	Improved version of web that tracks server response • Default filter: tcp and (port 80 or port 8080 or port 8000) • web decoder options: --web_maxurilen=WEB_MAXURILEN Truncate URLs longer than max len. Set to 0 for no truncating. (default: 30) --web_md5 calculate MD5 for each response. Available in CSV output.
emdivi_c2	TCP	deobfuscate Emdivi http c2 • Default filter: tcp and port 80
followstream	TCP	Generates color-coded Screen/HTML output similar to Wireshark Follow Stream • Default filter: tcp followstream decoder options: --followstream_ignore_handshake ignore TCP handshake --followstream_encoding=FOLLOWSTREAM_ENCODING attempt to interpret text as encoded with specified schema --followstream_hex two-column hex/ascii output --followstream_time include timestamp for each blob
grep	TCP	Search for patterns in streams. • Chainable • Default filter: tcp • grep decoder options: --grep_singleline Treat entire connection as single line of text. --grep_invert For chained only: Invert hit results. --grep_ignorecase Case insensitive search. --grep_iterate Iterate hits on match string. --grep_expression=GREP_EXPRESSION Search expression
merge	RAW	dump all packets to single file • Chainable • Default filter: none
synrst	RAW	detect failed attempts to connect (SYN followed by a RST/ACK)

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

		• Default filter: tcp[13]=2 or tcp[13]=20
writer	RAW	pcap/session writer • Default filter: none • writer decoder options: --writer_filename=WRITER_FILENAME
xor	TCP	XOR an entire stream with a given single byte key • Chainable • Default filter: tcp • xor decoder options: --xor_cskey=XOR_CSKEY c->s xor key [default None] --xor_resync resync if the key is seen in the stream --xor_sckey=XOR_SCKEY s->c xor key [default None] --xor_key=XOR_KEY xor key [default 255]
ether	RAW	raw ethernet capture decoder • Default filter: none
ip	RAW	IPv4/IPv6 decoder • Default filter: ip or ip6
protocol	RAW	Identifies non-standard protocols (not tcp, udp or icmp) • Default filter: (ip and not tcp and not udp and not icmp)
psexec	TCP	Extract command/response information from psexec over smb • Default filter: tcp and (port 445 or port 139) • psexec decoder options: --psexec_alertsonly only dump alerts, not content --psexec_htmlalert include html as named value in alerts --psexec_time display command/response timestamps
rip-smb-uploads	TCP	Extract files uploaded via SMB • Default filter: tcp and port 445 • rip-smb-uploads decoder options: --rip-smb-uploads_outdir=DIRECTORY Directory to place files (default: ./smb_out)
smbfiles	TCP	List files accessed via smb • Default filter: tcp and (port 445 or port 139) • smbfiles decoder options: --smbfiles_nopsexec supress psexecsvc streams from output --smbfiles_activeonly only output files with reads or writes
tftp	RAW	Find TFTP streams and, optionally, extract the files • Default filter: udp • tftp decoder options: --tftp_rip Rip files from traffic (default: off) --tftp_outdir=DIRECTORY Directory to place files when using --rip

Appendix C

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Decoder Templates

PacketDecoder.py

```
#!/usr/bin/env python

import dshell
import output
import util


class DshellDecoder(dshell.IPDecoder):

    '''generic packet-level decoder template'''

    def __init__(self, **kwargs):
        '''decoder-specific config'''

        '''pairs of 'option':{option-config}'''
        self.optiondict = {}

        '''bpf filter, for ipV4'''
        self.filter = ''
        '''filter function'''
        # self.filterfn=

        '''init superclasses'''
        self.__super__().__init__(**kwargs)

    def packetHandler(self, ip):
        '''handle as Packet() ojects'''
        pass

# create an instance at load-time
dObj = DshellDecoder()
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

SessionDecoder.py

```
#!/usr/bin/env python

import dshell
import output
import util


class DshellDecoder(dshell.TCPDecoder):

    '''generic session-level decoder template'''

    def __init__(self, **kwargs):
        '''decoder-specific config'''

        '''pairs of 'option':{option-config}'''
        self.optiondict = {}

        '''bpf filter, for ipV4'''
        self.filter = ''
        '''filter function'''
        # self.filterfn=

