Covering the global threat landscape

NOT A GAME MAKER

Raul Alvarez
Fortinet, Canada

Gamker is an information-stealing trojan. It uses simple decryption, then drops a copy of itself using a random filename and injects itself into a different process. It also exhibits other common trojan behaviours.

In this article, we will look into its code injection routine and at the twists in its API-hooking routine.

DECRYPTION #1 AND RESOLVING APIs

In preparation for carrying out its malicious functionalities, Gamker decrypts its code by using a simple XOR instruction with a fixed key, 0x5A8E.

After decrypting its code, the malware parses the PEB (Process Environment Block) to get the ImageBase of kernel32.dll.

This is followed by parsing the PE header of kernel32.dll in memory to get the address of the export table. The API names found in the export table are checked against the 'GetProcAddress' string. A simple byte-by-byte comparison is used to look for an exact string match. Once the 'GetProcAddress' string is found, the equivalent API address is taken using the index of the API name.

Next, the following APIs are resolved using the GetProcAddress API: LoadLibraryA, VirtualAlloc, VirtualFree, VirtualProtect, ExitThread and GetModuleHandleExW.

DECRYPTION #2 AND SELF CODE INJECTION

After resolving the aforementioned APIs, the malware decrypts a few more blocks of code using a XOR instruction with a static key, 0x4FA1, followed by a ROR (rotate right) instruction.

On executing the second decryption routine, a new executable image emerges in memory, complete with an MZ/PE header and the rest of the malware body.

Within the new executable image, the malware parses the PE header to determine its size. This is followed by changing the memory protection of the current module to PAGE_EXECUTE_READWRITE, using the VirtualProtect API. (Remember that the current module is the original malware loaded in memory.)

In further preparation, the current module's memory space is cleared by filling it with zeroes (null bytes). Then the MZ/PE header of the new executable image is copied to the location of the current module.

Afterwards, Gamker copies each and every section of the new executable image to the area of the current module, by calculating the exact location at which each section should be placed.

After every byte in every section has been copied, the current module's memory space contains the properly aligned version of the new executable image.

This method of self code injection is an effective measure to avoid being detected by anti-malware applications, as there is no physical file related to the actual malware, which means it cannot be detected on disk.

FIXING THE IAT

The IAT (Import Address Table) of the new module that has just been loaded does not contain the correct API addresses. To rectify this, the malware parses the PE header in order to locate the import table.

From within the import table, the malware loads every library it finds, using the LoadLibraryA API. This is followed by traversing all API names related to each library, and resolving their equivalent API addresses using the GetProcAddress API. Every resolved address replaces content in the IAT.

When the above routine has been completed, the IAT is left filled with the properly resolved API addresses. At this point, the new version of Gamker is ready to execute a new set of functionalities.

API HOOKING

Once the new version of Gamker is operational, it performs a simple hooking technique with a simple twist.

A common API-hooking technique is to replace a few bytes from the start of the function with a call or jump to the malware code, while the original API code is saved in the malware's memory space to be used later.

The same technique is used by Gamker, with a little twist.

One of the initial routines performed by the new Gamker is to hook the NtQueryInformationProcess API.

At first, it gets the address of the NtQueryInformationProcess API within ntdll.dll by using a combination of the GetModuleHandleA and GetProcAddress APIs.

This is followed by copying the first two instructions of the NtQueryInformationProcess API to the newly allocated memory (0x8F0000), which was created earlier by calling the VirtualAlloc API. Then it changes the memory protection to PAGE_EXECUTE_READWRITE using the VirtualProtect API.

Meanwhile, the protection of the NtQueryInformationProcess API in memory is changed to PAGE_EXECUTE_WRITECOPY.

Afterwards, an encryption key is generated from the result of the XORed value of the GetTickCount API output and the RDTSC (Read Time-Stamp Counter) value.

