Betabot

Configuration Data Extraction

Decoding Betabots created with the cracked builder

Tad Heppner
Contents

Executive summary..................................................................................................................3
Introduction...............................................................................................................................3
Betabot and Neurevt ...............................................................................................................4
The pirating and cracking of Betabot.......................................................................................5
Betabot command and control server. ....................................................................................8
Primary features ....................................................................................................................10
  Tasks and scheduled jobs......................................................................................................10
  The ability to detect and disable other malware and anti-virus programs........................10
  Proactive defense.................................................................................................................11
  DDoS.....................................................................................................................................12
  Form Grabbers......................................................................................................................12
  DNS Blocker + Redirector.....................................................................................................13
  Autorun spread.....................................................................................................................14
Betabot top level packer ..........................................................................................................15
  Unpacking the outer layer...................................................................................................15
Decrypting the configuration data ........................................................................................17
CnC communications.............................................................................................................21
  Decoding bot HTTP request to the CnC server ................................................................22
  Decoding CnC servers HTTP response to the bot .............................................................24
Summary................................................................................................................................26
Acknowledgements...............................................................................................................26
References..............................................................................................................................26
Executive summary

This paper explores the inner workings of Betabot, including capabilities of the associated botnet server components and technical detail on how to extract and decrypt configuration data.

It also includes information on how to use the cryptographic keys encoded in the malware’s configuration data to decode communications between the bot and the command-and-control server.

Most of the methods described here focus on Betabot 1.7 malware produced using a cracked Betabot builder, which has grown popular among malware pushers who would rather copy Betabot’s functionality than build their own from scratch. We also explore ways in which Betabot’s creators are trying to keep others from stealing their work.

Finally, we explore alternate methods to decode older Betabot versions as well as a related variant called Neurevt.

Introduction

Betabot has been around for a while. New variations have been discovered over time as the authors continually revise and update the malware. The most prevalent revision found in the wild at the time of finalizing this paper was Betabot revision 1.7.

Over the last few years, the number of Betabot infections found in the wild has continued to fluctuate as the larger campaigns introduce new methods of packing and distributing the bot client in an attempt to circumvent antivirus.

Some samples have also been discovered that attempt to connect to control servers without public domain or IP addresses, suggesting they may reside entirely within a private network.

There are rumors of a 1.8 version and links provided on several hacking sites. Investigations into this have exposed the majority of these as being the 1.7 versions of the botnet or versions that have been slightly modified after the fact with no significant changes to the existing Betabot components. The most significant change in 1.7 from its 1.6 predecessor is the inclusion of an additional layer of HC128
encryption on the individual command and control [CnC] server entries within the malware’s configuration data structure.

Betabot’s CnC server interface is fairly user friendly. As such, it can be used by cybercriminals who either lack the technical knowledge or do not wish to spend the significant effort authoring a botnet framework themselves.

The Betabot malware package is advertised on black markets for around $120 USD and is typically purchased by contacting the author directly to arrange payment.

More recently, a cracked version of the Betabot builder has been discovered that allows other cybercriminals to utilize Betabot without purchasing it from the author(s). As Betabot’s intended use is nefarious in nature, the existence of cracked versions of the builder indicates cybercriminals are not only targeting members of the unsuspecting public but are also engaged in activities related to hacking other malware to leverage the work of other malware authors for free.

Although this is not unprecedented, the increased availability due to the utilization of a software crack often results in an increase in the malware family’s use by new parties.

**Betabot and Neurevt**

Neurevt is very similar to Betabot in many respects -- so much so that it’s often considered a common alias for the Betabot malware family.

The Neurevt variant of Betabot may have been a fork off of the Betabot source code. Neurevt, like Betabot, also embeds two random seeds used as cryptographic keys comprised of 16 alphanumeric characters to facilitate obscuring the communications of the bot with its CnC server over HTTP. The custom variables and data formatting for sending this information within the HTTP post itself is different between the variants.

Both Betabot and Neurevt go to extra effort to try and deter analysis of the bot by hooking KiFastSystemCall, allowing it to intercept other function calls exported in NTDLL. This allows the malware to utilize some creative anti-debugging methods on a new instance of explorer.exe that it then spawns and injects with code to communicate with the CnC server. Attempts to dump the memory of or attach a debugger to this new instance of explorer.exe is complicated as the malware can then subvert calls used by the debugger or system tools attempting to read the process memory.

Betabot seems to go to extra effort in other aspects, attempting to mask the bot’s configuration data utilizing some obfuscation and additional XOR masks on both the configuration data and RC4 keys themselves. Betabot 1.7 also applies an additional encryption layer to the individual configuration sections. These methods have not been observed within the Neurevt family.

Another way to distinguish between the two variants is to look for strings within the injected process memory once the malware has unpacked and infected the system. Betabot can be identified by the presence of the Unicode string “Betabot (c) 2012-2014, coded by Userbased” and Neurevt by a single byte character encoding of the string “Neurevt” within the injected process.
The pirating and cracking of Betabot

Software piracy is not only a problem for public software companies. It would appear that it is also a problem for malware authors who are attempting to sell malware to other cybercriminals on black markets.