        '''init superclasses'''
        self.__super__().__init__(**kwargs)

    def packetHandler(self, udp, data):
        '''handle UDP as Packet(),payload data
                remove this if you want to make UDP into pseudo-sessions'''
        pass

    def connectionInitHandler(self, conn):
        '''called when connection starts, before any data'''
        pass

    def blobHandler(self, conn, blob):
        '''handle session data as soon as reassembly is possible'''
        pass

    def connectionHandler(self, conn):
        '''handle session once all data is reassembled'''
        pass

    def connectionCloseHandler(self, conn):
        '''called when connection ends, after data is handled'''

# create an instance at load-time
dObj = DshellDecoder()
```

Appendix D

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Gh0st RAT Codes

Command Code	Hex
COMMAND_ACTIVED	0x00
COMMAND_LIST_DRIVE	0x01
COMMAND_LIST_FILES	0x02
COMMAND_DOWN_FILES	0x03
COMMAND_UPLOAD_FILE	0x04
COMMAND_FILE_DATA	0x05
COMMAND_EXCEPTION	0x06
COMMAND_CONTINUE	0x07
COMMAND_STOP	0x08
COMMAND_DELETE_FILE	0x0a
COMMAND_DELETE_DIRECTORY	0x10
COMMAND_SET_TRANSFER_MODE	0x11
COMMAND_CREATE_FOLDER	0x12
COMMAND_RENAME_FILE	0x13
COMMAND_OPEN_FILE_SHOW	0x14
COMMAND_OPEN_FILE_HIDE	0x15
COMMAND_SCREEN_SPY	0x16
COMMAND_SCREEN_RESET	0x17
COMMAND_ALGORITHM_RESET	0x18
COMMAND_SCREEN_CTRL_ALT_DEL	0x19
COMMAND_SCREEN_CONTROL	0x20
COMMAND_SCREEN_BLOCK_INPUT	0x21
COMMAND_SCREEN_BLANK	0x22
COMMAND_SCREEN_CAPTURE_LAYER	0x23
COMMAND_SCREEN_GET_CLIPBOARD	0x24
COMMAND_SCREEN_SET_CLIPBOARD	0x25
COMMAND_WEBCAM	0x26
COMMAND_WEBCAM_ENABLECOMPRESS	0x27
COMMAND_WEBCAM_DISABLECOMPRESS	0x28
COMMAND_WEBCAM_RESIZE	0x29
COMMAND_NEXT	0x30
COMMAND_KEYBOARD	0x31
COMMAND_KEYBOARD_OFFLINE	0x32
COMMAND_KEYBOARD_CLEAR	0x33
COMMAND_AUDIO	0x34
COMMAND_SYSTEM	0x35
COMMAND_PSLIST	0x36
COMMAND_WSLIST	0x37

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

COMMAND_DIALUPASS	0x38
COMMAND_KILLPROCESS	0x39
COMMAND_SHELL	0x40
COMMAND_SESSION	0x41
COMMAND_REMOVE	0x42
COMMAND_DOWN_EXEC	0x43
COMMAND_UPDATE_SERVER	0x44
COMMAND_CLEAN_EVENT	0x45
COMMAND_OPEN_URL_HIDE	0x46
COMMAND_OPEN_URL_SHOW	0x47
COMMAND_RENAME_REMARK	0x48
COMMAND_REPLAY_HEARTBEAT	0x49

Token Code	**Hex**
IMPLANT_AUTH	0x64
IMPLANT_HEARTBEAT	0x65
IMPLANT_LOGIN	0x66
IMPLANT_DRIVE_LIST	0x67
IMPLANT_FILE_LIST	0x68
IMPLANT_FILE_SIZE	0x69
IMPLANT_FILE_DATA	0x6A
IMPLANT_TRANSFER_FINISH	0x6B
IMPLANT_DELETE_FINISH	0x6C
IMPLANT_GET_TRANSFER_MODE	0x6D
IMPLANT_GET_FILEDATA	0x6E
IMPLANT_CREATEFOLDER_FINISH	0x6F
IMPLANT_DATA_CONTINUE	0x70
IMPLANT_RENAME_FINISH	0x71
IMPLANT_EXCEPTION	0x72
IMPLANT_BITMAPINFO	0x73
IMPLANT_FIRSTSCREEN	0x74
IMPLANT_NEXTSCREEN	0x75
IMPLANT_CLIPBOARD_TEXT	0x76
IMPLANT_WEBCAM_BITMAPINFO	0x77
IMPLANT_WEBCAM_DIB	0x78
IMPLANT_AUDIO_START	0x79
IMPLANT_AUDIO_DATA	0x7A
IMPLANT_KEYBOARD_START	0x7B
IMPLANT_KEYBOARD_DATA	0x7C
IMPLANT_PSLIST	0x7D
IMPLANT_WSLIST	0x7E

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

IMPLANT_DIALUPASS	0x7F
IMPLANT_SHELL_START	0x80

Transfer Mode	**Hex**
TRANSFER_MODE_NORMAL	0x00
TRANSFER_MODE_ADDITION	0x01
TRANSFER_MODE_ADDITION_ALL	0x02
TRANSFER_MODE_OVERWRITE	0x03
TRANSFER_MODE_OVERWRITE_ALL	0x04
TRANSFER_MODE_JUMP	0x05
TRANSFER_MODE_JUMP_ALL	0x06
TRANSFER_MODE_CANCEL	0x07

©2016 SANS Institute, Author retains full rights.

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Appendix E

Gh0st RAT Proto-Decoder

