National Cyber Security Centre
a part of GCHQ

Busy Buzzard

Malware Analysis Report

Version 1.0

25 March 2022

National Cyber Security Centre
a part of GCHQ

Busy Buzzard

A plugin-based Windows remote access tool

Executive summary

- Busy Buzzard is deployed as an RC4-encrypted Windows DLL, with a shellcode stub to decrypt and load it into memory.
- Command and control (C2) communication is either sent over HTTPS with RC4 encryption and Base64 encoding of tasking or sent directly over raw TCP with RC4 encryption of tasking.
- The malware provides the capability to load and execute shellcode or Windows DLL plugins.

Introduction

Four samples were uploaded to Virus Total, by a user in Vietnam, in August 2021 (see the 'Malware details (Metadata)' section). The naming convention indicated that these files were memory dumps, uploaded in two parts, from two separate processes.

The first memory dump for each process contained the same shellcode loader stub, immediately followed by the same malicious Windows 64-bit DLL (decrypted and running in memory). This DLL was identified as the Busy Buzzard malware.

The second memory dump for each process was found to contain a suspected Windows 64-bit shellcode plugin for the Busy Buzzard malware.

Further searches in Virus Total identified two separate variants of the Busy Buzzard malware, one of which uses raw TCP-based command and control (C2) communications, and the other of which uses HTTPS-based C2 communications. Compilation timestamps indicate that the Busy Buzzard malware may have been in use since at least 2016.

National Cyber Security Centre
a part of GCHQ

Malware details

Metadata

Filename	process.0xffffa88b75bfd840.0x2b28af80000.dmp
Description	Process memory dump comprising shellcode loader stub and Busy Buzzard implant DLL (TCP variant)
Size	77823 bytes
MD5	bbc49eac5b7c30708704233416694591
SHA-1	8fd99d9066020003358aa3e23c9af3d4911ce979
SHA-256	41daf4c86e14da87bf2f94b36115a1e7da76d14af0aba0c251bb3e9dbfb40bad
Compile time	28th July 2016, 10:07:26 UTC (Busy Buzzard implant DLL)

Filename	process.0xffffa88b75bfd840.0x2b28b200000.dmp
Description	Busy Buzzard shellcode module
Size	12287 bytes
MD5	c5af2332d8f7bdd56ed2ae0091422153
SHA-1	f737067d41bc77dc7dd09ecb6eb710619bc2dfde
SHA-256	6cf6d1a9caee970bcb393a085d1dbb1f01a81fa684f6faf7ddbf0253302e1a4e

Filename	svchost.exe.3016.2b28af80000-2b28af92fff.dmp
Description	Process memory dump comprising shellcode loader stub and Busy Buzzard implant DLL (TCP variant)
Size	77824 bytes
MD5	d40a4f0b426b5500d0e7e331f99c6aca
SHA-1	266852db4ad2d293469515820fd5e7c228cd4b3e
SHA-256	79024943b61d9c7fe7f8e225f2825ee4fbdeb6dcf2ecdfda3f6414bd6f87bf32
Compile time	28th July 2016, 10:07:26 UTC (Busy Buzzard implant DLL)

Filename	svchost.exe.3016.2b28b200000-2b28b202fff.dmp
Description	Busy Buzzard shellcode module
Size	12288 bytes
MD5	f11471c0667eb010a319bb4765ed72c7
SHA-1	cf6339501de54590f8bbbc3cfb8051b95f6a1a42
SHA-256	bd6992029c879b74b255aeb3549b8da487aff75d3f614832c23b4cd3717a067b

Filename	83030f299a776114878bcd2ade585d97836ef4ddb6943cb796be2c88bcb83a83.sample
Description	Busy Buzzard implant DLL (TCP variant)
Size	71831 bytes
MD5	c0e649fa591ed6c5746d394cb2de3c72
SHA-1	d2b8f4fe6eedb8b87521772fc823da596f2403b7
SHA-256	83030f299a776114878bcd2ade585d97836ef4ddb6943cb796be2c88bcb83a83
Compile time	10th June 2019, 07:58:10 UTC

Filename	0b182464a2351a9d79c1222bb1fdf35e.dll
Description	Busy Buzzard implant DLL (HTTPS variant)
Size	68096 bytes
MD5	0b182464a2351a9d79c1222bb1fdf35e
SHA-1	6a673508d46c0bbff74ee24384c8bc841c11ea4d
SHA-256	6b52fd7ee1442b4ed2c675f958a42a6c793bfe14a75de0988c4381367284f085
Compile time	7th January 2019, 01:33:18 UTC

Filename	10000000.dll
Description	Busy Buzzard implant DLL (HTTPS variant)
Size	68096 bytes
MD5	037261d5571813b9640921afac8aafbe
SHA-1	e74affd6c766156e3fe803917f28da08fe7000ef
SHA-256	9d6e14cd244f6c49e11d2b47f12116b5848aaed7a6aaa218fb023b33f7c12a3b
Compile time	7th January 2019, 01:33:18 UTC

Filename	80f55.rec.dll
Description	Busy Buzzard implant DLL (HTTPS variant)
Size	127147 bytes
MD5	c5994f9fe4f58c38a8d2af3021028310
SHA-1	48152eeb1d74a84ba86b34f419cf1c7a105e41ff
SHA-256	ca9bcf268330a4fffcec025920514e0071651c35895b15b2f1dab8813c8b8e99
Compile time	7th January 2019, 01:33:18 UTC

National Cyber Security Centre
a part of GCHQ

MITRE ATT&CK®

This report has been compiled with respect to the MITRE ATT&CK® framework, a globally accessible knowledge base of adversary tactics and techniques based on real-world observations.