Legitimate software companies will often take additional steps to discourage people from attempting to violate licensing terms and ensure that only customers who have paid for the product are able to utilize it. Sometimes this is done through the use of product registration keys or physical USB hardware keys. People who wish to utilize the software without purchasing it are known as software pirates.

Pirates will often distribute patched versions of a program’s code in which the anti-piracy measures have been circumvented. The patched binaries are known as software cracks. If the pirate is able to reverse engineer the algorithm used to generate license key codes for a given program, a utility known as a key generator (keygen) may be created and distributed that allows non-licensed users to generate valid product keys for a given application.

As an additional cautionary note: Software cracks and keygens are often distributed via peer-to-peer file sharing networks. These peer-to-peer networks are a prime breeding ground for other malware as the decentralized servers are managed by individuals and may not have the same level of compliance or accountability with regard to their own computer and network security infrastructure. As such, other malware families, especially parasitic malware families such as Sality and Virut, will often spread by infecting files in a peer-2-peer user’s hosting directory, causing the malware to spread to the next user and infect their shared files, and so on. Many software cracks and keygens also contain other embedded malware due to the use of this method of distribution. In addition, many pirates also intentionally embed malware payloads into the pirated copies that are published and distributed via popular file sharing services as my colleague Paul Ducklin points out in this article:

“Will a visit to The Pirate Bay end in malware?”
https://nakedsecurity.sophos.com/2016/05/06/will-a-visit-to-the-pirate-bay-end-in-malware/

The creators of the Betabot malware have taken some additional steps to add some anti-piracy measures to this malware toolkit in order to ensure the authors receive payment for the use of the malware by other cybercriminals.

One of these measures is the complexity involved in the method of encoding the configuration data inside the bot payload. The configuration data, among other things, includes the URLs of the CnC server(s) that the bot will connect to as well as encryption keys that will be used to encrypt and decrypt the information sent to the individual CnC server(s). This configuration data itself is encrypted and stored within the bot itself when the payload is generated. The complexity of this packing method not only makes it difficult for anti-virus and other security tools to unpack the information statically, but it also deters other criminals from attempting to encode their own configuration data containing altered information.
In this way, the authors attempt to maintain control of the process of generating new bot payloads for a given CnC server. Although the method is complex, it is still technically possible to decode this information due to the fact that the decryption key must somehow be made available to the bot itself when it infects a new target.

Additionally, Betabot contains another interesting feature called “proactive defense”. The purpose of this feature is to try and prevent other competing bots or similar tools such as remote access trojans from installing and potentially hijacking the botnet.

Cracked Betabot builder screen capture

The crack itself consists of a console-based builder application with the compiled Betabot template code stored as a bytes array within the data section of the builder application itself.

When run, the crack allows the user to specify custom configuration information that is then encrypted and inserted into the included template code at the appropriate position. The whole PE file is then repacked in attempt to further obfuscate the generated bot in attempt to avoid detection by anti-virus software.

The custom configuration information provided to the builder instructs the bot to connect to a specified control server. Up to 16 individual CnC servers can be stored within a single Betabot configuration data structure. Typical Betabots found in the wild only specify one or two servers and do not seem to utilize all 16 of the available entries.

Registry startup name (max. 42):
> testreg

Install folder name (max. 42):
> testfolder
Number of host entries (1-16):
>1

==================================================================
Hostname (eg. google.com max. 42):
>localhost

Gate (eg. /path/order.php max. 42):
>/path/order.php

Port :
>80

Use SSL Connection (0/1):
>0

==================================================================

The cracked builder also generates some pseudo random keys to be used for the communications with the CnC server. It encrypts these keys into the bot’s configuration data along with the configuration information that the user has entered and generates a packed payload executable that can be distributed or deployed to the intended victims.

[+] Successfully built!
[+] Web Key 1: bf2ea8909eb787d9
[+] Web Key 2: 794be0e4c9fa1f74

At the completion of the build process, the communications keys that were generated and encoded into the configuration data are displayed on the screen so that the user can then configure their own CnC server to utilize the same keys as the bot payload.

I was also able to locate what appears to be some of the original sources used in the development of the cracked builder. Although these source files are incomplete, I believe them to be authentic as the methods used in the source describe what I have found to be the correct structures and methods for packing the configuration data for use with Betabot 1.7.

// Analysis by duyan13@HH
// Special Credits for this analysis : IDA and Olly, Fortinet, testacc@HH (Sample and mental support), Let it go (Idina Menzel)

The HC128 algorithm is included in the source code in the form of inline x86 assembly code intended for use with the Microsoft Visual Studio Compiler. The comments in the code at this point seem to indicate that the authors of the crack were not able to identify this encryption algorithm.

// This ! lol Couldn't reverse engineering this algo so I simply extracted it xD.
Betabot command and control server.

The Betabot CnC server side component is primarily written in PHP and encoded with ionCube (https://en.wikipedia.org/wiki/IonCube). It also requires the support of a MySQL database. This component is intended to reside on a webserver and facilitate communications with the individual systems that have been infected with the Betabot malware. It also features a web-based user interface that allows the bot herder to configure the individual bots and deploy tasks for them to execute on the infected systems.

Within a controlled and isolated network environment I was able to conduct some experiments allowing the cracked bots to connect to a CnC server and explore some of this functionality.