```
#!/usr/bin/env python

import dshell
import util
import zlib
import os
import datetime
import struct
import colorout
import collections


class DshellDecoder(dshell.TCPDecoder):

    def __init__(self, **kwargs):
        dshell.TCPDecoder.__init__(self,
            name='gh0st',
                    description='Gh0st RAT malware C2',
                    author='dmm',
                    filter='tcp and port 80',
                    optiondict={},
              ),

        self.out = colorout.ColorOutput()
        self.connectionCount = 0
        if 'setColorMode' in dir(self.out):
            self.out.setColorMode()
        self.msgqueue = {}

    def connectionInitHandler(self, conn):
        self.msgqueue[conn.addr] = []

    def blobHandler(self, conn, blob):
        #list of messages within the blob
        msglist = []
        #create a sorted dictionary to keep segments in correct order
        segments = collections.OrderedDict(sorted(blob.segments.items()))
        #iterate through the sorted segments in blob
        for k,v in segments.iteritems():
            #v is a list of a single string, so grab the string
            i = v[0]
            #dictionary to hold message
            msg = {}
            #decompress zlib compressed message body
            try:
                msg['data'] = i[:13] + zlib.decompress(i[13:])
            except:
                msg['data'] = i
            #direction of message (server->client/client->server)
            msg['direction'] = blob.direction
            #append the decoded message to the message list
            msglist.append(msg)
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

```
        if len(msglist) == 0:
            self.debug("No messages in this blob.")
            return
        #append the message list for this blob to the queue for the connection
        self.msgqueue[conn.addr].extend(msglist)

    def connectionCloseHandler(self, conn):
        if len(self.msgqueue[conn.addr]) == 0: pass #nothing to write
        else:
            #Print connection information
            self.out.write("%s connection from %s\n" % (self.name, conn.clientip),
formatTag='H1')
            self.out.write("Start: %s UTC\n  End: %s UTC\n" %
(datetime.datetime.utcfromtimestamp(conn.starttime),
datetime.datetime.utcfromtimestamp(conn.endtime)), formatTag='H2')
            self.out.write("%s:%s -> %s:%s (%d bytes)\n" % (conn.clientip,
conn.clientport, conn.serverip, conn.serverport, conn.clientbytes),
formatTag="H2", direction="cs")
            self.out.write("%s:%s -> %s:%s (%d bytes)\n\n" % (conn.serverip,
conn.serverport, conn.clientip, conn.clientport, conn.serverbytes),
formatTag="H2", direction="sc")
            #loop through messages in connection
            for m in self.msgqueue[conn.addr]:
                #output the now-uncompressed message body as hex
                self.out.write(m['data'], hex=True, direction=m['direction'])
                self.out.write("\n")

# create an instance at load-time
dObj = DshellDecoder()
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Appendix F

Completed Gh0st RAT Decoder