Another block of virtual memory is allocated at address 0x900000, which also has its protection changed to PAGE_EXECUTE_READWRITE. Using a series of InterlockedExchange API calls, Gamker now fills the newly allocated memory with the following values:

1. [00900000]: 0x90 00 00 04 (assuming the newly allocated memory starts at 0x900000)
2. [00900004]: 0x68 (a PUSH instruction)
3. DWORD value (XORed value of the 'address of the third instruction from the NtQueryInformationProcess API' and 'the encryption key')
4. [00900009]: 0x9c (PUSHFD)
5. [0090000A]: 0x81 74 24 04 (XOR DWORD PTR [ESP+4])
6. Encryption key (DWORD)
7. 0x9D (POPFD)
8. 0xC2 04 00 (RETN 4)

Executing the above generated instructions simply executes the third instruction of the NtQueryInformationProcess API.

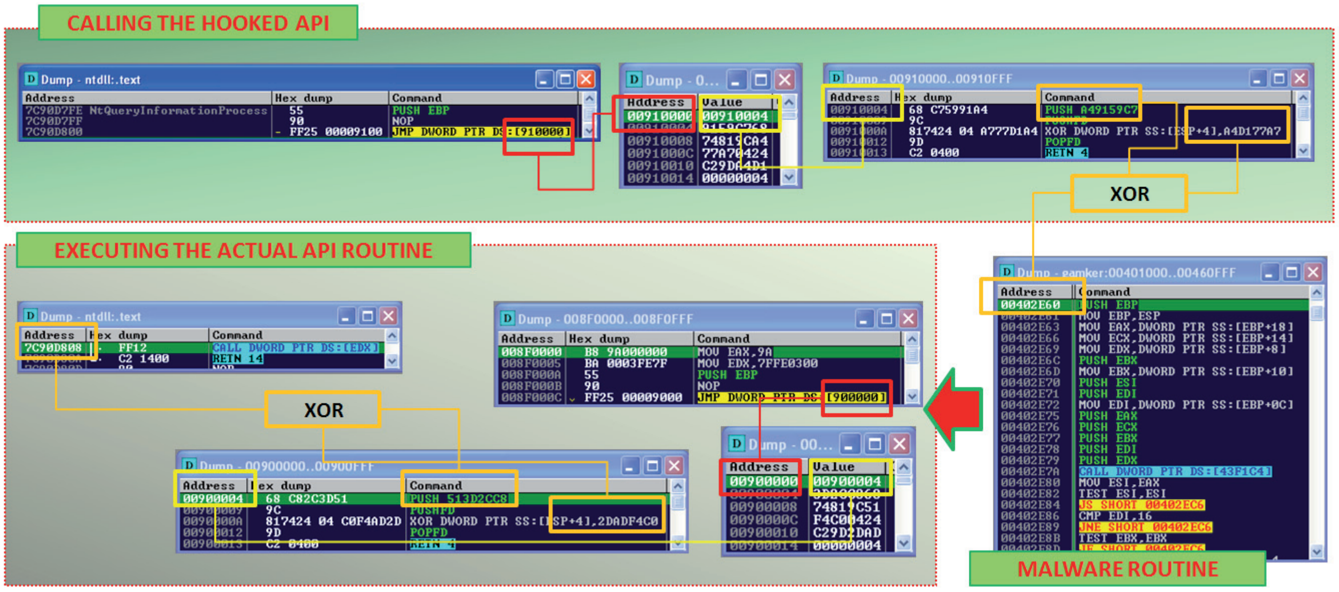
Recall that the first two instructions of the NtQueryInformationProcess API were copied to the allocated memory at 0x8F0000. To connect the generated third instruction, Gamker adds the following bytes using an uncommon assembly instruction: 'LOCK CMPXCHG8B QWORD PTR [ESI]', where ESI is 0x008F000A, ECX is 0x00900000, and EBX is 0x25FF9055. (CMPXCHG8B indicates an eight-byte compare-exchange instruction.)

The result of the instruction can be found at the location 0x8F000A:

1. [008F0000]: first instruction of the NtQueryInformationProcess API
2. [008F0005]: second instruction of the NtQueryInformationProcess API
3. [008F000A]: 0x55 (PUSH EBP)
4. [008F000B]: 0x90 (NOP)
5. [008F000C]: 0xFF 25 00 00 90 00 (JMP DWORD PTR [00900000]) (executes the third instruction)

The above set of instructions mimics the NtQueryInformationProcess API call.

