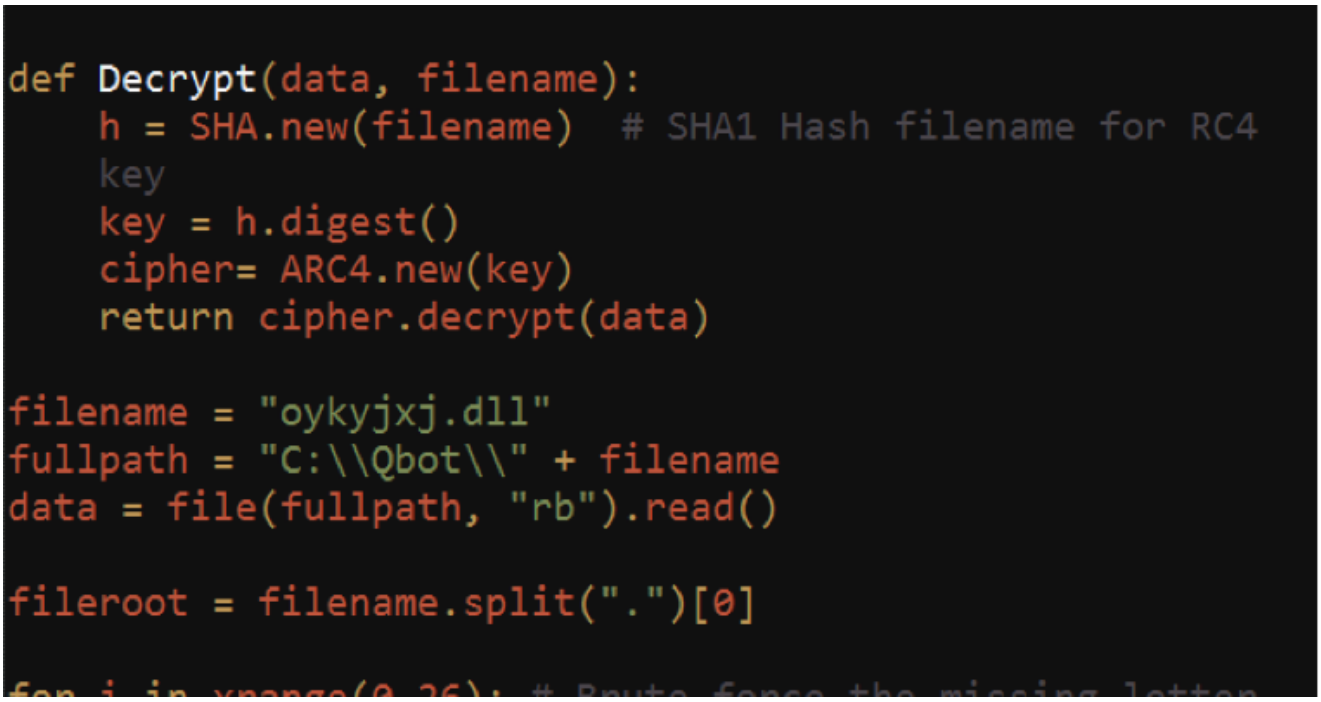
Research Spotlight: The Resurgence of Qbot

blog.talosintelligence.com/2016/04/qbot-on-the-rise.html



```
def Decrypt(data, filename):
    h = SHA.new(filename)  # SHA1 Hash filename for RC4
    key
    key = h.digest()
    cipher= ARC4.new(key)
    return cipher.decrypt(data)

filename = "oykyjxj.dll"
fullpath = "C:\\Qbot\\" + filename
data = file(fullpath, "rb").read()

fileroot = filename.split(".")[0]
```

The post was authored by Ben Baker.

Qbot, AKA Qakbot, has been around for since at least 2008, but it recently experienced a large surge in development and deployments. Qbot primarily targets sensitive information like banking credentials. Here we are unveiling recent changes to the malware that haven't been made public yet.