Tactic	ID	Technique	Procedure
Execution	T1129	Shared Modules	Busy Buzzard uses the `LoadLibraryW` and `GetProcAddress` API functions to load additional modules and execute functions from them.
Execution	T1106	Native API	Busy Buzzard uses the `VirtualAlloc` and `CreateThread` API functions to inject and execute shellcode plugins.
Defense Evasion	T1055.009	Process Injection: Proc Memory	Busy Buzzard is injected into the memory of a process, with additional shellcode modules also injected into the memory of the same process.
Defense Evasion	T1497.001	Virtualization/ Sandbox Evasion: System Checks	Busy Buzzard tests the Windows Registry for the presence of a key associated with VMware. The HTTPS variant also tests for the presence of the VMXh port (associated with VMware), using the `IN` assembly instruction.
Discovery	T1083	File and Directory Discovery	The Busy Buzzard shellcode plugin provides the capability to browse files and directories.
Discovery	T1082	System Information Discovery	The Busy Buzzard beacon contains OS, username, hostname and process information.
Discovery	T1016	System Network Configuration Discovery	The Busy Buzzard beacon contains the victim IP address.
Discovery	T1124	System Time Discovery	The Busy Buzzard beacon contains the time at which the implant was started.
Command And Control	T1071.001	Application Layer Protocol: Web Protocols	Busy Buzzard (HTTPS variant) uses HTTPS protocol for C2 communication.
Command And Control	T1095	Non-Application Layer Protocol	Busy Buzzard (TCP variant) uses raw TCP protocol for C2 communication.
Command And Control	T1573	Encrypted Channel	The Busy Buzzard beacon is RSA-encrypted using a hard-coded public key. Other C2 communications are RC4-encrypted using a shared, randomly generated, key.

Functionality

Overview

Two variants of the Busy Buzzard malware were identified, one of which uses raw TCP-based C2 communications (referred to hereafter as 'the TCP variant'), and the other of which uses HTTPS-based C2 communications (referred to hereafter as 'the HTTPS variant').

Two different versions of the TCP variant were identified, compiled in July 2016 and June 2019 respectively. The versions are broadly similar in functionality with the main differences being in the implementation of the C2 communications (these differences are described in more detail in the 'Communications (Command and control)' section).

The samples of the HTTPS variant were all compiled in January 2019, indicating that this variant is most likely operated concurrently with the later version of the TCP variant.

The malware starts by checking whether it is executing within a VMware virtualised environment (see the 'Functionality (Anti-analysis techniques)' section), exiting if this is the case.

This is most likely an attempt to hinder sandbox analysis by security researchers since VMware is often used for dynamic malware analysis.

The TCP variant generates a unique mutex name associated with the malware. If a mutex with this name is found to already exist then the malware exits, ensuring that only a single instance of the malware is running at any given time. A description of how the mutex name is generated can be found in the 'Functionality (Mutex name generation)' section.

The malware then creates a structure containing pointers to the following 'helper' functions, that is passed to shellcode plugins (note that function names have been assigned by the report author):

- `send_data_to_c2` (sends data to the C2 server).
- `connect_to_c2` (connects a socket to the C2 server).
- `rc4_crypt` (encrypts or decrypts using a previously shared RC4 key).
- `calc_crc32` (calculates the CRC32 checksum of a sequence of bytes).
- `get_c2_beacon_response` (gets the value returned by the C2 server in response to an initial malware beacon).
- `send_data_to_c2_and_get_response` (sends data to the C2 server and then receives the response).

Finally, the malware enters the main C2 loop (see the 'Communications (Command and control)' section).

Decrypted commands are represented by a single byte value that determines the action to be taken by the malware. The HTTPS variant responds to commands identified by the byte values `0x64` ('`d`') and `0x73` ('`s`'). In addition to these, the TCP variant also responds to commands identified by the byte values `0x66` ('`f`') and `0x6c` ('`l`'). The commands provide the following functionality:

Command ID	Description

0x64 ('d')	Loads a DLL, specified as a UTF-16 string, into memory using the Windows API function `LoadLibraryW` and executes a function, specified as an ASCII string, from the DLL. The function can be referred to either by name or ordinal value. Any arguments to be passed to the function are specified after the library name and function name/ordinal. For example to call the function `myFunc` with the argument `Hello World`, from the library `myLib.dll`, the command data would appear as follows:
	
	`d\x00m\x00y\x00L\x00i\x00b\x00.\x00d\x00l\x00l\x00 \x00myFunc\x00Hello World\x00`
	
	where `\x00` is a separating null byte. Similarly to call a function with the ordinal value `2`, from the same library and with the same argument as above, the command data would appear as follows:
	
	`d\x00m\x00y\x00L\x00i\x00b\x00.\x00d\x00l\x00l\x00 \x002\x00Hello World\x00`

0x73 ('s')	Copies the specified blob of shellcode into memory and executes it, passing the address of the structure of 'helper' function pointers as an argument.
0x66 ('f')	Stores a 4-byte value that is sent by the C2 server in response to the initial malware beacon (see the 'Communications (Command and control)' section).
0x6c ('l')	Sets the interval (seconds) between keep-alive malware beacons (see the 'Communications (Command and control)' section). The interval is specified as a 4-byte integer value.

Table 1: Busy Buzzard command IDs and descriptions

Initial execution

The Busy Buzzard malware is deployed as a payload composed of a shellcode loader stub, immediately followed by an RC4-encrypted malicious Windows DLL. The exact mechanism used to initially execute the malware payload cannot be determined from the samples available, however open-source reporting[1] indicates that it is most likely to involve the use of a separate loader.

The shellcode stub begins by walking the Export Table of `kernel32.dll`, locating the Windows API functions `LoadLibraryA` and `GetProcAddress` by name. These two functions are then used to resolve Windows API functions that are required to load Busy Buzzard into memory, with the library and function names being constructed directly on the stack. The shellcode stub is followed by a sequence of 8 NOP (`0x90`) bytes, a 4-byte value that defines the size of the Busy Buzzard Windows DLL, the 16-byte RC4 key used for decryption, and finally the encrypted Busy Buzzard Windows DLL itself.

The shellcode stub extracts the size of the Busy Buzzard Windows DLL and the RC4 key for decryption using hard-coded address offsets. This information is then used to decrypt the Busy Buzzard Windows DLL before reflectively loading it into memory.

It should be possible to automatically extract Busy Buzzard Windows DLLs by searching for the sequence of NOP (`0x90`) bytes. This location can then be used to extract the size of the Busy Buzzard Windows DLL and the RC4 key used for decryption.