During the initial configuration and setup of the server, the communication keys that were generated by the builder are required as well as DB credentials to deploy and setup the initial database schema.

After a server is successfully deployed, a dashboard can be accessed. This dashboard provides information about the number of connected bots and the geographies in which they reside.
Primary features

Tasks and scheduled jobs

The dashboard also provides a user interface to create, schedule and assign tasks for the individual infected systems to perform. The bot will perform an HTTP request to the command-and-control server at configurable intervals and the server will send the task information to the bot in the form of POST data in the HTTP response.

These task types range from instructing the infected computers to:

- Participate in a DDoS attack against a given target
- Download and execute a file from a given URL
- Detect and grab information out of HTML forms displayed in a web browser when the browser is pointed at a specific web page. (can be used to steal logon information)
- Launch a web browser and direct it to a given URL
- Set a new web browser home page
- Clear the web browser’s cookies
- Create a SOCKS4 proxy (Allows others to redirect browser traffic through the victims IP address)
- Run a given system command (optionally with admin credentials if available)

The Betabot malware payload also contains a social engineering tactic in 12 different languages that attempts to trick the user into elevating the bot’s privileges. The author(s) claim that the method has proven 80% effective.

The ability to detect and disable other malware and anti-virus programs

The author(s) also claim Betabot is able to identify and disable about 30 different anti-virus or security programs when it infects a new host. The Betabot dashboard provides a user interface that shows the operator what security or anti-virus software was found on the host. I did some limited testing with a few commercial antivirus products. In most cases, Betabot was unable to identify corporate or
enterprise antivirus software installations. It was able to correctly identify at least one consumer anti-virus program I tested with but it did not appear to be able to effectively disable it.

Proactive defense

In addition to legitimate security and anti-virus software, Betabot also attempts to identify and remove other major competing malware botnets that may be found on the system.
DDoS

One of the task types available in the dashboard is the ability to configure connected bots to participate in a distributed denial of service attack against a given target. The dashboard provides configuration options to utilize one of four methods:

- UDP
- Rapid Connect/Disconnect
- HTTP GET
- Slowloris

It also provides the option for the connected bots to use local information to randomize the HTTP headers during the attack.

Form Grabbers

The dashboard also provides the bot controller a method to pull information entered into HTML forms on the victim’s computer. This allows the attacker to steal logon information when the victim enters credentials or other sensitive information into a web page.
DNS Blocker + Redirector

The dashboard is also able to configure connected bots to block or redirect DNS requests without modification to the HOSTS file. In this way the bot controller can redirect certain traffic to a sinkhole allowing them or a third party to potentially harvest information with some additional social engineering and/or a man-in-the-middle type attack.
Autorun spread

The dashboard optionally allows the controller to enable another bot feature that instructs it to attempt to spread itself as a worm. When this option is enabled, the infected system will attempt to copy an instance of the bot payload to removable media when inserted into an infected computer. It will additionally place a loader on the media that will attempt to run the malware when the device is inserted into a new computer via the addition of an autorun.inf, or utilizing a LNK-File swap technique.
Betabot top-level packer

Now that we understand some of the capabilities of Betabot and how the CnC server functions, we will take a closer look at the Betabot payload itself. This is the actual malware code that is intended to be distributed and executed on the target systems. In this section we will walk through the unpacking of a Betabot 1.7 sample in an attempt to extract the configuration data encoded into the malware that it will use to connect to its CnC servers to receive further instruction. (This information does not apply to the Neurevt variants, which are typically packed using another method.)

Unpacking the outer layer

First, we must decode the outermost PE packer layer. The purpose of this layer is an attempt to make it more difficult for antivirus software to identify Betabot. As most antivirus software can identify and unpack common commercial PE file packers, the most effective implementations often utilize a somewhat customized packing method. I was able to unpack the outmost layer first by first skipping over the second jump instruction from the entry point using a debugger as shown below.

```
0114120A >$/ S5    PUSH EBP
0114120B  .B8EC    MOV EBP,ESP
0114120D  .51      PUSH ECX
0114120E  .64:A1 30000000 MOV EAX,DWORD PTR FS:[30]
01141214  .53      PUSH EBX
01141215  .56      PUSH ESI
01141216  .85C0    TEST EAX,EAX
01141218  .74 06   JE SHORT .01141220
0114121A  .8078 02 01   CMP BYTE PTR DS:[EAX+2],1
0114121B  .85C0    TEST EAX,EAX
0114121D  .8365 FC 00   AND DWORD PTR SS:[EBP+4]
0114121F  .74 30   JE SHORT .01141250
01141220  >/8365 FC 00    AND DWORD PTR SS:[EBP-4],0
01141224  .E8 FBFDFEFF CALL Sample_.01141024
01141229  .85C0    TEST EAX,EAX
0114122B  .74 23   JE SHORT .01141250
0114122D  .8D4D FC LEA ECX,DWORD PTR SS:[EBP-4]
01141230  .51      PUSH ECX
01141233  .8BF0    MOV ESI,EAX
01141236  .E8 FBFDFEFF CALL Sample_.01141133
0114123B  .8BF0    MOV ESI,EAX
0114123E  .59      POP ECX
0114123F  .85C0    TEST EAX,EAX
01141241  .74 11   JE SHORT .01141250
01141243  .E8 54FEFFFF CALL Sample_.01141098
01141248  .88 99040000 CALL Sample_.011416E9 ;
0114124C  .66:00 000201401 CALL DWORD PTR DS:[<&KERNEL32.ExitProces>]; ExitProcess
```

Set breakpoints after both of the two ‘kernel32.VirtualAllocEx’ calls allowed it to execute until the debugger hit the first of these breakpoints. When it breaks, it will start unpacking itself to the newly allocated memory region.