Qbot's primary means of infection is as a payload in browser exploit kits. Website administrators often use FTP to access their servers, so Qbot attempts to steal FTP credentials to add these servers to its malware hosting infrastructure. Qbot can also spread across a network using SMB, which makes it very difficult to remove from an unprotected network.

Packer

Qbot uses a packer that can change drastically between samples. The packer's strings and code blocks are randomized in ways that make it difficult to create a detection signature. Randomization is a common theme in Qbot since filenames, domain names, and encryption keys are randomly generated.

Thankfully, the code obfuscation doesn't complicate the behavior of the packer's code. The unpacking code consistently built an entire PE file in memory (likely an MD5 match with the original unpacked executable), then used a custom loader to execute the unpacked file. The unpacked executable could be reliably extracted from memory when the loader called the Windows API VirtualProtect. Since the unpacking code uses VirtualProtect to enable execution for the unpacked memory sections, it is possible to dump the unpacked code before it is given a chance to execute and infect the VM.

We took advantage of the packer's predictability by using a Pykd script to debug each packed file. We set a breakpoint on VirtualProtect, then scan the process's memory to dump the unpacked executable. We terminate the process before the unpacked code runs, so the VM shouldn't need to be reverted between samples. This script unpacks multiple samples per second, enabling us to unpack large numbers of samples.

We analyzed 618 packed samples, which unpacked to 73 unique samples. Each sample is packed multiple different ways to make hash signatures less effective.

Installation

Once the packer finishes loading the unpacked executable in memory, Qbot checks to see if it has already been installed. If Qbot is not running from "%appdata%\Microsoft\ [RandomName]\ [RandomName].exe", it copies itself there and executes the copy.

Qbot generates random strings by seeding a Mersenne Twister random number generator with the CRC32 checksum of a string or SHA1 hash, then repeatedly generating random integers to use as indices in an alphabet array to pick letters. The name of the install folder is generated using a seed generated from the ProductId (found in HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProductId), the name of the computer (found via the Windows API GetComputerNameA), and the hard drive serial number (found via the Windows API GetVolumeInformationA).

Qbot uses a deterministic string generator which makes it so the sample will always generate the same filenames and mutexes if it's run on the same computer. For example, if a sandbox uses static Product ID, Computer Name, and Volume serial numbers, the filenames will appear static even though they would be different for any other computer. This makes it difficult to create generic IOCs for customers, but easy to create sandbox IOCs for identifying Qbot.

Once Qbot is executed in its install directory, it executes a new instance of the Windows executable "Explorer.exe". The injected process loads the resource "IDB_BITMAP1" which contains a malicious DLL. The resource is decrypted using the first 20 bytes as an RC4 key.

The decrypted data contains the DLL in a compressed form, preceded by the SHA1 hash of the compressed data.

The compression appears to be custom, but is similar to LZSS with offset-length pairs pointing to bytes in the existing decompressed data as a dictionary. Files are broken into compressed blocks, each starting with a 24 byte header in the form of:

Magic Bytes[8]: “\x61\x6C\xD3\x1A\x00\x00\x00\x01”
Compressed Data Size[4]
Compressed Data CRC32[4]
Decompressed Data Size[4]
Decompressed Data CRC32[4]

The DLL is broken into 3 compressed blocks, which means Qbot checks 6 CRC32 checksums along with the SHA1 hash from when it was first decrypted. That’s a lot of error checking just to load one DLL.