[1] http://jsac.jpcert.or.jp/archive/2021/pdf/JSAC2021_202_niwa-yanagishita_en.pdf

Mutex name generation

The TCP variant creates a mutex name by firstly calculating the CRC32 checksum of a range of bytes composed of a hard-coded RSA public key (used for C2 communications) plus 11 additional hard-coded bytes (see Figure 1). This value is then converted to a hex string in reverse byte order. The mutex name is this hex string with a `0x01` byte appended to it.

It isn't clear why the malware includes a `0x01` byte at the end of the generated mutex name. Named mutexes typically shouldn't include non-printable ASCII characters, so this could potentially be used to easily identify mutex names associated with the Busy Buzzard malware.

Using the example shown in Figure 1, the CRC32 checksum (as calculated over the indicated bytes) is `0xB5B364A0`. Converting to a hex string in reverse byte order results in the string `0A463B5B`. After appending a `0x01` byte to this string the generated mutex name, whose existence will be checked by the malware, is `0A463B5B\x01` (see the 'Detection (Indicators of compromise)' section).

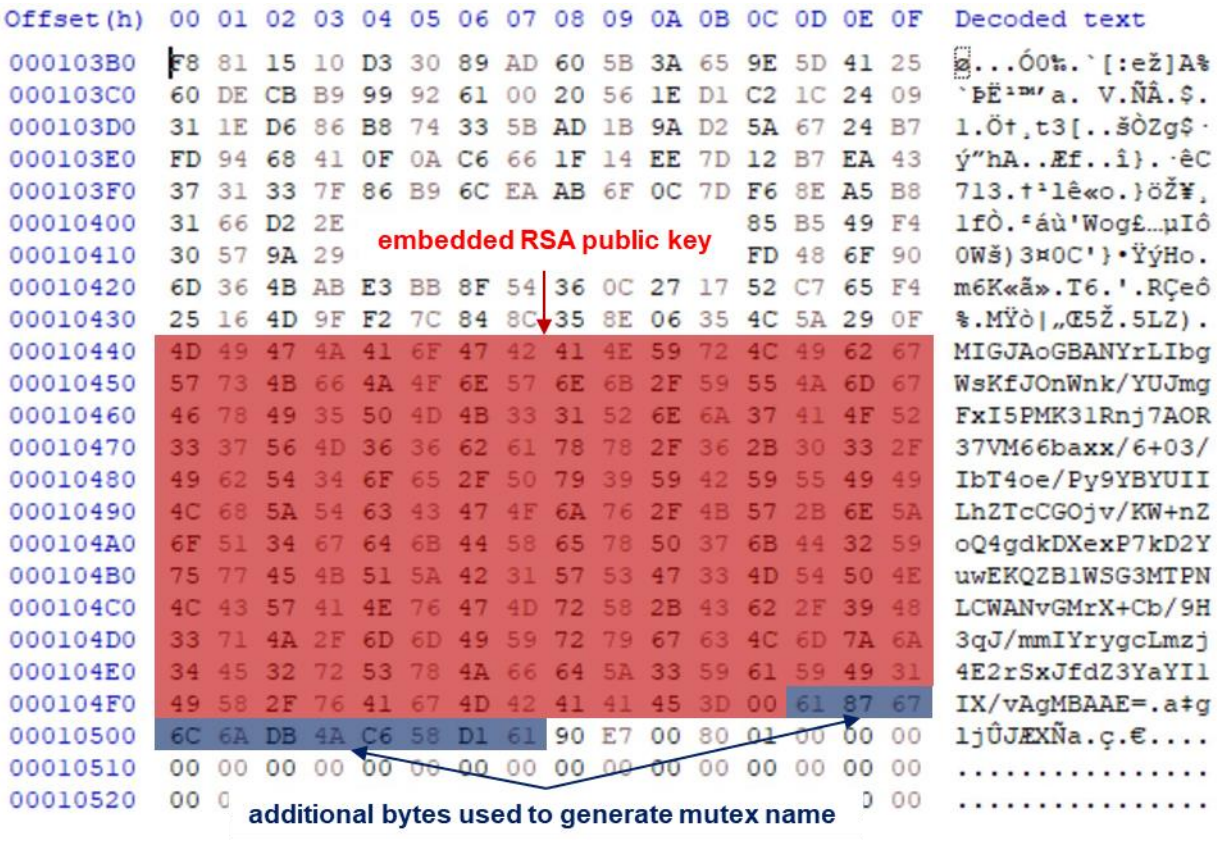

Figure 1: Bytes used to generate raw TCP variant mutex name

Defence evasion

Both the shellcode loader stub and the malware construct notable strings directly on the stack. This is most likely used to hinder basic static analysis.

Both the TCP and HTTPS variants check whether a key with the name `HKEY_CLASSES_ROOT\Applications\VMwareHostOpen.exe` is present within the Windows Registry. The HTTPS variant also attempts to read data from the `VX` virtual I/O port and checks whether this returns the value `VMXh`. These artefacts uniquely identify that the malware is executing within a VMware virtualised environment. If either of these checks is successful then the malware exits, which is likely intended to hinder dynamic analysis.

The HTTPS variant starts a separate thread that continually checks whether the network has been disconnected or the malware is being debugged, exiting if either of these is true.

Shellcode plugin

The shellcode plugin starts by resolving the base address of `kernel32.dll` and using this to locate the Windows API functions `GetProcAddress` and `LoadLibraryA` by the ROR13[2] hash of their name. These two functions are then used to resolve further Windows API functions based upon the ROR13 hash of their name. The names of additional libraries, needed to resolve the Windows API functions, are constructed directly on the stack.

The samples have been extracted from the memory of a running process; therefore the Windows API functions have already been resolved by the shellcode plugin. Windows API functions within `kernel32.dll` can be inferred because the shellcode plugin stores its base address. The addresses for the `VirtualAlloc` and `VirtualFree` Windows API functions can also be inferred based on the arguments being passed to them.

