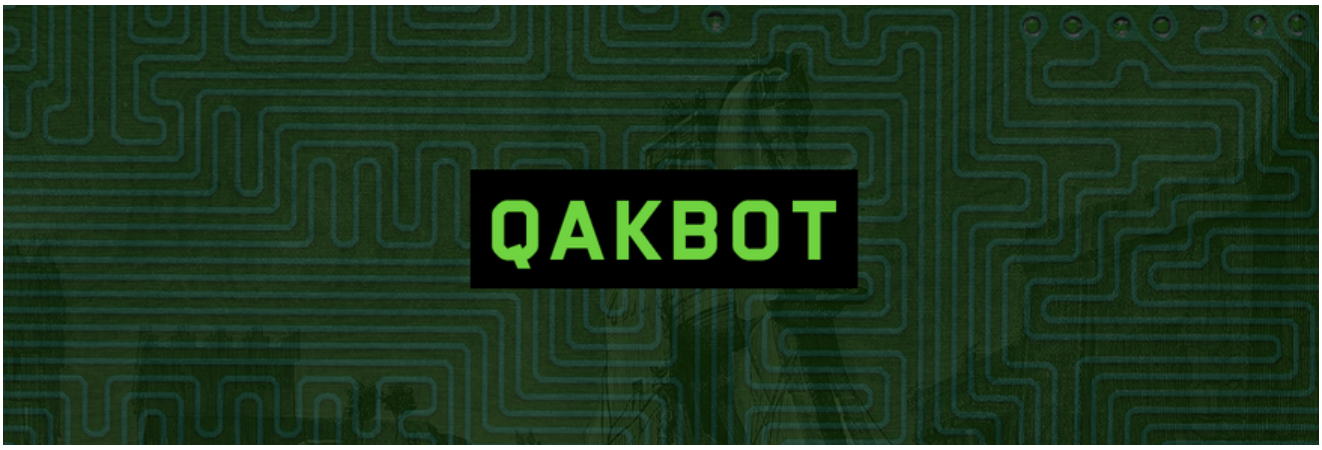
Threat Spotlight: The Return of Qakbot Malware

cylance.com/en_us/blog/threat-spotlight-the-return-of-qakbot-malware.html

The BlackBerry Cylance Threat Research Team



Introduction

The Cylance Threat Guidance team continuously looks for interesting trends and challenges that incite organizations to engage with us. Recently, an influx of thousands of Qakbot (aka Qbot) infections have brought people to us to discuss how to block this malware from gaining access to corporate systems.

Qakbot has been around for years (in fact, we made this video about it last April), but it's nothing to be complacent about. This malware is well-known for its ability to steal credentials and quickly spread through an enterprise over network shares. Given its age, it might seem logical that security controls would have this threat on lockdown. However, the occasional functional enhancements combined with its multiple layers of obfuscation and server-side polymorphism periodically breathe new life into this seemingly immortal malware.

While it's unclear why so many systems have suddenly fallen victim to Qakbot, it's possible that updated exploit kits play a role. After all, there is no shortage of new vulnerabilities and exploits for attackers to use to their advantage.

Since the core functionality of Qakbot has remained fairly consistent over the years and is well documented, we will not rehash that information here. Instead, we'll evaluate several samples from the recent surge in infections and explore how to correlate unique binaries.

Collecting Qakbot

As highlighted at the beginning of this article, the polymorphic nature of this threat is of great interest given its goal of evading detection. Prior versions of Qakbot configured a scheduled task to request updates, and the recent wave of infections was no different. For example, we observed a sample that configured the following command to run on a weekly basis:

cmd.exe /C "start /MIN C:\windows\system32\cscript.exe //E:javascript "C:\Users\ <USER>\AppData\Local\Microsoft\<5-8 random alphabetical characters>.wpl""

As described in the command line, the WPL file contains JavaScript. Just in case the cryptic code isn't convincing enough, the header of the target file makes it all too clear the JavaScript is obfuscated *(Figure 1)*.