Logging

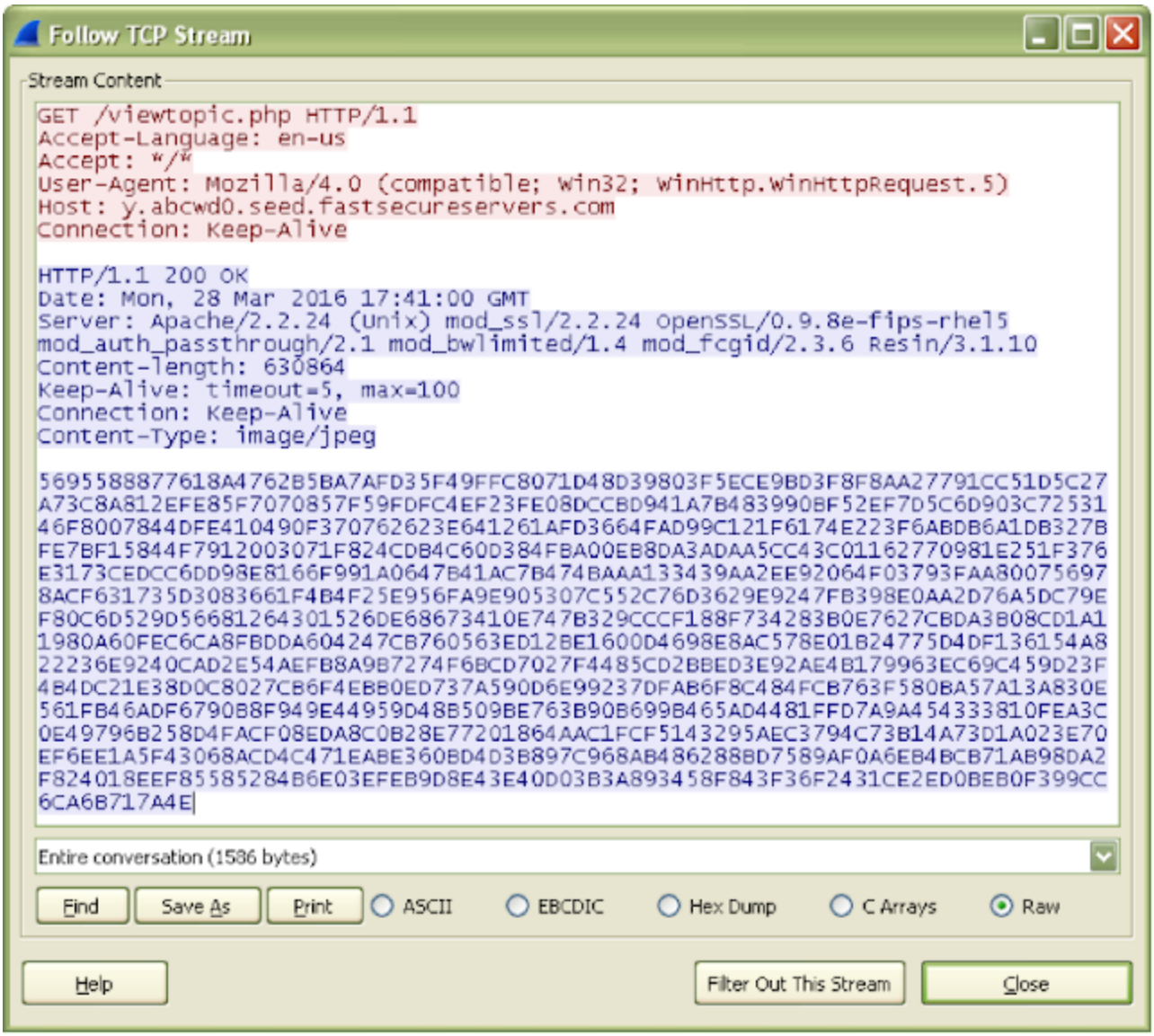
Qbot logs to an encrypted file in the install path. The log file can be identified as having a DLL extension, and a filename one letter short of the directory name where Qbot is installed in. For example, if Qbot is installed in “%appdata%\Microsoft\Oykyjxjx”, the log file will be “%appdata%\Microsoft\oykyjxjx\oykyjxj.dll”. The log file is encrypted with an RC4 key generated by converting the folder name to lowercase, then taking the SHA1 hash of the resulting string.