The version of `kernel32.dll` being utilised within the victim environment can be identified by matching the known offsets to the `GetProcAddress`, `LoadLibraryA`, `VirtualAlloc` and `VirtualFree` Windows API functions. A YARA rule to perform this matching can be found in the 'Appendix (YARA rule used to identify kernel32.dll)' section.

A sample of `kernel32.dll` was identified using this YARA rule, which enabled addresses for the following Windows API functions to be identified within the shellcode plugin:

- `FindFirstFileW`
- `FindNextFileW`
- `FindClose`
- `GetLogicalDriveStringsA`
- `GetDriveTypeA`
- `GetDiskFreeSpaceExA`
- `DeleteFileW`
- `MoveFileW`
- `RemoveDirectoryW`

The shellcode plugin initially enumerates all available disk drives on the host and lists the contents of the directory from which the malware is executing. The shellcode plugin appears to enable the actors to remotely browse the host filesystem and delete files and directories. The shellcode plugin doesn't appear to provide any capability to exfiltrate files and directories from the host. It is likely that this is provided by a separate shellcode plugin.

The plugin uses the 'helper' functions from the malware to create a new socket connection to the C2 server, for receiving plugin-specific commands and sending responses back. The plugin-specific commands and responses are RC4-encrypted using a randomly generated RC4 key (see the 'Communications (Command and control)' and 'Communications (RC4 key generation)' sections). The address of a structure containing 'helper' function pointers is passed as an argument when the malware executes the plugin.

[2] https://www.mandiant.com/resources/precalculated-string-hashes-reverse-engineering-shellcode

Communications

Command and control

Busy Buzzard malware command and control (C2) communication is either sent over HTTPS with RC4 encryption and Base64 encoding of tasking or sent directly over raw TCP with RC4 encryption of tasking. Key generation is described further in the 'Communications (RC4 key generation)' section.

The C2 communication for both the TCP and HTTPS variants of the Busy Buzzard malware are described in greater detail in the following sections.

HTTPS variant

The sequence of communications between the Busy Buzzard malware HTTPS variant and C2 server is illustrated in Figure 2 and described below.

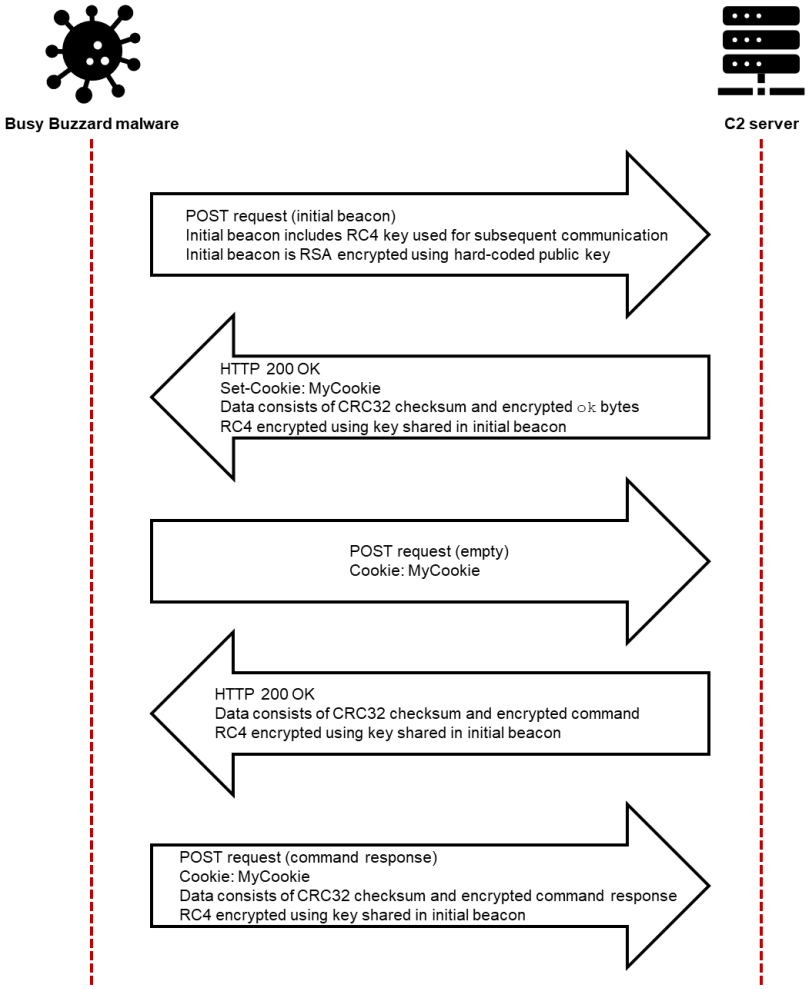

Figure 2: Communication between the Busy Buzzard malware HTTPS variant and C2 server

The Busy Buzzard malware HTTPS variant sends beacons to the C2 server at 30 second intervals. The beacon data is formatted using HTTP POST URI parameters, as follows:

- `b=<Hostname>|<Username>|<PID>&t=YY/MM/DD HH:MM:SS&r=<RC4 key>`
- The `b` parameter is composed of the following values separated with the pipe character (‘`|`’):
  - Hostname.
  - Logged on username.
  - PID of the process within whose context the malware is executing.
- The `t` parameter indicates the date/time at which the malware started operating, in the format:
  - `YY/MM/DD HH:MM:SS`
- The `r` parameter provides a randomly generated RC4 key (between 8 and 16 ASCII characters in length) used to encrypt malware tasking and responses.

The beacon data is RSA encrypted using Windows Cryptography API functions and a hard-coded RSA public key. After being encrypted the beacon data is Base64-encoded before being sent to the C2 server via a POST request. The malware will continue to beacon until it receives a `200 OK` response from the C2 server, containing a `Set-Cookie` header field and 6 bytes of response data.

The first 4 bytes of the response data consist of a CRC32 checksum value for the remainder of the response data after it has been decrypted using the previously shared RC4 key. The decrypted response should consist of the string `ok`. The CRC32 checksum value associated with the string `ok` is `0x79dcdd47`.