Figure 1: Excerpt of Obfuscated JavaScript Update Script

Brief behavioral analysis revealed the code reaches out to the following URLs:

- hxxp://css.kbaf.myzen.co(dot)uk/TealeafTarget.php
- hxxp://projects.montgomerytech(dot)com/TealeafTarget.php
- hxxp://n.abcwd0.seed.fastsecureservers(dot)com/TealeafTarget.php

Like earlier versions of Qakbot, a request to these update servers returns an encrypted payload, where the first 20 bytes serve as the RC4 key to decrypt the data. Once decrypted, the first 20 bytes represent a SHA1 hash of the executable, and the remaining bytes are the file contents.

To mimic Qakbot's update process in a controlled manner, we created a Python script to send HTTP requests to each of the three URLs over a 24-hour period. At the time of this writing, requests for the first listed URL returned an *HTTP 404 Not Found* error, but requests for the remaining two URLs returned the expected payload.

The script was executed from both a Windows 10 64-bit and Windows 10 32-bit OS for more than 24 hours from 5/16/17 to 5/17/17, and the scripts were running simultaneously across both operating systems during much of that time.

Although the script was configured to send HTTP requests to each of the two working URLs every 30 seconds, most requests resulted in pulling the same binary. In other words, the update server did not provide a new file with each request.

Files with a unique hash were only supplied approximately every 10 minutes. In total, we collected 245 files across the two machines. However, because the servers supplied the same file to each machine at a given time (i.e., they appeared to be in sync), the resulting unique set of files was much smaller, totaling 141 files.

Reviewing the Collection

All 141 downloaded files were 32-bit Windows executables. Searching public repositories for these files revealed that only one had been previously logged, and that was within the previous 12 hours.

Across the 141 files, all have unique compile timestamps, and the earliest one occurred on May 15, 2017.

While all 141 files have unique sha-256 file hashes, there are some similarities to consider. First, calculating the import hashes across the files showed three distinct groups:

- 2E6AC2290F1E3D80ABC8D0D6F736D965
- 651EF2DBA96011F47EED9B72BE7B4B8C

- F3CAA54DDE4056FADD52A024CF6B82ED

Although import hashes are often used to correlate malware over long periods of time, the earliest compile timestamp we discovered for a file with any of the above import hashes was 5/15/17. Given the polymorphic capabilities of this threat, this observation is not surprising.

Let's briefly take a look at two files that have the same import hash (2E6AC2290F1E3D80ABC8D0D6F736D965) but different file hashes:

- 7DBD0DF279062090C34F796EFC7DD239ECCD46B99B67AAC370D6048D5ADBB9EC
- 67F3BD674647CA0D294A894B6702362B6CFC4B6C2E147643E100903A6B4D715A

Both files are 458,752 bytes, and they consist of the following PE sections:

- .text
- .code
- .rdata
- .data
- .CRT
- .exp
- .code (yes, again)
- .rsc
- .reloc

Among these, all section hashes match except those for .text, .rdata, and .data. A different .text section may indicate a change in executable code. To investigate this observation, we can use diaphora, a binary diffing tool compatible with IDA Pro. Performing a code comparison presents the following results *(excerpt in Figure 2).*

Figure 2: Binary Diff of Two Qakbot Samples With the Same Import Hash

Diaphora concludes all 27 identifiedfunctions are a 100% match.

While we could investigate individual variations across the three PE sections mentioned earlier, it makes more sense to unravel any layers of obfuscation and compare the underlying code.

Unpacking Qakbot

To unpack recent Qakbot samples, the below approach worked reliably. All instructions assume use of x32dbg, but similar steps can be performed with a debugger of your choice.