We’ve created a short Python script for decrypting the log file so incident responders can get additional information about infections. This will print out configuration information including initial infection time and FTP exfil server information.

```
from Crypto.Cipher import ARC4
from Crypto.Hash import SHA

def Decrypt(data, filename):
    h = SHA.new(filename)  # SHA1 Hash filename for RC4
    key
    key = h.digest()
    cipher= ARC4.new(key)
    return cipher.decrypt(data)

filename = "oykyjxj.dll"
fullpath = "C:\\Qbot\\" + filename
data = file(fullpath, "rb").read()

fileroot = filename.split(".")[0]

for i in xrange(0,26): # Brute force the missing letter
    letter = chr(0x61 + i)
    key = fileroot + letter
    decrData = Decrypt(data,key)

    if decrData[0:20] in SHA.new(decrData[20:]).digest():
     # We found the key if the first 20 bytes match the
    SHA1 of the remaining bytes
        print key + ": \n" + decrData[20:]
```

Updater

Qbot updates itself using an obfuscated script with the extension “.wpl”. This script attempts to download an encrypted executable hosted from numerous domains in URIs like "http[:]//<maliciousdomain.com>/viewtopic.php". The script hex decodes the server response, then uses the first 20 bytes as an RC4 key to decrypt the remaining bytes. The decrypted buffer contains a 20-byte SHA1 hash, followed by the updated version of the Qbot executable.



Info Theft

Qbot primarily targets sensitive information like banking credentials. It does so by stealing data like stored cookies or credentials, and by injecting code into web browsers to manipulate live browsing sessions. Qbot lets malicious actors piggyback on the victim's browsing sessions, enabling them to bypass security like simple implementations of two-factor authentication.

Qbot recently added Webinjects to its arsenal of info stealing techniques. Webinjects, commonly associated with Zeus and Spyeye, make it easy to inject malicious javascript in browser sessions while logging or redirecting the victim's activity. Webinjects can be very powerful, in some cases automating large banking transactions without any interaction from the user.

Qbot contains code for parsing advanced Webinjects like automated transfers, but all 618 Qbot samples we analyzed simply redirected the browsers when the victim attempts to logout of an online banking page. Stolen cookies and session tokens remain active for much longer if the victim is unable to logout of the targeted sites.

Sample Qbot Webinject config:
“set_url https://*.<BankDomain>.com/*logoff* GPR
http://<MaliciousSite>/fakes/onlineserv_cm_logoff.html”

In this example of a Webinject, any hooked browser attempting to navigate to the logoff page for the targeted bank would be intercepted. The “GPR” flags indicate GETs and POSTs will be intercepted and Redirected to the malicious URL. This R flag seems to be rarely used by other malware that support Webinjects. Malware forum posts about Webinjects focus mainly on injecting malicious code or simply logging HTTPS parameters like credentials.

Command and Control

FTP Exfil

Qbot exfiltrates data over FTP to a list of servers hardcoded in its config file. The exfil files are compressed, then RC4 encrypted with a randomly generated key, similar to how resources are encrypted inside the executables. I’m intentionally leaving out how to decrypt these files, since disclosing that info could enable anyone to retrieve sensitive information stolen by Qbot.

The exfil files are uploaded to the FTP servers with file names like “article_covezh618946_1450458170.zip” where “article” is hard coded, “covezh618946” is randomly generated, and 1450458170 is seconds since Linux Epoch (Dec 18, 2015 in this case).

```
Info
Response: 220---------- Welcome to Pure-FTPd [privsep] [TLS] ----------
Request: USER wpadmin@
Response: 331 User wpadmin@                    OK. Password required
Request: PASS
Response: 230-OK. Current restricted directory is /
Request: TYPE I
Response: 200 TYPE is now 8-bit binary
Request: PASV
Response: 227 Entering Passive Mode (                    )
Request: STOR article_covezh618946_1450458170.zip
Response: 150 Accepted data connection
Response: 226-6078415 Kbytes used (11%) - authorized: 51200000 Kb
Response: 226 Logout.
```