Step though the unpacking loop a few times and locate the jump where the loop repeats. Set a new breakpoint so that it will break immediately after this loop exits.
When the loop completes, the memory region will now contain the unpacked PE file. Dump this region to disk and we now have an unpacked Betabot sample which we will be able to further analyze.
Decryption the configuration data

There are several other custom layers of obfuscation and encoding that are applied to the bot’s configuration data in an attempt to mask and deter reverse engineering. Some of these methods seem to vary slightly with each significant update of Betabot. The structure of the config data and the use of RC4 in the primary layer seem to have remained fairly constant from version to version. However, the XOR key values, initialization vectors, and other minor variations seem to be introduced or modified with each update.

1. Extract the RC4 key

Inside the unpacked PE the config data is encrypted using RC4. The encryption key can be found by looking for some marker bytes highlighted in yellow in the unpacked PE. The bytes in green contain the decryption key but first it must be decoded before we can utilize it.

```
.0135Aa9A:  DC 3A 16 F7 B1 91 4C 66 00 00 00 C1 11 00 00
.0135aaaA:  CC 18 8E A6 85 6C DF E4 AE 8A B2 F0 CE E1 EF DB
.0135Aaba:  DC 11 25 20 A5 CF 38 39 4D AF 6B 99 FF 06 39 4D
```

The method of decoding this key is slightly different between different versions of the bot. For example:

- For Betabot 1.6 every even (zero indexed) byte in the green block (starting with the first byte) must be XORed with 0x2e.
- For Betabot 1.7 every even (zero indexed) byte in the green block (starting with the first byte) must be XORed with 0x1a plus the value of its index position in the array (starting at zero).
- For Neurevt variants, this marker will be in code that has been injected into the process memory of explorer.exe. The RC4 key is comprised of every even byte and will not need to be XORed.

The result should be a key that is 16 bytes in length as we are using every other byte from the block of 20.

Using the above example after applying the XOR for Betabot 1.6 the resulting key would be:
```
e2 a0 ab f1 80 9c e0 c1 f2 0b 8b 16 63 45 d1 17
```

If the above example was from Betabot 1.7 the resulting key would be:
```
d6 92 9b ff 8c 96 e8 c7 f6 09 8b 08 7f 5f c9 01
```

If the above example was from Neurevt the resulting key would be:
```
cc 8e 85 df ae b2 ce ef dc 25 a5 38 4d 6b ff 39
```
2. Locate and extract the configuration data

The next step would be to locate the encoded configuration data. This can be accomplished by locating the bytes in yellow in the layer 1 unpacked PE. The encoded configuration data will be the next 0xd56 bytes following this marker. The data is truncated in the following example.

```
.01352AB7:  54 2E BB C9 FF F7 77 39 1A 8C FE AC 96 2D 54
.01352AC7:  B2 E5 9C 90 A2 C6 87 A9 14 67 A5 40 AB D3 33 65
.01352AD7:  AB 19 B1 2D C0 0A 5F D1 2D 8C F7 18 F8 CD C1 07
.01352AE7:  E2 40 7D A2 A5 71 9B 5F 91 3D 30 83 9B E1 86 93
.01352AF7:  ...
```

Locating the encrypted configuration data in Neurevt is a bit more difficult.

The marker to look for is 1 byte different (54 2E BB C9 FF) and the size of the data is 0xd46. It will also only exist in this format within the injected explorer.exe process which can be tricky to obtain due to the anti-debugging tricks discussed earlier. It should be possible to extract this parent process as well in this way on some of the earlier versions of Neurevt. Some extra work may be required with the current campaigns, however.

3. Custom Obfuscation layer

Next, apply the following routine to the configuration data using the RC4 Key. This is only needed for Betabot variants of the malware family. To decrypt Neurevt, skip directly to the RC4 decryption in step 4.

Here is the routine as a python function:

```python
def customencode(config, key):
    OutData = ""
    i=0
    for character in config:
        keypos = i & 3
        OutData += chr(ord(character) ^ ord(key[keypos]))
        i = i + 1
    return OutData
```

4. Decrypt the configuration data block

The next step will be to take the output from the above function and apply the RC4 algorithm using the same key we extracted earlier.

The methods for locating the encryption key should work the same for earlier Betabot versions as well. If we are working with Betabot 1.6 at this point in the process, we would be mostly complete with the extraction of the configuration data. The individual server URLs would still need to be run though a mild deobfuscation routine that we will discuss later. (If working with Neurevt or Betabot version 1.6 and earlier, skip to section 6.)
5. **Decrypt the individual configuration entries  
(Betabot version 1.7 only)**

An additional difference with Betabot 1.7 is that each of the 1-16 individual server structures are each individually encrypted with HC128. Thanks to my colleague Dorka Palotay for her assistance with the identification of this algorithm. ([https://en.wikipedia.org/wiki/HC-256](https://en.wikipedia.org/wiki/HC-256))

The good news is that the HC128 encryption key is now exposed in the block of data we just decrypted using the RC4 algorithm.