So far, the malware has just generated a copy of the NtQueryInformationProcess API code. The next routine performs the actual hooking by replacing some of the API's code.



```
Dump - ntdll:.text
Address                            Hex dump          Command
7C90D7FE NtQueryInformationProcess   55              PUSH EBP
7C90D7FF                             90              NOP
7C90D800                           - FF25 00009100   JMP DWORD PTR DS:[910000]
```

```
Dump - 0...
Address   Value
00910000  00910004
00910008  74819CA4
0091000C  77A70424
00910010  C29DA4D1
00910014  00000004
```

```
Dump - 00910000..00910FFF
Address   Hex dump              Command
00910004  68 C75991A4           PUSH A49159C7
          9C                    PUSHFD
0091000A  817424 04 A777D1A4    XOR DWORD PTR SS:[ESP+4],A4D177A7
00910012  9D                    POPFD
00910013  C2 0400               RETN 4
```

```
Dump - ntdll:.text
Address   Hex dump    Command
7C90D808  FF12        CALL DWORD PTR DS:[EDX]
          C2 1400     RETN 14
```

```
Dump - 008F0000..008F0FFF
Address   Hex dump        Command
008F0000  B8 9A000000     MOV EAX,9A
008F0005  BA 0003FE7F     MOV EDX,7FFE0300
008F000A  55              PUSH EBP
008F000B  90              NOP
008F000C  FF25 00009000   JMP DWORD PTR DS:[900000]
```

```
Dump - 00900000..00900FFF
Address   Hex dump              Command
00900004  68 C82C3D51           PUSH 513D2CC8
          9C                    PUSHFD
0090000A  817424 04 C0F4AD2D    XOR DWORD PTR SS:[ESP+4],2DADF4C0
00900012  9D                    POPFD
00900013  C2 0400               RETN 4
```

```
Dump - 00...
Address   Value
00900000  00900004
00900008  74819C51
0090000C  F4C00424
00900010  C29D2DAD
00900014  00000004
```

```
Dump - gamker:00401000..00460FFF
Address   Command
00402E60  PUSH EBP
          MOV EBP,ESP
00402E63  MOV EAX,DWORD PTR SS:[EBP+18]
00402E66  MOV ECX,DWORD PTR SS:[EBP+14]
00402E69  MOV EDX,DWORD PTR SS:[EBP+8]
00402E6C  PUSH EBX
00402E6D  MOV EBX,DWORD PTR SS:[EBP+10]
00402E70  PUSH ESI
00402E71  PUSH EDI
00402E72  MOV EDI,DWORD PTR SS:[EBP+0C]
00402E75  PUSH EAX
00402E76  PUSH ECX
00402E77  PUSH EBX
00402E78  PUSH EDI
00402E79  PUSH EDX
00402E7A  CALL DWORD PTR DS:[43F1C4]
00402E80  MOV ESI,EAX
00402E82  TEST ESI,ESI
00402E84  JS SHORT 00402EC6
00402E86  CMP EDI,16
00402E89  JNE SHORT 00402EC6
00402E8B  TEST EBX,EBX
```

Figure 1: The execution path when a hooked API is called.

The routine is similar to the one described above, with some variation in the values. The encryption key is different due to a different value of another call to the GetTickCount API and RDTSC.

Then another block of virtual memory (at location 0x00910000) is allocated. The routine performed to allocate this virtual memory is similar to that used for the memory at 0x900000, except that the encryption key is different, and the address points to the hook function.

The actual hooking happens by executing another 'LOCK CMPXCHG8B QWORD PTR [ESI]' instruction. This time, ESI is 0x7C90D7FE (the address of the NtQueryInformationProcess API), ECX is 0x00910000, and EBX is still 0x25FF9055.

After executing this instruction, the first few bytes of the hooked NtQueryInformationProcess API becomes a jump to the hook function.

CODE INJECTION

After hooking the NtQueryInformationProcess API, the malware performs a few common malware routines such as creating a mutex, dropping a new version of the malware, and decrypting a few more bytes.

Next, we will look into Gamker's code injection routine and at how it hooks a few more APIs.

To make sure that the malware is not running within the context of explorer.exe, it calls the GetCurrentProcessId API, and compares the PID of the current process against that of explorer.exe.