The decrypted exfil file contains a lot of information about the victim’s machine, a chunk of which is formed with a call to the function wvnsprintf using the format string:

" ext_ip=[%s] dnsname=[%s] hostname=[%s] user=[%s] domain=[%s] is_admin=[%s] os=[%s] qbot_version=[%s] install_time= %s] exe=[%s]"

The qbot_version string is generated by a previous call to wvnsprintf using the format string "%04x.%u" with parameters that trace back to the first 2 DWORDs in the data section. Extracting these DWORDs directly proved to be an easy and consistent way to extract Qbot's version information without running the malware.

HTTP DGA

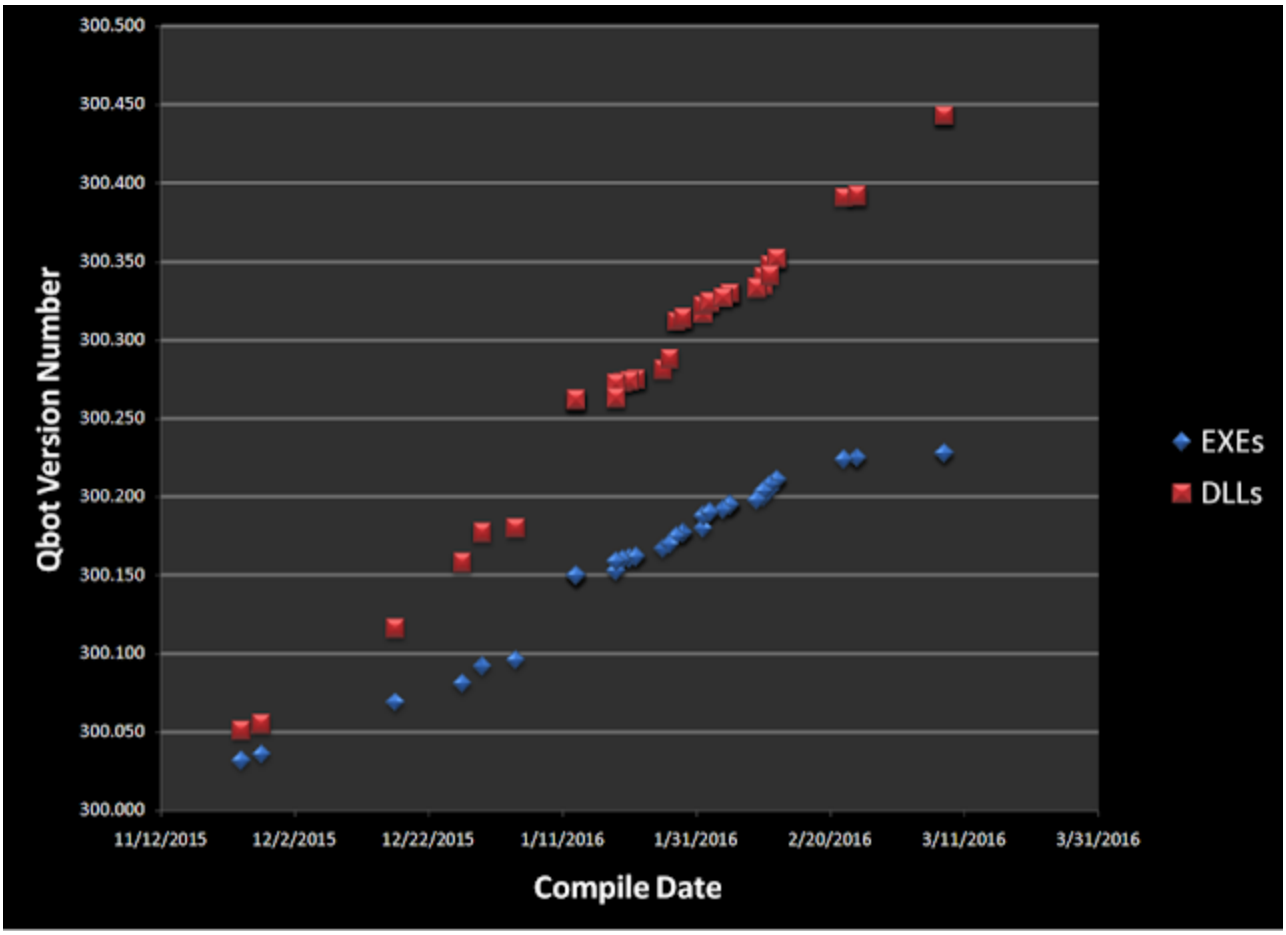
Qbot's DGA generates a string like "2.mar.2016.00000001" where 00000001 is constant and the first digit is the tens digit of the day of the month (though 2 is also used for days 30 and 31). This means there are only 3 DGA seeds for each month. Qbot gets the date by sending an innocent looking HTTP request to Google, and parsing the date from the HTTP 301 response.