At this point we will be able to verify the success of the previous RC4 decryption of the configuration data, as some of the common elements in the containing structure should now be exposed. For example, 6 bytes from the start of the decrypted blob will be a string of random ASCII printable values (key material we will later use). The registry keys name and folder names to be utilized should now be exposed as Unicode strings as offsets 0x46 and 0xc6.

Starting at offset 0x156 in the RC4 decoded data, there are 16 consecutive sub structures that are each 0xc0 in length. We will need to decode these in order to extract the individual CnC server configuration information.

The HC128 key to use is formatted as two null bytes followed by the next 12 bytes starting at offset 0x8 from the start of the configuration data, followed by another 18 null bytes.

For example if the start of the RC4 decrypted configuration data was this:

```
00000000:  56 0D AA 0F 92 C1 67 4A 6D 79 64 77 6F 57 36 31
00000010:  4E 66 4N 39 31 32 42 45 4K 2Y 6F 47 74 00 00
```

The HC128 Key would be:

```
00000000:  00 00 6D 79 64 77 6F 57 36 31 4E 66 4N 39 31 32
00000010:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Each of the 16 sections will need to be decrypted independently using the above key. Depending on the implementation of HC128 that you are using, the initialization vectors may salt a seed value and generate a key to pass into the algorithm for you. If so, bypass this code as the above should be the completed key value needed for the decryption routine.

After all 16 blocks of 0xc0 bytes are decrypted, the CnC server information should be visible. We still need to deobfuscate the CnC servers as discussed later.

6. **Extract the folder and registry key names**

At this point we have several new pieces of information we can extract from the configuration data structure.

At offset 0x46 we have the string value of the registry key name that will be utilized for the creation of a run key load the bot after a system reboot.

At offset 0xC6 we have the string value for the folder name that will be created when the bot is initially run. A copy of the bot will be placed in this folder and referenced in the run key mentioned above.
Both of these values should be completely decrypted at this point.

7. **Deobfuscate the CnC URLs**

Within the configuration data structure there are 16 individual configuration entry structures. Each of these 16 configuration entry structures hold specific information for a single CnC server. Another difference between Betabot 1.6 and 1.7 is the offset to the beginning of these individual configuration data structures.

For Betabot 1.6, the individual config entries start at offset 0x146 from the beginning of the configuration data.

For Betabot 1.7, the individual config entries start at offset 0x156 from the beginning of the configuration data.

The 16 individual configuration entry structures are 0xC0 in length for both 1.6 and 1.7 versions.

The offsets for the information contained within an individual config entry are as follows and are the same for both versions of Betabot.

- 0x12: Number of Retries
- 0x14: Port
- 0x1E: Use SSL
- 0x26: Host name
- 0x66: URL
- 0xAE: CnC Crypto Key 1
- 0xB7: CnC Crypto Key 2

Most of this above data can be extracted as is with the exception of the string value of the host name. If the host name value is greater than 8 characters but less than 42 characters in length, the string may be mildly obfuscated and need to be converted using a routine similar to the following python version of the deobfuscation method:

```python
def deobfuscate_host(host):
    slen = len(host)
    array = [ord(x) for x in host]
    if slen < 8 or slen > 0x40:
        return host
    val_0 = array[0]
    loc1 = (val_0 + 2 * slen) % (slen - 2) + 1
    loc2 = (val_0 + 8 * (slen + 1)) % (slen - 3) + 2
    xor1 = 655 * val_0 % 3 + 24
    xor2 = 1424 * val_0 % 6 + 23
    if xor1 != array[loc1]:
        array[loc1] ^= xor1
    if xor2 != array[loc2]:
        array[loc2] ^= xor2
    return ''.join([chr(x) for x in array])
```
CnC communications

Once Betabot infects a system it will call home to the CnC server in the form an HTTP request typically for a document named order.php (Although this can be renamed on different CnC server installations).

Although the communications methods may be similar with older variants of the botnet, for this example we will look at the 1.7 variants only.

When the bot attempts to call home to the CnC server(s) specified in the CnC data we extracted earlier, it includes several pieces of encoded information in the HTTP header in the form of POST data. This data contains ASCII representations of hexadecimal values which the bot is sending back to the server.

A typical bot check-in HTTP request may look something like this like this:

```
POST /panel/order.php?pid=208 HTTP/1.1
Content-Type: application/x-www-form-urlencoded