```
#!/usr/bin/env python

import dshell
import util
import zlib
import os
import struct
import colorout
import datetime
import socket
import collections

class DshellDecoder(dshell.TCPDecoder):

    def __init__(self, **kwargs):
        dshell.TCPDecoder.__init__(self,
            name='gh0st',
                        description='Gh0st RAT malware C2',
                        author='dmm',
                        filter='tcp and port 80',
                        optiondict={
                'alertsonly':{'action':'store_true','help':'only dump alerts, not content'},
                'ignoreinit':{'action':'store_true','help':'ignore the gh0st init string and
attempt to decode all matching traffic'},
                'dumpfiles':{'action':'store_true','help':'dump files to disk'},
                },
                ),

        self.connectionCount = 0
        self.msgqueue = {}
        self.out = colorout.ColorOutput()
        self.dumpdir = {}
        self.dumpfile_basepath = '.'
        if not os.access(self.dumpfile_basepath, os.W_OK):
                        try:
                                import tempfile
                                self.dumpfile_basepath = tempfile.mkdtemp()
                        except:
                                self.out.warn("Unable to write to current directory or create a
temporary directory for file dumping.")
                                self.dumpfiles = False
        self.initStrings = ["gh0st", "7hero", "Adobe", "B1X6Z", "BEiLa", "BeiJi", "ByShe",
"FKJP3", "FLYNN", "FWAPR", "FWKJG", "GWRAT", "Gh0st", "GOLDt", "HEART", "HTTPS", "HXWAN", "Heart",
"IM007", "ITore", "KOBBX", "KrisR", "LUCKK", "LURK0", "LYRAT", "Level", "Lover", "Lyyyy", "MYFYB",
"MoZhe", "MyRat", "OXXMM", "PCRat", "QWPOT", "Spidern", "Tyjhu", "URATU", "W0LFKO", "Wangz",
"Winds", "World", "X6RAT", "XDAPR", "Xjjhj", "ag0ft", "attac", "cb1st", "https", "whmhl", "xhjyk",
"00000", "ABCDE", "apach", "Assas", "Blues", "chevr", "CHINA", "cyl22", "DrAgOn", "EXXMM",
"Eyes1", "Gi0st", "GM110", "Hello", "httpx", "kaGni", "light", "LkxCq", "lvxYT", "Naver", "NIGHT",
"NoNul", "Origi", "QQ_124971919", "Snown", "SocKt", "Super", "Sw@rd", "v2010", "VGTLS", "wcker",
"Wh0vt", "wings", "X6M9K", "xqwf7", "YANGZ"]
        self.opcodes={"64": "IMPLANT_AUTH", "65": "IMPLANT_HEARTBEAT", "66": "IMPLANT_LOGIN",
"67": "IMPLANT_DRIVE_LIST", "68": "IMPLANT_FILE_LIST", "69": "IMPLANT_FILE_SIZE", "6a":
"IMPLANT_FILE_DATA", "6b": "IMPLANT_TRANSFER_FINISH", "6c": "IMPLANT_DELETE_FINISH", "6d":
"IMPLANT_GET_TRANSFER_MODE", "6e": "IMPLANT_GET_FILEDATA", "6f": "IMPLANT_CREATEFOLDER_FINISH",
"70": "IMPLANT_DATA_CONTINUE", "71": "IMPLANT_RENAME_FINISH", "72": "IMPLANT_EXCEPTION", "73":
"IMPLANT_BITMAPINFO", "74": "IMPLANT_FIRSTSCREEN", "75": "IMPLANT_NEXTSCREEN", "76":
"IMPLANT_CLIPBOARD_TEXT", "77": "IMPLANT_WEBCAM_BITMAPINFO", "78": "IMPLANT_WEBCAM_DIB", "79":
"IMPLANT_AUDIO_START", "7a": "IMPLANT_AUDIO_DATA", "7b": "IMPLANT_KEYBOARD_START", "7c":
"IMPLANT_KEYBOARD_DATA", "7d": "IMPLANT_PSLIST", "7e": "IMPLANT_WSLIST", "7f":
"IMPLANT_DIALUPASS", "80": "IMPLANT_SHELL_START", "00": "COMMAND_ACTIVED", "01":
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

```
"COMMAND_LIST_DRIVE", "02": "COMMAND_LIST_FILES", "03": "COMMAND_DOWN_FILES", "04":
"COMMAND_UPLOAD_FILE", "05": "COMMAND_FILE_DATA", "06": "COMMAND_EXCEPTION", "07":
"COMMAND_CONTINUE", "08": "COMMAND_STOP", "0a": "COMMAND_DELETE_FILE", "10":
"COMMAND_DELETE_DIRECTORY", "11": "COMMAND_SET_TRANSFER_MODE", "12": "COMMAND_CREATE_FOLDER",