The date string is CRC32 checksummed and used as a seed for the Mersenne Twister random number generator, which Qbot uses to create a predictable list of domains. Qbot generates a list of 5 domains at a time, and randomly picks from them until it finds an active domain. If none of the first set of domains are active, it will generate a new set and repeat as necessary.

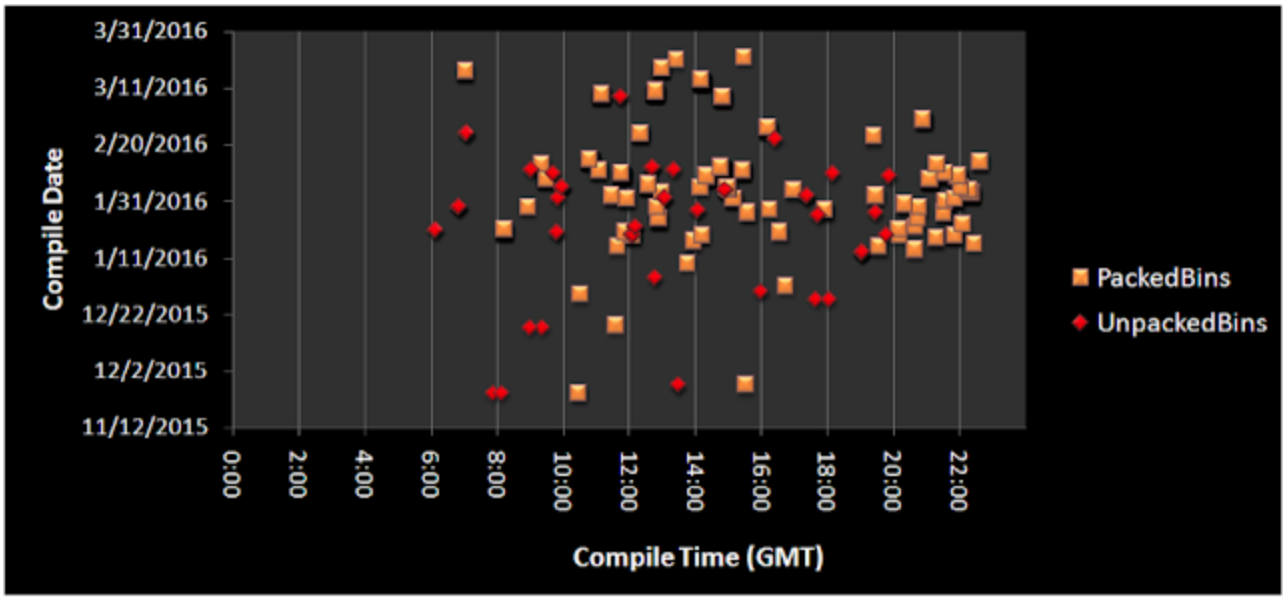
Qbot checks DNS responses for any strings containing "sinkhole" or ".csof.net" and skips those results. It also checks for monitoring tools like Wireshark, and will alter the seed if any are found. The altered seed causes the malware to generate a fake list of domains.