User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; 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)
Host: 192.168.54.111
Content-Length: 1032
Cache-Control: no-cache

fkvalqb=34F54ED21FA175214D28ECE4D4817E180B038A141B202DF88B457F68554106590A158F2040E388&dqspsb=439587855&hobivcawj=068def49ff46dbef6b559b3191051a2b16fa606f618f4df473aa6f2f75338e3de01b6a71db04457448a1c702628f2eeodd883a447717ce4ad4d38b926a3de13d38167464e012ee037692e1723a9376182a4dd5bbf804fe5dd4ae205f5e5b6f577d87a7c82418980d8a2634a30755ba589b2e431c42f743c12253725888b24f76f6127f0f6127fe0adee8e621532a28ff838e9d77cfe2695e5ee3e3d06&jshqf1=61F04BC27EF021C2250F01ECC24F003C243F01CC266F010C256F010C27EF038C24CF002C256F010C24EF01DC264F01ECC24EF014C247F002C247F010C24EF014C24EF018C257F00BC247F010C247F010C244F010C247F010C247F010C24EF010C247F010C24EF010C244F010C247F010C247F010C247F010C24EF010C247F010C247F010C24EF010C244F010C247F010C247F010C247F010C24EF010C247F010C247F010C24EF010C244F010C247F010C247F010C247F010C24EF010C247F010C

*jshqf3 has been removed from this example. Normally this will only contain user and host information in hexadecimal representation.
```
Decoding bot HTTP request to the CnC server

1. Construct the RC4 Key

The first step in decoding this data is the construction of the RC4 encryption key that we’ll use to decrypt the header information. To do this we’ll need the value of “CnC Crypto Key 1” that we extracted from the bot’s configuration data at offset 0xAE within the individual configuration data section for the server, which the bot is attempting to contact.

This new key can be constructed by combining the value of “CnC Crypto Key 1” with the value in the first parameter in the HTTP post, then xor-ing each byte with the static value 0xCB.

In the above example we would append the bytes in the HTTP post parameter named “fkvalqb” and append them to the end of the value of the “CnC Crypto Key 1” we extracted from the configuration data earlier in the paper.

The resulting byte array after the XOR operation would then be:

```
81BD1AF7D8825AFF3E8506A31DC99DF1945051F4AB5D0C0CR41DFD0E6E633408EB4A39EBACD92C1DE44EB3FC5F3
```

2. Decrypt the header data

This is the RC4 decryption key that we will then use to decode the information in the bot communication header. The encrypted header data in the above example that we will decode with this key is the third http post parameter, which in this case is named “hobivcpwj”. After decryption, we will get a structure as follows:

```
00000000: 00 00 00 00-00 00 00 00-A4 00 E5 C1-16 05 00 00 ................
00000010: 05 00 00 00-02 00 00 00-01 07 00 01-20 05 00 00 ................
00000020: 36 00 00 00-05 19 0F 58-6E 6B 57 00-C4 FF 00 00 ................
00000030: 55 53 00 00-00 00 00 00-00 00 00-00 00 00 00 ................
00000040: 89 7C 40 30-4B 94 06 C1-00 00 00-00 00 00 00 ................
00000050: 00 00 00 00-00 00 00 00-00 00 00 00 00 00 00 00 ................
00000060: 00 00 00 00-00 00 00 00-00 00 00 00 00 00 00 00 ................
00000070: 00 00 00 00-00 00 00 00-00 00 00 00 00 00 00 00 ................
00000080: 00 00 00 00-00 00 00 00-00 00 00 00 00 00 00 00 ................
00000090: 00 00 00 00-00 00 00 00-00 00 00 00 00 00 00 00 ................
000000A0: 00 00 00 00-00 00 00 00-00 00 00 00 00 00 00 00 ................
```

Using the deobfuscated order.php, I was able to work out the structure of this decoded header. It contains the following information:
This HTTP request can send different types of information back to the CnC server. A bitmasked value in the header at offset 0x14 (request_type) tells the CnC server what types of information are contained in a single HTTP request. There are up to 5 different request types that can be combined into one CnC server HTTP request.

'BOT_REQUEST_TYPE_CHECK_IN' = 1
'BOT_REQUEST_TYPE_SYSTEM_BOOT_CHECKIN' = 2
'BOT_REQUEST_TYPE_UPDATE_STATISTICS' = 4
'BOT_REQUEST_TYPE_FORMGRAB_DATA' = 8
'BOT_REQUEST_TYPE_STEALER_DATA' = 16

As we can see from the example header we decoded here, this mask value says that the HTTP request we captured only contains 'BOT_REQUEST_TYPE_SYSTEM_BOOT_CHECKIN' type data.

3. Decode the system information data

The additional data contained in the HTTP post for the 'BOT_REQUEST_TYPE_SYSTEM_BOOT_CHECKIN' request should contain 5 additional parameters using the same base name with the numbers 1-5 appended. In the above example these are named jshqf1, jshqf2, jshqf3, jshqf4, and jshqf5.

I was able to find where the numbered HTTP post parameter names were constructed in the CnC servers PHP code I deobfuscated. The variable names that the author(s) used provide some insight as to what data is contained in these variables.