If it is not running in the context of explorer.exe, it will proceed with the code injection routine, otherwise it exits the current routine and performs the rest of its malicious functionalities.

The malware then takes a snapshot of the explorer.exe process by calling the CreateToolhelp32Snapshot API with parameters: TH32CS_SNAPMODULE(dwFlags) and a specific PID (explorer.exe's). This snapshot includes all modules found within the explorer.exe process.

This is followed by enumerating the list of module names found in explorer.exe's process using a combination of the Module32First and Module32Next APIs. The malware looks for the kernel32 module by checking whether each module's name has a substring of 'kernel32' and '.dll', using a subsequent call to the StrStrIA API. Gamker did this to make sure that kernel32.dll is available once the malware code is injected into the explorer.exe process.

This is followed by opening the explorer.exe process by calling the OpenProcess API with access parameter equal to PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE | PROCESS_QUERY_INFORMATION | SYNCHRONIZE.

Then, another check is performed to see whether the malware is running in a 64-bit system, using the IsWow64Process API.

Meanwhile, Gamker modifies some of the header values of the new executable image. (Recall that the image was decoded from the decrypted block of bytes earlier.) The size (0x5CA00) of the image is placed in the CheckSum field of the PE header.

This is followed by calling the GetTickCount API, where the least significant WORD value of the result is saved for later use. There follows another call to the GetTickCount API – this time, the DWORD result is used to overwrite the start of the PE header. Overwriting the PE header corrupts the executable image, but perhaps this is a trick to avoid being scanned by anti-virus memory scanners.

Afterwards, Gamker allocates 764,416 bytes (0xBAA00) of remote virtual memory within the explorer.exe process using the VirtualAllocEx API. Then a series of calls to the WriteProcessMemory API is performed to copy the new executable image to the remotely allocated virtual memory in the explorer.exe process.

After copying the malware code, the memory protection of the remote virtual memory is changed to PAGE_EXECUTE_READWRITE using the VirtualProtect API. To make sure that all bytes have been copied to the explorer.exe process, the malware calls the FlushInstructionCache API.

Finally, to execute the remote malware code in explorer.exe, the malware calls the CreateRemoteThread API.

After executing the remote thread, the malware performs the rest of its malicious functionalities, which are not covered in this article. However, we will look into one of the main functions of the malware that is only performed within the context of the explorer.exe process.

API HOOKING WITHIN EXPLORER

Once the malware realizes that it is running within the context of the explorer.exe process, it will hook the rest of the APIs that it needs to monitor the host system.

Gamker uses the same API-hooking technique as discussed earlier.

In a subsequent execution of the hooking routine, the following APIs are hooked in series in the following order: NtQueryInformationProcess, SendInput, TranslateMessage, GetMessageA, GetMessageW, getaddrinfo, gethostbyname, CryptEncrypt, send, WSASend, WSARecv, recv, HttpSendRequestA, HttpSendRequestW, HttpSendRequestExA, HttpSendRequestExW, InternetQueryDataAvailable, InternetReadFile, InternetReadFileExA, InternetWriteFileExA, InternetCloseHandle, CreateFileW, GetWindowTextA and CreateDialogParamW.

These are the observed APIs being hooked during the execution of the malware in the context of the explorer.exe process.

WRAP UP

For the sake of brevity, some of Gamker's algorithms have not been discussed in detail here. However, its API-hooking routine is worthy of an in-depth look. Its use of instructions such as InterlockedExchange and CMPXCHG8B instead of simple MOV instructions is a crafty technique – and adding simple obfuscation is cunning. It might not hide the fact that an API is already hooked, but the malware hopes that it will be missed.

Gamker has lots of malevolent functionalities and algorithms that are not covered in this article. Some of them bear similarities to other malware, and perhaps we will discuss them in the near future, in the context of a different piece of malware.

Editor: Martijn Grooten

Chief of Operations: John Hawes

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Lewis Jones

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

Consultant Technical Editor: Dr Morton Swimmer

© 2015 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England

Tel: +44 (0)1235 555139 Fax: +44 (0)1865 543153

Email: editorial@virusbtn.com Web: http://www.virusbtn.com/