Evolution

We automated unpacking 618 Qbot samples via Pykd, then created Python scripts to decrypt and decompress the embedded resources. We extracted DLLs and config data, as well as Qbot version information and Compile Times for each file. Compile times are often used to attribute malicious activity, though it is important to note compile times can be manipulated.



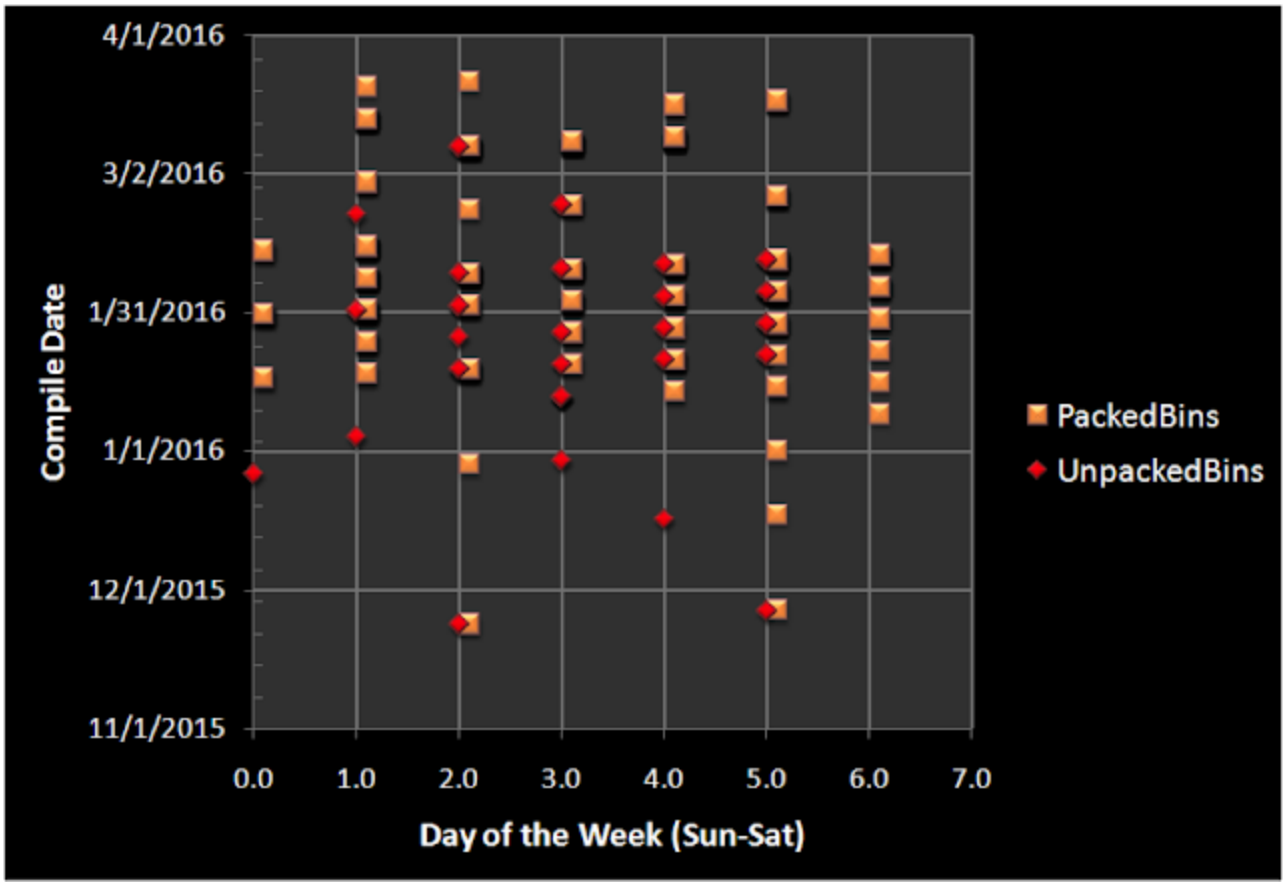
Qbot compile times show a surge of activity in January and February, when developers released multiple versions per day on at least 4 days during this period. Qbot's authors appear to have stopped development on 3/08/2016, although their servers continued to host Qbot executables packed as late as 3/25/2016.