"13": "COMMAND_RENAME_FILE", "14": "COMMAND_OPEN_FILE_SHOW", "15": "COMMAND_OPEN_FILE_HIDE", "16":
"COMMAND_SCREEN_SPY", "17": "COMMAND_SCREEN_RESET", "18": "COMMAND_ALGORITHM_RESET", "19":
"COMMAND_SCREEN_CTRL_ALT_DEL", "20": "COMMAND_SCREEN_CONTROL", "21": "COMMAND_SCREEN_BLOCK_INPUT",
"22": "COMMAND_SCREEN_BLANK", "23": "COMMAND_SCREEN_CAPTURE_LAYER", "24":
"COMMAND_SCREEN_GET_CLIPBOARD", "25": "COMMAND_SCREEN_SET_CLIPBOARD", "26": "COMMAND_WEBCAM",
"27": "COMMAND_WEBCAM_ENABLECOMPRESS", "28": "COMMAND_WEBCAM_DISABLECOMPRESS", "29":
"COMMAND_WEBCAM_RESIZE", "30": "COMMAND_NEXT", "31": "COMMAND_KEYBOARD", "32":
"COMMAND_KEYBOARD_OFFLINE", "33": "COMMAND_KEYBOARD_CLEAR", "34": "COMMAND_AUDIO", "35":
"COMMAND_SYSTEM", "36": "COMMAND_PSLIST", "37": "COMMAND_WSLIST", "38": "COMMAND_DIALUPASS", "39":
"COMMAND_KILLPROCESS", "40": "COMMAND_SHELL", "41": "COMMAND_SESSION", "42": "COMMAND_REMOVE",
"43": "COMMAND_DOWN_EXEC", "44": "COMMAND_UPDATE_SERVER", "45": "COMMAND_CLEAN_EVENT", "46":
"COMMAND_OPEN_URL_HIDE", "47": "COMMAND_OPEN_URL_SHOW", "48": "COMMAND_RENAME_REMARK", "49":
"COMMAND_REPLAY_HEARTBEAT", "00": "COMMAND_SHELL_TRAFFIC", "FF": "ZLIB_DECOMPRESSION_FAILED"}

    def connectionInitHandler(self, conn):
        self.msgqueue[conn.addr] = []
        self.command_shell = False
        self.file_transfer = ""
        self.size_total = 0
        self.size_transferred = 0
        self.transfer_name = ""
        if self.dumpfiles:
            self.dumpdir[conn.addr] = os.path.join(self.dumpfile_basepath, "%s_%s_%s_%s" %
(conn.sip, conn.sport, conn.dip, conn.dport))
            while os.path.exists(self.dumpdir[conn.addr]): self.dumpdir[conn.addr] += '_'

    def blobHandler(self, conn, blob):
        msglist = []
        opcode = "00"
        longmsg = {}
        #create a sorted dictionary to keep segments in correct order
        segments = collections.OrderedDict(sorted(blob.segments.items()))
        #iterate through segments in blob
        for k,v in segments.iteritems():
            for i in v:
                #skip segments not contain gh0st header
                if not ((i[0:5] in self.initStrings) and self.ignoreinit): pass
                #dictionary to hold message
                msg = {}
                #decompress zlib compressed message body
                try:
                    data = zlib.decompress(i[13:])
                    #handle command shell traffic that does not have an opcode
                    if self.command_shell:
                        msg['opcode'] = "00"
                        msg['data'] = data[0:]
                    #otherwise extract opcode and rest of message data
                    else:
                        msg['opcode'] = data[0].encode('hex')
                        msg['data'] = data[1:]
                #if zlib decompression fails, message probably spans multiple segments
                except:
                    #this is the start of the long message
                    if i[0:5] in self.initStrings:
                        longmsg['init'] = i[0:5]
                        longmsg['data'] = i[13:]
                        longmsg['pLength'] = struct.unpack("<L", i[5:9])[0]
                        longmsg['uLength'] = struct.unpack("<L", i[9:13])[0]
                    #this is a continuation with no header
                    else:
                        try:
                            longmsg['data'] += i
                        except:
                            longmsg['data'] = i
                    try:
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

