Old Malware Tricks To Bypass Detection in the Age of Big Data

By Suguru Ishimaru

Published: 2017-04-13 · Archived: 2026-04-05 15:09:26 UTC

Kaspersky Lab has been tracking a targeted attack actor's activities in Japan and South Korea recently. This attacker has been using the XXMM malware toolkit, which was named after an original project path revealed through a pdb string inside the file: "C:\Users\123\documents\visual studio 2010\Projects**xxmm2**\Release\test2.pdb". We came across an unusual technique used by a sample which contained no pdb strings but was very similar to a variant of XXMM malware in terms of code similarity, malware functionality, crypto-algorithm, data structures and module configuration.

The malware sample we observed was named "srvhost.exe" to resemble a standard system process name. It came from one of our partners at the beginning of 2017. One of the most surprising features of the malware was its file size, which is not commonly seen in malware – it was **over 100MB**. According to our analysis, this malware is a Trojan loader component that activates a backdoor. We could not confirm pdb strings from this malware, however the backdoor module seems to be named "wali" by the author, according to strings from the embedded config block.

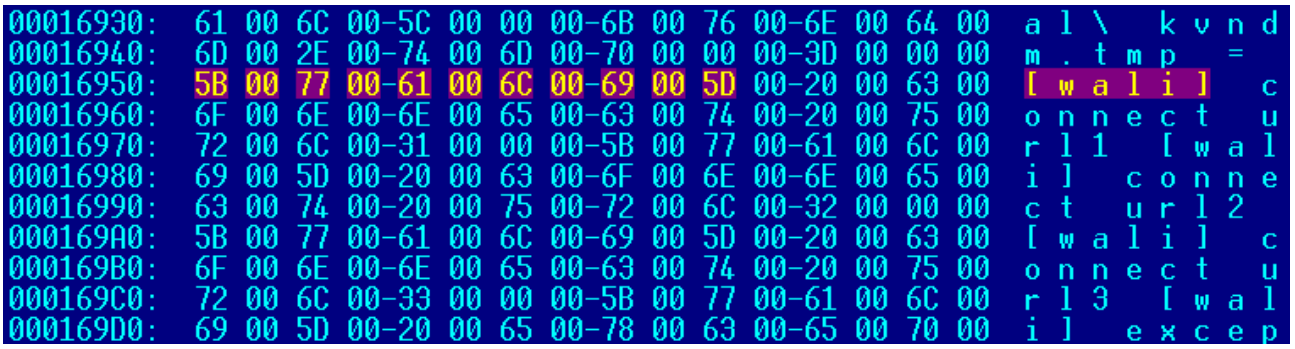

Fig. config strings with "[wali]" section

Fig. "wali.exe" name in the malware body

The wali loader decrypts the embedded wali backdoor using the "\x63" byte and a simple XOR operation. The XOR key is not only "\x63", we confirmed others. Then, the wali backdoor module is injected into the memory of the iexplore.exe process by the loader.

What is inside the wali loader that makes it so big in size? The reason is that this sample has a very big overlay of junk data. We found more than 20 other similar samples (wali loader + overlay) using open source intelligence and by searching our malware collection using YARA rule. After removing the overlay, there were only six unique samples.

md5_payload	md5_payload+overlay	size
d1e24c3cc0322b22988a1ce366d702e5	8bd0ddeb11518f3eaaddc6fd82627f33	105982049
e4811950899f44f9d14a786b4c5b1faa	2871ec229804a6e872db55dafa5c9713	105997178
	3e24710d7ade27316d367dd8cb2a0b1a	105996860
	3e9feea893482b65a68b1feecb71cd4d	105997043
	558ca7fa8ed632fa4f8c69e32888af0f	105997191
	d11f7b25823ce474e30e8ab9c8d567b0	105996847
	f4c3f06faf53ad2bbc047818344a2323	105997181
	f7cc6a5a06cd032c6172d14c1568b976	105997102
e7492f11c88d32e1e0b43f6b29604ec8	6a5558e4ab530f9b5c2d5bcc023d3218	105997658
	bb8cef31cf6211c584d245be88573e1f	105997755

Table. Some samples of 100M+ bytes wali loader + overlay

The overlay data is generated by the wali dropper when the wali loader is installed onto the victim's machine. The following figure shows the structure of malware components and how they are related to each other:

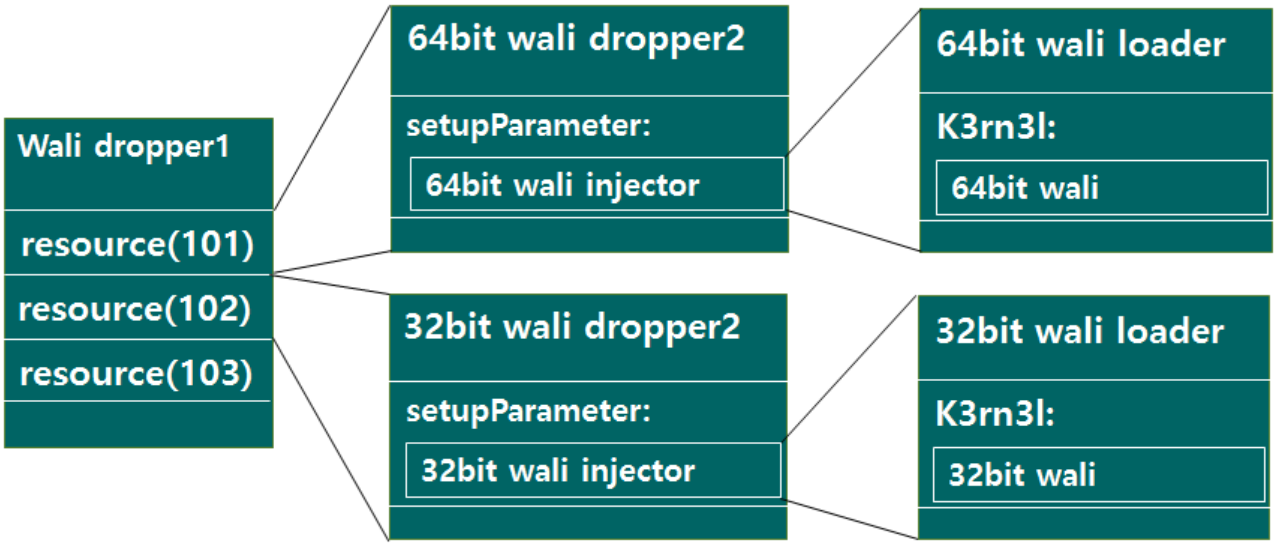

Fig. Structure of wali modules

Wali dropper1 checks the CPU architecture. If the CPU is 64-bit, this malware decrypts the 64-bit version of the wali loader from resource id 101. Otherwise, it decrypts the 32-bit version of the wali loader from resource id 102. To extract the resource data it uses RC4 with "12345" as the cryptokey, and LZNT1 to decompress the data after that. Dropper1 creates a file named "win${random4 chr}.tmp.bat" in the current temp directory from the decrypted wali dropper2 data. Finally, it appends generated **garbage data** to the overlay of the dropped file and runs wali dropper2

Wali dropper2 checks if the user account has admin privileges, and decrypts the wali loader using the same algorithm and the same key as of dropper1, and creating new files using the following file paths:

- %ProgramFiles%\Common Files\System\Ole DB\srvhost.exe
- %appdata%\Microsoft\Windows\Start Menu\Programs\srvhost.exe

It also appends generated **garbage data** to the overlay as well, using the same function. Finally, it creates a registry value of "sunUpdate" in "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run" to ensure malware persistence.

Generation of Junk Data

The feature to appending junk data to the malware executable to inflate the file size is quite unique to wali dropper1 and wali dropper2. We assume that by creating a large file the authors wanted to avoid AV detection, complicate sample exchange and stay below the radar of the most commonly used YARA rules. The function that generates the junk data is shown below:

```
sub_402E10((char *)&unk_40C11C, 0, a1);
v1 = _time64(0);
srand(v1);
rand();
v2 = rand() % 500 + 0x100000;
lpAddress = (char *)VirtualAlloc(0, v2, 0x1000u, 4u);
if ( lpAddress )
{
  count = 1000;
  do
  {
    v3 = rand() % v2;
    v4 = rand() % 256;
    v5 = rand();
    memset(&lpAddress[v3], v4, v5 % (v2 - v3));
    --count;
  }
  while ( count );
  sub_402E10(lpAddress, (void *)v2, a1);
  VirtualFree(lpAddress, 0, 0x8000u);
  result = a1;
}
```

Fig. Function to create junk data (create_garbage_data).

The create_garbage_data function generates a random byte in a loop with 1,000 iterations. In every iteration it fills blocks of data of random length within certain dynamically calculated limits. After that the result of create_garbage_data is written to the overlay of the decrypted wali loader and the process is repeated 100 times. This produces junk data of ~100MB which is appended to the executable.

```
WriteFile(v19, v20, v21, v22, v23);
count = 100;
do
{
  --count;
  create_garbage((int)&lpBuffer);
  created_garbage_data = lpBuffer;
  if ( v33 < 0x10 )
    created_garbage_data = &lpBuffer;
  WriteFile(hFile_1, created_garbage_data, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0);
  if ( v33 >= 0x10 )
    operator delete((void *)lpBuffer);
}
while ( count );
CloseHandle(hFile_1);
```

Fig. Loop to append the junk data to overlay.

The size of one wali loader (MD5: d1e24c3cc0322b22988a1ce366d702e5) was initially **1,124,352 bytes**. The function that appends garbage produced a new malware file in a real attack (MD5: 8bd0ddeb11518f3eaaddc6fd82627f33) and the file size was increased to **105,982,049 bytes**.