Once the correct response has been received from the C2 server, the malware will respond with an empty POST request with the `Cookie` header field set to the value previously specified by the `Set-Cookie` header field from the C2 server response. The malware expects to receive a `200 OK` response from the C2 server, with the response data containing tasking to be executed.

As before, the first 4 bytes of the response data consist of a CRC32 checksum value for the remainder of the response data, after it has been decrypted using the previously shared RC4 key. The first byte of the decrypted response specifies the command to be executed by the malware, followed by additional command-specific parameters.

Once the requested command has been carried out and a response sent back to the C2 server, the malware loops and begins the sequence of beaconing and receiving commands again. The RC4 key used to encrypt tasking and responses is re-generated each time the malware loops, so each sequence of beacon/tasking/response is encrypted differently.

The malware either uses a hard-coded `User-Agent` string for POST requests sent to the C2 server, or the default system `User-Agent` string for the host. Where the `User-Agent` string was found to be hard-coded, the value was:

```
Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; WOW64; Trident/4.0;
SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media
Center PC 6.0; .NET4.0C; .NET4.0E)
```

TCP variant

The sequence of communications between the Busy Buzzard malware TCP variant and C2 server is illustrated in Figure 3 and described below.

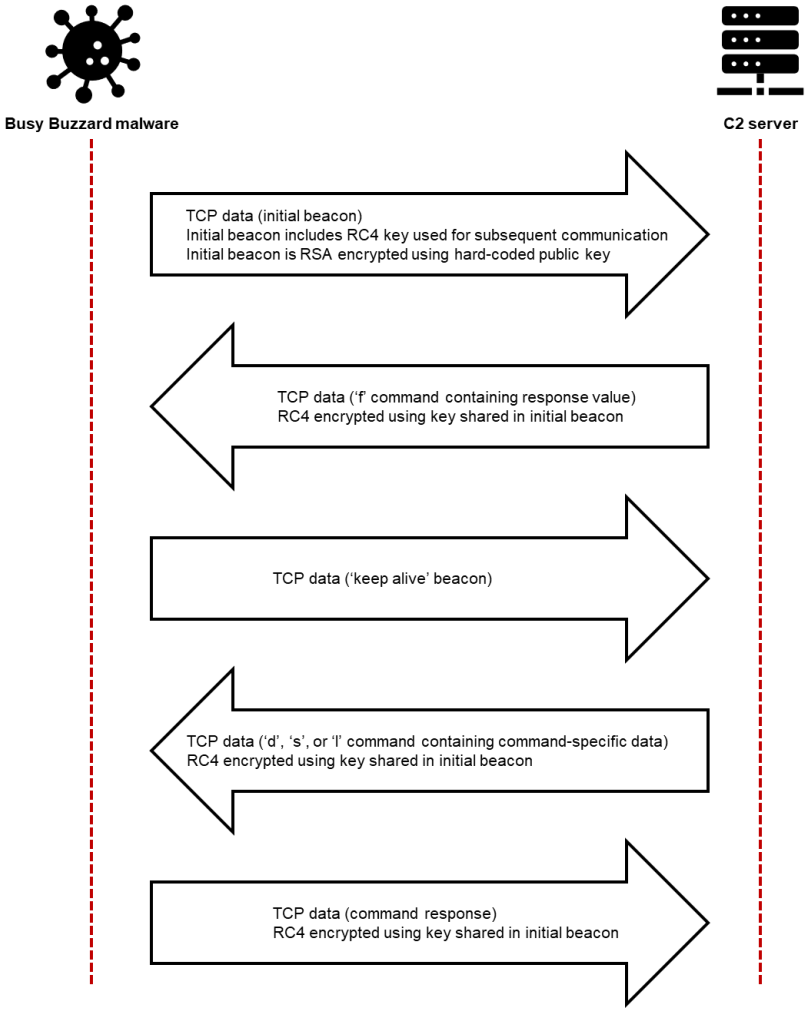

Figure 3: Communication between the Busy Buzzard malware TCP variant and C2 server

The Busy Buzzard malware TCP variant sends beacons to the C2 server at 60 second intervals. The beacon data is composed of a contiguous series of binary records, containing similar information to the HTTPS variant. The binary records typically use type-length-value (TLV) encoding as shown in Table 2. The full set of supported record types are described in Table 3. Record types `0x4` (Process record) and `0x8` (IP address) do not use TLV encoding and are instead fixed-length records.

Record format		
AA BB CC_1 CC_2 … CC_N		
data type	data length	data value

Table 2: TLV record format

Type field	Data Length	Data field
0x03	Variable defined by 1-byte length field	Username string (UTF-16, maximum 32-bytes)
0x04	Fixed 15-byte length	Process record (see Table 4)
0x05	Variable defined by 1-byte length field	Timestamp string (ASCII, formatted as 'YYYY/MM/DD HH:MM:SS')
0x06	Variable defined by 1-byte length field	RC4 key string (ASCII, maximum 16-bytes)
0x07	Variable defined by 1-byte length field	Hostname string (UTF-16, maximum 32-bytes)
0x08	Fixed 4-byte length	IP address (formatted as 4 binary octets), populated using the Windows API function `getsockname`

Table 3: Busy Buzzard beacon record types

Process record							
04 AA AA AA AA BB **40** CC DD EE FF FF GG GG HH HH							
process ID	privilege level	host architecture	OS major version	OS minor version	OS build number	system type	OS type

Table 4: Busy Buzzard beacon process record format

- `privilege level` is set to `0x1` if the malware is executing as `LOCAL SYSTEM`, `0x2` if the malware is executing as `ADMIN`, or `0x3` otherwise.
- The remaining fields are populated with system data retrieved using the Windows API functions `GetNativeSystemInfo`, `RtlGetVersion` and `GetProductInfo`.

The beacon data is RSA encrypted using Windows Cryptography API functions and a hard-coded RSA public key. The earlier version of the TCP variant (compiled in July 2016) appends at least 5 bytes of random padding to the beacon data before it is sent to the C2 server.