This graph shows 132 unique compile timestamps showing time of day for each binary. The unpacked binaries all have compile times between 6am and 8pm GMT. This time distribution looks more like a full-time job than a side project. The packed binaries have compile times with a lot more variance, suggesting the packing is either done by another person with a much more flexible schedule or with the assistance of some kind of automation. Developing source code requires a lot more experience than running a packing tool, so packing could be done by less technically skilled members of the team.

The packed compile times mostly fall between 8am and 10pm GMT, 2 hours shifted from the average work day for the unpacked samples. If the packing process was completely automated, we would expect to see these compile times distributed more randomly (with some binaries compiled between midnight and 6am) or with a predictable schedule such as once a day at a specific time.

55% of packed binaries have compile timestamps within the range of 2h3m and 2h6m after their unpacked counterparts. This consistency of time shift may mean these files were packed according to a very consistent process that takes just over 2 hours, or that the computer doing the packing could have its system clock off by 2 hours.



This graph shows the day of the week using offsets from Sunday (e.g. 1 = Monday, 2 = Tuesday, etc.). The compile times for the unpacked binaries form a nice Monday-Friday work

week. Over this 17 week period, only one set of binaries was compiled during a weekend.

The packed binaries continue to show a much greater variance in compile times, with binaries compiled on 6 Saturdays in a row. There is at least one packed binary compiled per day between Jan 27 and Feb 6, showing work on 11 consecutive days.

We also analyzed the Rich Headers associated with Qbot binaries. Rich Headers are undocumented sections in Visual Studio executables that contain information about the compiler and linker version used to build the executable. Rich Headers can vary greatly between two seemingly identical development environments, so they may give clues into how many computers are used in a development project.

The 154 unpacked binaries contained only 6 unique Rich Headers in, some of which were almost identical and were likely caused by minor compiler updates on the same computer. These headers suggest the unpacked binaries were compiled in 3 unique environments (likely different computers or Virtual Machines). The packed binaries contained 44 unique Rich Headers, 35 of which seemed to be slight variants of the others. The packed binaries appear to be compiled from 9 unique environments, none of which matched the 3 Rich Headers from the unpacked binaries.

The differences in these Rich Headers further supports the theory that packing is done on different computers than the ones that develop and compile the main Qbot source code. The Rich Header data suggests there are 12 unique environments for compiling and packing binaries, which could give a hint to the size of the team developing and packing Qbot.

Developing and maintaining malware and a malicious infrastructure requires a lot of time and effort. The malicious actors behind large scale crimeware like Qbot work as a team, and approach their nefarious activities as a full time job.

Coverage

For the most current rule information, please refer to your Defense Center, FireSIGHT Management Center or Snort.org.

PRODUCT	PROTECTION
AMP	✔
CWS	✔
ESA	N/A
Network Security	✔
WSA	✔

Advanced Malware Protection (AMP) is ideally suited to prevent the execution of the malware used by these threat actors.

CWS or WSA web scanning prevents access to malicious websites and detects malware used in these attacks.

The Network Security protection of IPS and NGFW have up-to-date signatures to detect malicious network activity by threat actors.