Qakbot Series: String Obfuscation

Published: 2022-04-10 · Archived: 2026-04-05 14:23:34 UTC

In late March 2022, I was requested to analyze a software artifact. It was an instance of Qakbot, a modular information stealer known since 2007. Differently to other analyses I do as part of my daily job, in this particular case I can disclose wide parts of it with you readers. I'm addressing them in a post series. Here, I'll discuss about the string obfuscation techinque based on this specific sample.

Looking at the strings embedded into a software artifact is one of the first approaches an analyst may attempt during the triaging stage. For those of you don't know what I'm talking about, strings are sequences of bytes that, once interpreted as characters, form meaningful words. During the process that starts with a program source code and ends with that program being compiled and ready to run, strings are *usually* preserved. What does that mean? It means that if you have a string in the source code, e.g. a path or a function name, then that string will lie into the object code after the compilation of the sources. Strings may be an useful source of information to quickly understand the capabilities of a piece of software and immediately focus on specific areas of the artifact deserving a closer look. Strings can be statically extracted, i.e. you don't need to exectute the software to obtain them.

Naturally, malware developers know the importance of the strings during the analyis process. Therefore, they tend to hide this source of information from their products. One easy and cheap technique to hide strings like variable names and function names is to stripe them from the binary. This is acheavable simply by compiling the source with particular flags. Another technique aimed at hiding the most valuable strings is called **string obfuscation**. String obfuscation consists in storing the strings in encrypted or obfuscated form so that they cannot be recognized and extracted from the artifact. Those strings are decrypted at runtime and consumed by the software when they are needed. The Qakbot sample I analyzed implements a string obfuscation technique.

Indeed, the strings analysis for the sample object of analysis wasn't really fruitful when I tried to do it. There were not so many meaningful strings overall and most of them were unreferenced or, in general, not providing any insight. There was only one exception: the presence of 18 very long and apparently meaningless strings. We postpone a discussion about their purpose to another post since it regards another anti-analysis technique. The reason why the string analysis wasn't so effective is that the vast majority of them are obfuscated to hide evidence of the malware capabilities.

```
undefined * __thiscall deobfuscate_string(undefined *blob,uint blob_size,undefined *key,uint offset)

{
  int iVar1;
  undefined *puVar2;
  uint uVar3;
  byte *pbVar4;
  uint local_8;

  local_8 = 0;
  uVar3 = offset;
  if (offset < blob_size) {
    do {
      if (key[uVar3 % 0x5a] == blob[uVar3]) {
        local_8 = uVar3 - offset;
        break;
      }
      uVar3 = uVar3 + 1;
    } while (uVar3 < blob_size);
  }
  puVar2 = (undefined *)allocate_heap(local_8 + 1);
  uVar3 = 0;
  if (puVar2 == (undefined *)0x0) {
    puVar2 = &NULL;
  }
  else {
    if (local_8 != 0) {
      do {
        pbVar4 = puVar2 + uVar3;
        iVar1 = uVar3 + offset;
        uVar3 = uVar3 + 1;
        *pbVar4 = key[(uint)(pbVar4 + (offset - (int)puVar2)) % 0x5a] ^ blob[iVar1];
      } while (uVar3 < local_8);
    }
  }
  return puVar2;
}
```

Figure 1

-

Qakbot string deobfuscation function

All the meaningful and relevant strings are stored in obfuscated form in two continuous blobs. The first blob is located at *0xb542b8* and the second blob is located at *0xb551f8*. A string is de-obfuscated by xor-ing the specific part of the blob with a key stored as a continuous sequence of bytes. Each blob is xor-ed with a different key. There are two instances of the function implementing the string de-obfuscation, one starts at *0xb302c6* and another one starts at *0xb227a1*. As you may notice from **Figure 1**, showing the decompiled code for one of those instances, it expects four arguments: the blob where the string is contained, the size of the blob, the xoring key, and the starting offset of the obfuscated string within the blob.

```
def deobfuscate_string(blob1: bytes, p1: int, blob2: bytes, p3: int) -> bytes:
   l8 = 0
   i = p3
   if p3 <= p1:
       while i <= p1:
           if blob2[i % 0x5a] == blob1[i]:
               l8 = i - p3
```

```
                break
            i += 1
    lc = bytearray([0] * (l8))
    i = 0
    if l8 > 0:
        while i < l8:
            lc[i] = blob2[(p3 + i) % 0x5a] ^ blob1[p3:][i]
            i += 1
    return bytes(lc)
```

Listing 1

-

Python translation of the string deobfuscation function

Listing 1 shows a Python3 translation of the Qakbot string deobfuscation function. I had to code it since the offset of many strings into the sample code is computed at runtime instead of being hardcoded. That function replicates the algorithm implemented in the sample. Here you will find a complete list of the de-obfuscated strings produced by our script. For each string I mention the containing blob and the starting offset within the blob. In that list you'll find a lot of potentially interesting elements regarding the malware capabilities. I'll discuss about some of them in the coming blog posts.

As always, if you want to share comments or feedbacks (rigorously in broken Italian or broken English) do not esitate to drop me a message at **admin[@]malwarology.com**.