Tasking from the C2 server is expected to be formatted as follows:

C2 tasking format		
AA AA AA AA BB BB BB BB CC_1 CC_2 ... CC_N		
length of tasking data	crc32 of decrypted tasking data	encrypted tasking data

Table 5: Busy Buzzard C2 tasking format

The earlier version (compiled in July 2016) of the TCP variant expects the `length of tasking data` value to be RC4-encrypted using the previously shared RC4 key, however later versions do not appear to encrypt this field. In all observed TCP variants, the tasking data is RC4-encrypted using the previously shared RC4 key. The first byte of the decrypted tasking data specifies the command to be executed by the malware, followed by additional command-specific parameters (see the 'Functionality (Overview)' section).

The malware will continue to beacon until it receives tasking from the C2 server that specifies the `0x66` (‘`f`’) command followed by a non-zero 4-byte value. Once this command has been received the malware will respond to further tasking from the C2 server and start to send ‘keep alive’ beacons to the C2 server.

The ‘keep alive’ beacons are sent regularly at a default interval of 60 seconds. The command `0x6c` (‘`l`’) can be used to modify the ‘keep alive’ beacon interval. The ‘keep alive’ beacons contain randomised data and are formatted as follows:

‘keep alive’ beacon format	
AA AA AA AA FF BB_1 BB_2 ... BB_N	
length of beacon data	random bytes

Table 6: Busy Buzzard ‘keep alive’ beacon format

The earlier version of the TCP variant RC4 encrypts `length of beacon data` using the previously shared RC4 key.

RC4 key generation

The RC4 key used for C2 communications is randomly generated via the Windows API function `rand`, seeded by a call to the Windows API function `time`. The RC4 key used for C2 communication is:

- Randomly generated, between 8 and 16 characters in length, and only contains characters in the printable ASCII range (see the ‘Appendix (Pseudocode for RC4 key generation)’ section).
- Sent to the C2 server underneath RSA encryption, using the Windows Cryptography API functions and a hard-coded RSA public key.

Conclusion

Two variants of the Busy Buzzard malware have been identified, using different network protocols to support C2 communications (TCP and HTTPS). Overlap between the compilation timestamps associated with the two variants would appear to indicate that they have been used concurrently (assuming that the timestamps are legitimate and have not been deliberately modified).

It is possible that the choice of whether to use either the TCP or HTTPS variant of the malware is dependent upon the victim environment, for example protocols supported or known defences.

Two different versions of the TCP variant have been identified, one compiled in July 2016 and the other compiled in June 2019. The versions are broadly similar in functionality with the main differences being in the implementation of the C2 communications. This indicates that the Busy Buzzard malware has been actively supported during this period.

A significant amount of attention has been given to ensuring that the C2 communications (TCP and HTTPS) are difficult to detect and track. In particular the RC4 key, used to encrypt the C2 communications, is pre-shared in the initial beacon using RSA public key cryptography. This makes it very difficult to intercept and decrypt the C2 communications without firstly being able to identify the initial beacon, and then having access to the associated RSA private key.

The use of an in-memory shellcode loader to deploy the malware, multiple defence evasion techniques, multiple communication protocols, the difficulty in detecting and tracking the C2 communications, and its modular nature lead to the NCSC conclusion that Busy Buzzard is a medium sophistication piece of malware.

Recent open-source reporting by Trend Micro[3] has identified a newer version of the TCP variant, compiled in April 2021 and with added functionality. It has not been possible to analyse a sample of this newer version, but this does indicate that the actors are continuing to develop the Busy Buzzard malware.

Although the Busy Buzzard malware executes in-memory only, it is worth noting that the `d` command uses the `LoadLibraryW` Windows API function to load a selected plugin DLL into memory. This function loads a DLL file from disk, so any plugin DLLs would need to be present on the victim filesystem and could potentially be identified during forensic analysis.

The hard-coded `User-Agent` string found in samples of the HTTPS variant is very similar to one that has been uniquely attributed to the ChChes malware family by PwC[4]. The only difference between the strings is that the Busy Buzzard malware refers to `Trident/4.0` instead of `Trident/6.0`.

[3] https://blog.trendmicro.co.jp/archives/29842

[4] https://www.pwc.co.uk/cyber-security/pdf/cloud-hopper-annex-b-final.pdf

Detection

Indicators of compromise

Type	Description	Values
Domain name	C2 server domain	www.rare-coisns[.]com
IPv4 address	C2 server IP address	206.189.46[.]22
IPv4 address	C2 server IP address	88.198.101[.]58
URL	C2 URL	https[:]//www.rare-coisns[.]com/image/look/javascript/index.php
User Agent	Busy Buzzard User-Agent	Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; WOW64; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; .NET4.0E)
Mutex name	Busy Buzzard mutex	0A463B5B\x01[5]
Mutex name	Busy Buzzard mutex	51A50CE6\x01[5]

[5] Note that `\x01` refers to the single byte value `01`, which is a non-printable ASCII character

Rules and signatures

Description	Detects code bytes used by Busy Buzzard to switch between the different commands, as well as a test for an "ok" response from the C2 server
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_command_switching_and_ok_response
{
    meta:
        author = "NCSC"
        description = "Detects code bytes used by Busy Buzzard to switch
between the different commands, as well as a test for an 'ok' response
from the C2 server"
        date = "2022-03-25"
        hash1 = "8fd99d9066020003358aa3e23c9af3d4911ce979"
        hash2 = "266852db4ad2d293469515820fd5e7c228cd4b3e"
        hash3 = "d2b8f4fe6eedb8b87521772fc823da596f2403b7"
        hash4 = "6a673508d46c0bbff74ee24384c8bc841c11ea4d"
        hash5 = "e74affd6c766156e3fe803917f28da08fe7000ef"
        hash6 = "48152eeb1d74a84ba86b34f419cf1c7a105e41ff"

    strings:
        // switch between 'd', 'f', 'l', 's' commands (TCP variant)
        $tcp   = {3C 64 74 ?? 3C 66 74 ?? 3C 6C 74 ?? 3C 73 75 ?? 8D 57
FB 48 8D 4B 05 E8}
        // switch between 'd', 's' commands only (HTTPS variant)
        $http1 = {3C 64 74 ?? 3C 73}
        // test for 'ok' response from C2 server (HTTPS variant)
        $http2 = {80 3B 6F 75 ?? 80 7B 01 6B 74}