```php
$g_szPost_HeaderStringInstallName = $g_szPost_HeaderStringName . '1';
$g_szPost_HeaderStringBrowserName = $g_szPost_HeaderStringName . '2';
$g_szPost_HeaderStringUserName = $g_szPost_HeaderStringName . '3';
$g_szPost_HeaderStringCpuName = $g_szPost_HeaderStringName . '4';
$g_szPost_HeaderStringVideoName = $g_szPost_HeaderStringName . '5';
```

<table>
<thead>
<tr>
<th>offset</th>
<th>size</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x04</td>
<td>crypt_reserved1</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
<td>crypt_reserved2</td>
</tr>
<tr>
<td>0x08</td>
<td>0x02</td>
<td>header_size</td>
</tr>
<tr>
<td>0x0A</td>
<td>0x02</td>
<td>magic [0xC1E5]</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x04</td>
<td>header_crc32</td>
</tr>
<tr>
<td>0x10</td>
<td>0x04</td>
<td>request_type</td>
</tr>
<tr>
<td>0x14</td>
<td>0x04</td>
<td>bot_os</td>
</tr>
<tr>
<td>0x18</td>
<td>0x04</td>
<td>bot</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x04</td>
<td>bot_version</td>
</tr>
<tr>
<td>0x20</td>
<td>0x04</td>
<td>bot_attributes</td>
</tr>
<tr>
<td>0x24</td>
<td>0x04</td>
<td>bot_time_current</td>
</tr>
<tr>
<td>0x28</td>
<td>0x04</td>
<td>bot_time_uptime</td>
</tr>
<tr>
<td>0x2C</td>
<td>0x04</td>
<td>bot_time_gmt</td>
</tr>
<tr>
<td>0x30</td>
<td>0x04</td>
<td>locale</td>
</tr>
<tr>
<td>0x34</td>
<td>0x02</td>
<td>bots_killed</td>
</tr>
<tr>
<td>0x36</td>
<td>0x02</td>
<td>socks_port</td>
</tr>
<tr>
<td>0x38</td>
<td>0x10</td>
<td>machine_id</td>
</tr>
<tr>
<td>0x40</td>
<td>0x04</td>
<td>cfg_port</td>
</tr>
<tr>
<td>0x44</td>
<td>0x04</td>
<td>cfg_versions_config</td>
</tr>
<tr>
<td>0x48</td>
<td>0x04</td>
<td>cfg_versions_dns_blocklist</td>
</tr>
<tr>
<td>0x4C</td>
<td>0x04</td>
<td>cfg_versions_url_tracklist</td>
</tr>
<tr>
<td>0x50</td>
<td>0x04</td>
<td>cfg_versions_plugin</td>
</tr>
<tr>
<td>0x54</td>
<td>0x04</td>
<td>cfg_versions_reserved1</td>
</tr>
<tr>
<td>0x58</td>
<td>0x04</td>
<td>cfg_versions_plugins</td>
</tr>
<tr>
<td>0x5C</td>
<td>0x04</td>
<td>cfg_versions_reserved2</td>
</tr>
<tr>
<td>0x60</td>
<td>0x04</td>
<td>taskid_error_1</td>
</tr>
<tr>
<td>0x64</td>
<td>0x04</td>
<td>taskid_error_2</td>
</tr>
<tr>
<td>0x68</td>
<td>0x04</td>
<td>taskid_error_3</td>
</tr>
<tr>
<td>0x6C</td>
<td>0x04</td>
<td>taskid_error_4</td>
</tr>
<tr>
<td>0x70</td>
<td>0x04</td>
<td>taskid_error_5</td>
</tr>
<tr>
<td>0x74</td>
<td>0x04</td>
<td>taskid_error_6</td>
</tr>
<tr>
<td>0x78</td>
<td>0x04</td>
<td>taskid_error_7</td>
</tr>
<tr>
<td>0x7C</td>
<td>0x04</td>
<td>taskid_error_8</td>
</tr>
<tr>
<td>0x80</td>
<td>0x04</td>
<td>taskid_error_9</td>
</tr>
<tr>
<td>0x84</td>
<td>0x04</td>
<td>taskid_error_10</td>
</tr>
<tr>
<td>0x88</td>
<td>0x04</td>
<td>taskid_error_11</td>
</tr>
<tr>
<td>0x8C</td>
<td>0x04</td>
<td>taskid_error_12</td>
</tr>
<tr>
<td>0x90</td>
<td>0x04</td>
<td>taskid_error_13</td>
</tr>
<tr>
<td>0x94</td>
<td>0x04</td>
<td>taskid_error_14</td>
</tr>
<tr>
<td>0x98</td>
<td>0x04</td>
<td>taskid_error_15</td>
</tr>
</tbody>
</table>
```
In looking at the CnC sources, I discovered that these items are decoded in a different way than the header. I discovered that these values are simply xored with the key value 22F071C2.

```
$g_btPost_StringXorKeys = array( 34, 240, 113, 194 );
```

When decoded, these parameters contain information about the infected system and align with the variable names found in the CnC server sources:

- `jshqf1 = C:\ProgramData\installFolderName\uznffnsd.exe`
- `jshqf2 = firefox.exe`
- `jshqf3 = [computername]\\[username]`
- `jshqf4 = Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz`
- `jshqf5 = VirtualBox Graphics Adapter`