- Load the sample into x32dbg (we'll be working with the file that has sha-256 hash 7DBD0DF279062090C34F796EFC7DD239ECCD46B99B67AAC370D6048D5ADBB9EC) .
- Set a breakpoint on VirtualProtect *(Figure 3).*

Figure 3: Set a Breakpoint on VirtualProtect Within x32dbg

• Execute the code. On the first call to VirtualProtect, the protection on all sections owned by the process will be modified to 0x04, or PAGE_READWRITE *(see red box in Figure 4).*

Figure 4: Protection on Code Changed to PAGE_READWRITE

- As the code executes, those sections in memory will be manually overwritten using loops.

Let the breakpoint hit two more times, and on the second hit notice that protection on the .text section is changed to 0x20, or PAGE_EXECUTE_READ *(see red box in Figure 5)*.

Figure 5: Protection on .text Section Changed to PAGE_EXECUTE_READ

- Allow the call to complete and return back to user code so that the permissions change takes effect.

- Next, browse to the Memory Map and choose to disassemble the .text section *(Figure 6).*

Figure 6: Memory Map With Unpacked Code

• Once there, set a “Hardware on Execution” breakpoint to catch when this new code is executed *(Figure 7).*

Figure 7: Hardware Breakpoint on Original Entry Point (OEP)

• Run the code until the hardware breakpoint is triggered. You can then dump the process using a plugin like OllyDumpEx and fix the import table using Scylla.

Correlating Qakbot

Following the above process for both files produced two process dumps of the same size (418,304 bytes) and different file hashes. Turning again to diaphora revealed that most of the 227 functions identified matched 100%. Only nine functions did not match 100% *(see excerpt in Figure 8).*

Figure 8: Binary Diff of Unpacked Code From Two Samples

A review of these nine functions showed that each referenced the file name and/or location of the file on disk. Since each file tested did indeed have a different filename and location, we can explain this discrepancy. Therefore, despite variations in the sections across the code, the resulting unpacked binary contains identical functionality.

To further explore correlation between deobfuscated samples, we can apply the same unpacking process to a sample with a different import hash. For example, the file with hash 8891524E468BE1BD44723385C9238017090B536F922CCC007D8AC47C66802E3C is 450,560 bytes and has the import hash 651EF2DBA96011F47EED9B72BE7B4B8C. It is 8,192 bytes smaller than the previous two files and has only six sections (no section hashes match when compared to previous samples):

- .text
- .code
- .rdata
- .data
- .rsrc
- .reloc

The unpacking approach outlined above results in another dumped 418,304 byte file with a different file hash. However, a code diff confirms that most of the 227 identified functions match 100%, and the nine functions that do not only differ by the filename and path, as described earlier.

Conclusion

Qakbot continues to be a significant threat due to its credential collection capabilities and polymorphic features. Unhindered, this malware family can rapidly propagate through network shares and create an enterprise-wide incident. In this post, we explored how to dissect, unpack, and compare multiple downloaded samples.

If you use our endpoint protection product CylancePROTECT®, you were already protected from this attack. If you don't have CylancePROTECT, contact us to learn how our AI-driven solution can predict and prevent unknown and emerging threats.

Indicators of Compromise (IoCs)

Sha-256 Hashes:

7DBD0DF279062090C34F796EFC7DD239ECCD46B99B67AAC370D6048D5ADBB9EC
67F3BD674647CA0D294A894B6702362B6CFC4B6C2E147643E100903A6B4D715A
8891524E468BE1BD44723385C9238017090B536F922CCC007D8AC47C66802E3C

URLs:

hxxp://css.kbaf.myzen.co(dot)uk/TealeafTarget.php
hxxp://projects.montgomerytech(dot)com/TealeafTarget.php
hxxp://n.abcwd0.seed.fastsecureservers(dot)com/TealeafTarget.php

About The BlackBerry Cylance Threat Research Team

The BlackBerry Cylance Threat Research team examines malware and suspected malware to better identify its abilities, function and attack vectors. Threat Research is on the frontline of information security and often deeply examines malicious software, which puts us in a unique position to discuss never-seen-before threats.