```
                        data = zlib.decompress(longmsg['data'])
                        longmsg['opcode'] = data[0].encode('hex')
                        longmsg['data'] = data[1:]
                        msg = longmsg
                        longmsg = {}
                    except:
                        continue
                #direction of message (server->client/client->server)
                msg['direction'] = blob.direction
                #gh0st init string
                msg['init'] = i[0:5]
                #convert packed (little-endian hex) compressed message length to decimal
                try:
                    msg['pLength'] = struct.unpack("<L", i[5:9])[0]
                except:
                    msg['pLength'] = 0
                #convert packed (little-endian hex) uncompressed message length to decimal
                try:
                    msg['uLength'] = struct.unpack("<L", i[9:13])[0]
                except:
                    msg['uLength'] = 0
                #Extract system information from implant initial login message
                if msg['opcode'] == "66":
                    #extract Windows version information
                    msg['version'] = "%d.%d Build %d (%s)" % (int(msg['data'][7].encode('hex')),
int(msg['data'][11].encode('hex')), struct.unpack('=H', msg['data'][15:17])[0],
msg['data'][23:37])
                    #extract host IP and convert to dotted decimal notation
                    msg['host_ip'] = socket.inet_ntoa(msg['data'][163:167])
                    #extract hostname and strip trailing nulls
                    msg['hostname'] = msg['data'][167:217].rstrip("\x00")
                    #extract processer speed in MHz
                    msg['procspeed'] = struct.unpack('=H', msg['data'][159:161])[0]
                #Parse Drive Listing
                if msg['opcode'] == "67":
                    drives = []
                    o = 0
                    while o < len(msg['data']):
                        drive_letter = msg['data'][o]
                        drive_size = struct.unpack("<L", msg['data'][o+2:o+6])[0]
                        drive_free = struct.unpack("<L", msg['data'][o+6:o+10])[0]
                        drive_type = ""
                        drive_fs = ""
                        o += 10
                        for char in msg['data'][o:]:
                            if char == "\x00":
                                o += 1
                                break
                            else:
                                drive_type += char
                        o += len(drive_type)
                        for char in msg['data'][o:]:
                            if char == "\x00":
                                o += 1
                                break
                            else:
                                drive_fs += char
                        o = o + len(drive_fs)
                        drives.append([drive_letter, drive_size, drive_free, drive_type,
drive_fs])
                    msg['drives'] = drives
                #Parse directory listing
                if msg['opcode'] == "68":
                    files = []
                    dirs = []
                    o = 0
                    while o < len(msg['data']):
                        file_type = msg['data'][o].encode('hex')
                        file_name = ""
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

```
                        o += 1
                        for char in msg['data'][o:]:
                            if char == "\x00":
                                #o += 1
                                break
                            else:
                                file_name += char
                        o = o + len(file_name) + 5
                        if len(file_name) < 8:
                            file_name += "\t\t\t"
                        elif len(file_name) < 16:
                            file_name += "\t\t"
                        elif len(file_name) < 24:
                            file_name += "\t"
                        file_size = str(struct.unpack("<L", msg['data'][o:o+4])[0])
                        o += 4
                        if len(file_size) < 8:
                            file_size += "\t"
                        dt = (struct.unpack("<Q", msg['data'][o:o+8])[0] / 10)
                        file_time = datetime.datetime(1601,1,1) +
datetime.timedelta(microseconds=dt)
                        o += 8
                        if file_type == "00":
                            files.append([file_name, file_size, file_time])
                        if file_type == "10":
                            dirs.append([file_name, file_size, file_time])
                    msg['files'] = files
                    msg['dirs'] = dirs
                #Parse File Upload/Download commands
                if msg['opcode'] == "04" or msg['opcode'] == "69":
                    msg['size'] = struct.unpack("<L", msg['data'][4:8])[0]
                    self.size_total = msg['size']
                    msg['data'] = msg['data'][8:]
                    self.transfer_name = msg['data'].rstrip("\x00")
                #Parse file transfers 05/6a
                if msg['opcode'] == "05" or msg['opcode'] == "6a":
                    msg['total'] = self.size_total
                    msg['offset'] = struct.unpack("<L", msg['data'][4:8])[0]
                    self.file_transfer += msg['data'][8:]
                    self.size_transferred += len(msg['data'][8:])
                    msg['transferred'] = self.size_transferred
                    msg['file_name'] = self.transfer_name
                    if self.size_total == self.size_transferred:
                        self.size_total = 0
                        self.size_transferred = 0
                        msg['data'] = self.file_transfer
                        self.file_transfer = ""
                        self.transfer_name = ""
                #parse process listing
                if msg['opcode'] == "7d":
                    procs = []
                    o = 0
                    while o < len(msg['data']):
                        proc_name = ""
                        proc_file = ""
                        pid = struct.unpack("<L", msg['data'][o:o+4])[0]
                        o += 4
                        for char in msg['data'][o:]:
                            if char == "\x00":
                                o += 1
                                break
                            else:
                                proc_name += char
                        o += len(proc_name)
                        for char in msg['data'][o:]:
                            if char == "\x00":
                                o += 1
                                break
                            else:
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