### Decoding CnC servers HTTP response to the bot

#### 1. Overview of the HTTP Response from the CnC server to the bot

In the HTTP response, the first 4 bytes (Key A) will be used in the construction of a key to decrypt the header. The next 4 bytes (Key B) will be used to create a second key to decode the rest of the information. The next 0x5c block of data is the encrypted header information. We will start with decrypting this header. Any information following after the 0x5c block is extended data and will be decrypted separately.

```
00000000: 08 00 27 71-44 3F 08 00-27 25 B0 34-08 00 45 00 .................
00000010: 36 70 00 50-0C 28 C4 F0-7A EC E0 62-EC 35 50 18 .................
00000020: 01 00 BA 2E-00 00 48 54-54 50 2F 0D 0A 00 00 74 0D 0A 1B 7E 00 7E ..
00000030: 50 01 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00000040: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00000050: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00000060: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00000070: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00000080: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
00000090: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
000000A0: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
000000B0: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
000000C0: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
000000D0: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
000000E0: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
000000F0: 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..
```

#### 2. Construct the response header key

The first key is constructed by taking the value of the same “CnC Crypto Key 1” we extracted from the configuration data and appending the value of Key A shown above in the first 4 bytes of the HTTP response. Then XOR that value with 0xCB the same way we did to decode the HTTP request header.
Web key value value Key A
4A76D13B403499122210000

The resulting byte array after the XOR operation would then be:
81BD1AF07F18825AE9EACBC

3. Decrypt the response header

Next we will decrypt the next 0x54 bytes using RC4 with the key we constructed above. The resulting structure including the key values should be similar to the following:

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>0x04</td>
</tr>
<tr>
<td>0x04</td>
<td>0x04</td>
</tr>
<tr>
<td>0x08</td>
<td>0x04</td>
</tr>
<tr>
<td>0x0C</td>
<td>0x04</td>
</tr>
<tr>
<td>0x10</td>
<td>0x04</td>
</tr>
<tr>
<td>0x14</td>
<td>0x04</td>
</tr>
<tr>
<td>0x18</td>
<td>0x04</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x04</td>
</tr>
<tr>
<td>0x20</td>
<td>0x04</td>
</tr>
<tr>
<td>0x24</td>
<td>0x04</td>
</tr>
<tr>
<td>0x28</td>
<td>0x04</td>
</tr>
<tr>
<td>0x2C</td>
<td>0x04</td>
</tr>
<tr>
<td>0x30</td>
<td>0x04</td>
</tr>
<tr>
<td>0x34</td>
<td>0x04</td>
</tr>
<tr>
<td>0x38</td>
<td>0x04</td>
</tr>
<tr>
<td>0x3C</td>
<td>0x04</td>
</tr>
<tr>
<td>0x40</td>
<td>0x04</td>
</tr>
<tr>
<td>0x44</td>
<td>0x04</td>
</tr>
<tr>
<td>0x48</td>
<td>0x04</td>
</tr>
<tr>
<td>0x4C</td>
<td>0x04</td>
</tr>
<tr>
<td>0x50</td>
<td>0x04</td>
</tr>
<tr>
<td>0x54</td>
<td>0x04</td>
</tr>
<tr>
<td>0x58</td>
<td>0x04</td>
</tr>
</tbody>
</table>

KEY A KEY B Decrypted Block

4. Decrypt the response extra data

This is similar to as before, but this time we will use “CnC Crypto Key 2” and the next 4 bytes in the HTTP response and the static XOR key value of 0x1F

web key value value Key B
5f00fa7a21f477402A340000

The resulting byte array after the XOR operation would then be:
401FE5653EEB685F35281F1F
Betabot Configuration Data Extraction

We will decrypt the information using RC4 and the newly constructed key. The data block to decrypt comes immediately after the HTTP response header. With the above example, I had sent the bot a task to run notepad.exe.

The size of the information we want to decrypt with the new key is also specified in the previously decrypted header information at offset 0x3c (hdr_ext_cmds_size) which in this case has a value of 0x2c.

When this extra data is decoded it shows us the task being transmitted in this particular response.

```
00000000: 01 00 14 00-00 00 00 00-4F AF 1C 50-00 00 00 00 ............... 
00000010: 00 00 00 00-00 00 00 00-2E 73 79 73-20 2D 73 20 ........ sys -s 
00000020: 6E 6F 74 65-70 61 64 2E-65 78 65 00- notepad.exe. 
```

It should be noted that the transmission of a task is just one type of response that can be sent back to the bot. Other configuration information and instructions can be transmitted in this way as well. The extra data structures returned will be different for different types of data being sent back to the bot from the command-and-control server. The good news is they can all be decoded in the same way as this example.

**Summary**

Although the Betabot family has been around for a while, it is still prevalent and used to spread other malware campaigns and harvest site login credentials. The availability of a crack and the simplicity of the CnC web portal make it attractive to cybercriminals to use without putting forth a lot of effort.

**Acknowledgements**

Thanks to the following researchers for providing review feedback and technical assistance.

Andrew O'Donnell, Sophos
Dorka Palotay, Sophos
Fraser Howard, Sophos
Gabor Szappanos, Sophos

**References**

http://resources.infosecinstitute.com/beta-bot-analysis-part-1

http://resources.infosecinstitute.com/beta-bot-analysis-part-2


https://www.virusbulletin.com/virusbulletin/2014/05/neurevt-botnet-new-generation
https://www.arbornetworks.com/blog/asert/beta-bot-a-code-review

https://blog.fortinet.com/2014/01/29/neurevt-bot-analysis