    condition:
        $tcp or (all of ($http*))
}
```

Description	Detects code bytes used by the TCP variant of Busy Buzzard to convert a CRC32 value to its hex string representation in reverse-nibble order
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_convert_crc32_to_mutex_name
{
    meta:
        author = "NCSC"
        description = "Detects code bytes used by the TCP variant of Busy
Buzzard to convert a CRC32 value to its hex string representation in
reverse-nibble order"
        date = "2022-03-25"
        hash1 = "8fd99d9066020003358aa3e23c9af3d4911ce979"
        hash2 = "266852db4ad2d293469515820fd5e7c228cd4b3e"
        hash3 = "d2b8f4fe6eedb8b87521772fc823da596f2403b7"

    strings:
        // loop that converts a 4-byte value to its hex string
        // representation (in reverse)
        $ = {0F B6 C2 24 0F 3C 09 76 02 04 07 04 30 48 FF C1 C1 EA 04 49
FF C8 88 41 FF 75 E5}

    condition:
        all of them
}
```

Description	Detects Busy Buzzard original DLL names embedded within the binary
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_original_dll_names
{
    meta:
        author = "NCSC"
        description = "Detects Busy Buzzard original DLL names embedded
within the binary"
        date = "2022-03-25"
        hash1 = "8fd99d9066020003358aa3e23c9af3d4911ce979"
        hash2 = "266852db4ad2d293469515820fd5e7c228cd4b3e"
        hash3 = "6a673508d46c0bbff74ee24384c8bc841c11ea4d"
        hash4 = "e74affd6c766156e3fe803917f28da08fe7000ef"
        hash5 = "48152eeb1d74a84ba86b34f419cf1c7a105e41ff"

    strings:
        $ = "httpsWin32.dll\x00"
        $ = "tcpcX64.dll\x00"

    condition:
        any of them
}
```

National Cyber Security Centre
a part of GCHQ

Description	Detects code bytes used by Busy Buzzard to generate the randomised RC4 key
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_random_rc4_key_generator
{
    meta:
        author = "NCSC"
        description = "Detects code bytes used by Busy Buzzard to
generate the randomised RC4 key"
        date = "2022-03-25"
        hash1 = "8fd99d9066020003358aa3e23c9af3d4911ce979"
        hash2 = "266852db4ad2d293469515820fd5e7c228cd4b3e"
        hash3 = "d2b8f4fe6eedb8b87521772fc823da596f2403b7"
        hash4 = "6a673508d46c0bbff74ee24384c8bc841c11ea4d"
        hash5 = "e74affd6c766156e3fe803917f28da08fe7000ef"
        hash6 = "48152eeb1d74a84ba86b34f419cf1c7a105e41ff"

    strings:
        // x86
        $ = {99 B9 3E 00 00 00 F7 F9 83 FA 0A 73 05 83 C2 30 EB 0D 83 FA
25 73 05 83 C2 37 EB 03 83 C2 3C 8B 03 88 14 06 46 3B F7}
        // x64
        $ = {44 8B D8 B8 43 08 21 84 41 F7 EB 41 03 D3 C1 FA 05 8B CA C1
E9 1F 03 D1 6B D2 3E 44 2B DA 41 83 FB 0A 7D 06 41 83 C3 30 EB 10 41 83
FB 25 7D 06 41 83 C3 37 EB 04 41 83 C3 3C 48 8B 4D 00 48 FF C3 48 FF CE
44 88 5C 0B FF 75 B0}

    condition:
        any of them
}
```

National Cyber Security Centre
a part of GCHQ

Description	Detects command response strings, built on the stack, used by the HTTPS variant of Busy Buzzard
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_https_cmd_response_stack_strings
{
    meta:
        author = "NCSC"
        description = "Detects command response strings, built on the
stack, used by the HTTPS variant of Busy Buzzard"
        date = "2022-03-25"
        hash1 = "6a673508d46c0bbff74ee24384c8bc841c11ea4d"
        hash2 = "e74affd6c766156e3fe803917f28da08fe7000ef"
        hash3 = "48152eeb1d74a84ba86b34f419cf1c7a105e41ff"

    strings:
        // "cmd="
        $ = {C7 (45 ?? | 85 ?? ?? ?? ??) 63 6D 64 3D}
        // "&type="
        $ = {C7 (45 ?? | 85 ?? ?? ?? ??) 26 74 79 70 66 C7 (45 ?? | 85 ??
?? ?? ??) 65 3D}
        // "&ret="
        $ = {C7 (45 ?? | 85 ?? ?? ?? ??) 26 72 65 74 66 C7 (45 ?? | 85 ??
?? ?? ??) 3D 00}

    condition:
        all of them
}
```

National Cyber Security Centre
a part of GCHQ

Description	Detects code bytes used by Busy Buzzard shellcode plugin helper functions
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_plugin_helper_functions
{
    meta:
        author = "NCSC"
        description = "Detects code bytes used by Busy Buzzard shellcode
plugin helper functions"
        date = "2022-03-25"
        hash1 = "f737067d41bc77dc7dd09ecb6eb710619bc2dfde"
        hash2 = "cf6339501de54590f8bbbc3cfb8051b95f6a1a42"

    strings:
        $ = {56 48 8B F4 48 83 E4 F0 48 83 EC 20 E8 F7 DF FF FF 48 8B E6
5E C3 48 C7 C0 30 00 00 00 65 48 8B 00 48 8B 40 60 48 8B 40 18 48 8B 40
10 48 8B 00 48 8B 00 48 8B 40 30 C3}

    condition:
        (all of them) and (filesize < 20KB)
}
```