```
                            proc_file += char
                        o += len(proc_file)
                        if len(proc_name) < 8:
                            proc_name += "\t\t"
                        elif len(proc_name) < 16:
                            proc_name += "\t"
                        procs.append([pid, proc_name, proc_file])
                    msg['procs'] = procs
                #parse program listing
                if msg['opcode'] == "7e":
                    procs = []
                    o = 0
                    while o < len(msg['data']):
                        proc_name = ""
                        proc_file = ""
                        pid = struct.unpack("<L", msg['data'][o:o+4])[0]
                        o += 4
                        for char in msg['data'][o:]:
                            if char == "\x00":
                                o += 1
                                break
                            else:
                                proc_name += char
                        o += len(proc_name)
                        procs.append([pid, proc_name])
                    msg['procs'] = procs
                #If command shell initiated, set command_shell flag, remainder of blob will be
treated as shell commands/responses
                if msg['opcode'] == "80":
                    self.command_shell = True
                msglist.append(msg)
        if len(msglist) == 0:
            self.debug("No messages in this blob.")
            return
        self.msgqueue[conn.addr].extend(msglist)

    def connectionCloseHandler(self, conn):
        #out = self.out
        if len(self.msgqueue[conn.addr]) == 0: pass#self.self.out.write("No %s found in this
stream.\n" % self.name)
        else:
            #create string to concatenate keylogger data from multiple segments
            keylogger_string = ""
            #Print connection information
            if self.alertsonly:
                self.alert(conn.info())
            else:
                self.out.write("%s connection from %s\n" % (self.name, conn.clientip),
formatTag='H1')
                self.out.write("Start: %s UTC\n  End: %s UTC\n" %
(datetime.datetime.utcfromtimestamp(conn.starttime),
datetime.datetime.utcfromtimestamp(conn.endtime)), formatTag='H2')
                self.out.write("%s:%s -> %s:%s (%d bytes)\n" % (conn.clientip, conn.clientport,
conn.serverip, conn.serverport, conn.clientbytes), formatTag="H2", direction="cs")
                self.out.write("%s:%s -> %s:%s (%d bytes)\n\n" % (conn.serverip, conn.serverport,
conn.clientip, conn.clientport, conn.serverbytes), formatTag="H2", direction="sc")
                #loop through messages in connection
                for m in self.msgqueue[conn.addr]:
                    opcode = m['opcode']
                    #enrich message with opcode description
                    if (m['opcode'] in self.opcodes): m['opcode'] = self.opcodes[opcode] + "(0x" +
opcode + ")"
                    else: m['opcode'] = "Unknown OpCode (0x" + opcode + ")"
                    #
                    # Handle commands with nonstandard output formats
                    #
                    #Format output for command shell traffic
                    if opcode == "00":
                        self.out.write(m['data'], direction=m['direction'])
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

```
                    #Append keystroke to keylogger string
                    elif opcode == "7c":
                        keylogger_string += m['data']
                    else:
                        #Print standard information
                        self.out.write("Init string: %s \n Packet Length: %d \n Uncompressed
Length: %d \n Opcode: %s \n\n" % (m['init'], m['pLength'], m['uLength'], m['opcode']),
direction=m['direction'])
                        #Print system information from implant login message
                        if opcode == "66":
                            self.out.write("   Windows Version: %s\n" % m['version'],
direction=m['direction'])
                            self.out.write("   Host IP: %s \n" %(m['host_ip']),
direction=m['direction'])
                            self.out.write("   Hostname: %s \n" %(m['hostname']),
direction=m['direction'])
                            self.out.write("   Processor Speed: %d MHz\n" %(m['procspeed']),
direction=m['direction'])
                        elif opcode in ("02", "03", "0a", "6c"):
                            self.out.write(m['data'], direction=m['direction'])
                            self.out.write("\n")
                        #Format drive listings
                        elif opcode == "67":
                            self.out.write("Drive\tType\t\tFS\tSize\tFree\n",
direction=m['direction'])
                            for drive in m['drives']:
                                self.out.write("%s\t%s\t%s\t%s\t%s\n" % (drive[0], drive[3],
drive[4], drive[1], drive[2]), direction=m['direction'])
                        #Format file/directory listings
                        elif opcode == "68":
                            self.out.write("File Name\t\t\tSize\t\tTime Stamp\n",
direction=m['direction'])
                            for d in m['dirs']:
                                self.out.write("%s\t%s\t%s\n" % (d[0], d[1], d[2]),
direction=m['direction'])
                            for f in m['files']:
                                self.out.write("%s\t%s\t%s\n" % (f[0], f[1], f[2]),
direction=m['direction'])
                        #Format/print process listings
                        elif opcode == "7d":
                            self.out.write("PID\tProcess Name\t\tFile\n",
direction=m['direction'])
                            for proc in m['procs']:
                                self.out.write("%s\t%s\t%s\n" % (proc[0], proc[1], proc[2]),
direction=m['direction'])
                        #Format/print program listings
                        elif opcode == "7e":
                            self.out.write("PID\tProgram Name\n", direction=m['direction'])
                            for proc in m['procs']:
                                self.out.write("%s\t%s\n" % (proc[0], proc[1]),
direction=m['direction'])
                        #Print name and size of file to be transferred
                        elif opcode == "04" or opcode == "69":
                            self.out.write("%s (%d Bytes)\n" % (m['data'], m['size']),
direction=m['direction'])
                        #Handle file transfers
                        elif opcode == "05" or opcode == "6a":
                            #Try to import libmagic to extract exif data
                            try:
                                import magic
                                magic_is_real = True
                            except:
                                magic_is_real = False
                            #Show status of file transfer reconstruction
                            self.out.write("Offset: %d (%d/%d) Bytes transferred.\n" %
(m['offset'],m['transferred'], m['total']), direction=m['direction'])
                            if m['total'] == m['transferred']:
                                #Check if python-magic library is available and if so, print file
exif data
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