As the appended junk data is created dynamically and depends on random values, the size of it may vary. We have seen **100MB** files as well as **50MB** samples used in real world attacks. The largest we observed was a **200MB** malware sample created with the same trick. This technique currently doesn't affect detection of the malware by Kaspersky Lab products. The malware is detected as:

- Trojan.Win32.Xxmm
- Trojan.Win64.Xxmm
- Trojan-Downloader.Win32.Xxmm
- Trojan-Downloader.Win64.Xxmm
- Trojan-Dropper.Win32.Xxmm
- Trojan-Dropper.Win64.Xxmm

Inflating file size with garbage data is not a completely new technique. Previously polymorphic viruses and worms used this technique a lot to mix original code with garbage data spread across the malware file, sometime increasing the file size by hundreds of kilobytes and even megabytes. Certain software protectors may also insert decoy files into packed files and inflate file size up to 1MB. We have also seen executable malwares disguised as movie files and ISO files spread over torrents, which in these cases, the malware size is inflated to a few gigabytes in order to mimic true content .

What is quite unique in using this method and appending junk data to a file is that in this case this technique is used in targeted attacks and is happening after the initial infection, during the later phases of attack with the intention of increasing file size to avoid detection.

While this technique may seem inefficient in its primitive approach to bypass detection, we believe that in certain cases this malware may stay below the radar of incident responders and forensic analysts who use YARA rules to scan harddrives. The reason is that one of the common practices for YARA rule authors is to limit the size of scanned files, which is aimed mainly at improving performance of the scanning process. Large files, like the ones produced by XXMM malware, may become invisible for such rules, which is why we would like to recommend security researchers to consider this when creating rules for dropped malwares.

Indicators of Compromise

SHA256sum of samples

Wali dropper1:

- 9b5874a19bf112832d8e7fd1a57a2dda180ed50aa4f61126aa1b7b692e6a6665

Wali dropper2:

- da05667cd1d55fa166ae7bd95335bd080fba7b53c62b0fff248ce25c59ede54a
- 10fca84ae22351356ead529944f85ef5d68de38024d4c5f6058468eb399cbc30

Wali loader + overlay:

- 1f73d3a566ab7274b3248659144f1d092c8a5fc281f69aa71b7e459b72eb6db2
- 24835916af9b1f77ad52ab62220314feea91d976fdacad6c942468e20c0d9ca1
- 303c9fabf6cff78414cebee9873040aeb9dcf6d69962bd9e0bbe1a656376ed16
- 3ffd5d3579bddbfd7136a6969c03673284b1c862129cfafe7a40beea1f56e790
- 803a5a920684a5ab1013cb73bf8581045820f9fc8130407b8f81475d91ff7704
- d2126d012de7c958b1969b875876ac84871271e8466136ffd14245e0442b6fac
- d7b661754cae77aa3e77c270974a3fd6bda7548d97609ac174a9ca38ee802596
- dc5e8c6488f7d6f4dcfac64f8f0755eb8582df506730a1ced03b7308587cdc41
- f4a07e6dcb49cb1d819c63f17a8250f6260a944e6e9a59e822e6118fb1213031
- ffd45bde777b112206b698947d9d9635e626d0245eb4cfc1a9365edc36614cbe

Wali loader:

- a24759369d794f1e2414749c5c11ca9099a094637b6d0b7dbde557b2357c9fcd
- b55b40c537ca859590433cbe62ade84276f3f90a037d408d5ec54e8a63c4ab31
- c48a2077e7d0b447abddebe5e9f7ae9f715d190603f6c35683fff31972cf04a8
- 725dedcd1653f0d11f502fe8fdf93d712682f77b2a0abe1962928c5333e58cae
- cfcbe396dc19cb9477d840e8ad4de511ddadda267e039648693e7173b20286b1

C2 (compromised web sites) of wali:

- hXXp://******essel[.]com/mt/php/tmpl/missing.php
- hXXp://******essel[.]com/mt/mt-static/images/comment/s.php
- hXXp://******hi[.]com/da******/hinshu/ki******/ki******.php
- hXXp://******an[.]jp/_module/menu/menug/index.php
- hXXp://******etop.co[.]jp/includes/firebug/index.php
- hXXp://******etop.co[.]jp/phpmyadmin/themes/pmahomme/sprites.html
- hXXp://******usai[.]com/ex-engine/modules/comment/queries/deleteComment.php
- hXXp://******1cs[.]net/zy/images/patterns/preview/deleteComments.php
- hXXp://******1cs[.]net/zy/images/colorpicker/s.php

Filename (over 50MB size):

- srvhost.exe
- propsyse.exe
- perfcore.exe
- oldb32.exe
- oledb32.exe
- javaup.exe