National Cyber Security Centre
a part of GCHQ

Description	Detects code bytes used for resolving function addresses in the Busy Buzzard shellcode loader
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_shellcode_loader_x64
{
    meta:
        author = "NCSC"
        description = "Detects code bytes used for resolving function
addresses in the Busy Buzzard shellcode loader"
        date = "2022-03-25"
        hash1 = "8fd99d9066020003358aa3e23c9af3d4911ce979"
        hash2 = "266852db4ad2d293469515820fd5e7c228cd4b3e"

    strings:
        // resolving GetProcAddress
        $ = {3D 67 65 74 70}
        $ = {3D 64 64 72 65}
        // function name "IsBadReadPtr" built on stack
        $ = {C7 45 50 49 73 42 61}
        $ = {C7 45 54 64 52 65 61}
        $ = {C7 45 58 64 50 74 72}
        // function name "ImageRvaToVa" built on stack
        $ = {C7 45 20 49 6D 61 67}
        $ = {C7 45 24 65 52 76 61}
        $ = {C7 45 28 54 6F 56 61}
        // resolving kernel32 base address
        $ = {48 C7 C0 30 00 00 00 65 48 8B 00 48 8B 40 60 48 8B 40 18 48
8B 40 10 48 8B 00 48 8B 00 48 8B 40 30 C3}

    condition:
        all of them
}
```

National Cyber Security Centre
a part of GCHQ

Description	Detects hard-coded RSA public keys used by both the TCP and HTTPS variants of Busy Buzzard
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_rsa_public_keys
{
    meta:
        author = "NCSC"
        description = "Detects hard-coded RSA public keys used by both
the TCP and HTTPS variants of Busy Buzzard"
        date = "2022-03-25"
        hash1 = "8fd99d9066020003358aa3e23c9af3d4911ce979"
        hash2 = "266852db4ad2d293469515820fd5e7c228cd4b3e"
        hash3 = "d2b8f4fe6eedb8b87521772fc823da596f2403b7"
        hash4 = "6a673508d46c0bbff74ee24384c8bc841c11ea4d"
        hash5 = "e74affd6c766156e3fe803917f28da08fe7000ef"
        hash6 = "48152eeb1d74a84ba86b34f419cf1c7a105e41ff"

    strings:
        // Hard-coded RSA public keys used by Busy Buzzard samples
        $rsa1 =
"MIGJAoGBANYrLIbgWsKfJOnWnk/YUJmgFxI5PMK31Rnj7AOR37VM66baxx/6+03/IbT4oe/P
y9YBYUIILhZTcCGOjv/KW+nZoQ4gdkDXexP7kD2YuwEKQZB1WSG3MTPNLCWANvGMrX+Cb/9H3
qJ/mmIYrygcLmzj4E2rSxJfdZ3YaYI1IX/vAgMBAAE=\x00"
        $rsa2 =
"MIGJAoGBAK7mh7RSMtisKLn+Jfkq9AUlOHqUe4zjTLVC89k+sPux5ZMr9ndtjzdx8bCcSfCQ
temKrR2LY4lRr5cZs3jgwaBbHS2SdCezUuNUdrEEsfWX8BlK13G8djFmmYZqKeQFnUrKZn+uA
0A4nIGPRFKB2fKfBjh4Y5qN2IoyV9Y0e8HHAgMBAAE=\x00"
        $rsa3 =
"MIGJAoGBALkXcETCNbKRUMlz0Bkl8Mr/Jm1A4VKxdLBlDXCtD/9fCrfSDl2z/JhykFJik787
pT05QuKIsLWZLv2/lqMlDxnKEPEQRDBdm900If27xShcK/qRoSOO8edUD44PphF5cMfK16VMo
N9e3DEVeP4zduCanP4vbFpH3vwaTI1Or1QRAgMBAAE=\x00"

    condition:
        (1 of ($rsa*))
}
```

National Cyber Security Centre
a part of GCHQ

Description	Detects additional random bytes used by TCP variants of Busy Buzzard to generate a unique mutex name
Precision	No false positives have been identified during VT retrohunt queries
Rule type	YARA

```
rule BusyBuzzard_mutex_random_bytes
{
    meta:
        author = "NCSC"
        description = "Detects additional random bytes used by TCP
variants of Busy Buzzard to generate a unique mutex name"
        date = "2022-03-25"
        hash1 = "8fd99d9066020003358aa3e23c9af3d4911ce979"
        hash2 = "266852db4ad2d293469515820fd5e7c228cd4b3e"
        hash3 = "d2b8f4fe6eedb8b87521772fc823da596f2403b7"

    strings:
        // Additional random bytes used by TCP variants of Busy Buzzard
to generate a unique mutex name
        $rand1 = {61 87 67 6C 6A DB 4A C6 58 D1 61}
        $rand2 = {CF 4F 26 6A 57 77 5B B7 AB D4 29}

    condition:
        (1 of ($rand*))
}
```

Appendix

Pseudocode for RC4 key generation

```
Seed random number generator with output of time()
Generate a random key length as rand() & 0xf
If key_length < 8:
    key_length += 8
Loop over key_length:
    Generate next key byte as rand % 0x3e
    If next key byte < 0xa:
        next key byte += 0x30
    Else:
        If next key byte < 0x25:
            next key byte += 0x37
        Else:
            next key byte += 0x3c
```

YARA rule used to identify kernel32.dll

```
import "pe"

rule identify_kernel32_version
{
    condition:
        (uint16(0) == 0x5a4d) and pe.is_dll() and pe.is_64bit() and
        (pe.export_details[pe.exports_index("GetProcAddress")].offset ==
         pe.rva_to_offset(0x19d70)) and
        (pe.export_details[pe.exports_index("LoadLibraryA")].offset ==
         pe.rva_to_offset(0x1f560)) and
        (pe.export_details[pe.exports_index("VirtualAlloc")].offset ==
         pe.rva_to_offset(0x1b220)) and
        (pe.export_details[pe.exports_index("VirtualFree")].offset ==
         pe.rva_to_offset(0x1b970))
}
```

National Cyber Security Centre
a part of GCHQ

Disclaimer

This report draws on information derived from NCSC and industry sources. Any NCSC findings and recommendations made have not been provided with the intention of avoiding all risks and following the recommendations will not remove all such risk. Ownership of information risks remains with the relevant system owner at all times.

This information is exempt under the Freedom of Information Act 2000 (FOIA) and may be exempt under other UK information legislation.

Refer any FOIA queries to ncscinfoleg@ncsc.gov.uk.