```
                                if magic_is_real:
                                    mymagic = magic.open(magic.MAGIC_RAW)
                                    mymagic.load()
                                    mtype = mymagic.buffer(m['data'])
                                    self.out.write("%s Complete.\n%s\n\n" % (m['file_name'],
mtype), direction=m['direction'])
                                #Write file to disk if --gh0st_dumpfiles switch is used
                                if self.dumpfiles:
                                    localfilename = self.localfilename(self.dumpdir[conn.addr],
m['file_name'])
                                    self.out.write("Writing content to file:\n%s" % localfilename,
direction=m['direction'])
                                    fh = open(localfilename, "w")
                                    fh.write(m['data'])
                                    fh.close()
                                self.out.write("\n")
                            #Don't write out screen captures we can't decode
                            elif opcode in("73", "74", "75"):
                                pass
                            #If we don't have a parser for opcode, output as hexdump
                            else:
                                self.out.write(m['data'], hex=True, direction=m['direction'])
                        self.out.write("\n")
                    #Print keylogger data after it has been reassembled
                    self.out.write("%s\n" % keylogger_string.lstrip(), direction=m['direction'])

    def localfilename(self, path, origname):
        #Check if otput path exists and create it if not
        if not os.path.exists(path): os.mkdir(path)
        #Replace offensive characters
        tmp = origname.replace("\\", "_")
        tmp = tmp.replace("/", "_")
        tmp = tmp.replace(":", "_")
        localname = ''
        for c in tmp:
            if ord(c) > 32 and ord(c) < 127:
                localname += c
            else:
                localname += "%%%02X" % ord(c)
        localname = path + '/' + localname
        while os.path.exists(localname): localname += '_'
        return localname

# create an instance at load-time
dObj = DshellDecoder()
```

David Martin,
dmartin@mastersprogram.sans.edu

© 2016 The SANS Institute Author retains full rights.

Last Updated: May 22nd, 2020

Upcoming SANS Training

Click here to view a list of all SANS Courses

SANS Amsterdam August 2020	**Amsterdam, NL**	**Aug 03, 2020 - Aug 08, 2020**	**Live Event**
SANS OnDemand	**OnlineUS**	**Anytime**	**Self Paced**
SANS SelfStudy	**Books & MP3s OnlyUS**	**Anytime**	**Self Paced**