

## Virus Bulletin :: Andromeda 2.7 features

By Suweera De SouzaFortinet, CanadaNeo TanFortinet, CanadaEditor: Helen Martin

Archived: 2026-04-05 18:07:14 UTC

2013-08-01

### Abstract

A new version of the Andromeda bot was recently spotted in the wild with strengthened self-defence mechanisms and novel methods for keeping its process hidden and running persistently. Moreover, its communication data structure and encryption scheme have changed, rendering previous Andromeda IPS/IDS signatures useless. Suweera De Souza and Neo Tan take a detailed look at Andromeda 2.7.

*Copyright © 2013 Virus Bulletin*

---

Recently, we found a new version of the Andromeda bot in the wild. This version has strengthened its self-defence mechanisms by utilizing more anti-debug/anti VM tricks than its predecessors. It also employs some novel methods for trying to keep its process hidden and running persistently. Moreover, its communication data structure and encryption scheme have changed, rendering the old Andromeda IPS/IDS signatures useless.

In this article, we will look at the following:

- Its unpacking routine
- Its anti-debug/anti VM tricks
- Its malicious code injection routine
- The interaction between its twin injected malicious processes
- Its communication protocol, encryption algorithm and command control.

### Overview of unpacking routine

The sample we analysed is firstly packed with UPX. However, once unpacked, the code inside is another custom packer. This custom packer creates dynamic memory and decrypts code into this memory ([Figure 1](#)). It jumps to a lot of addresses by pushing the offset onto the stack and then returning to it. The code in memory calls VirtualAlloc three times. The first allocated memory is used for storing bytes copied from the original file. Those bytes are then copied over to the third allocated memory where they are rearranged by swapping bytes (using the algorithm shown in [Figure 2](#)). Finally, the partially decrypted bytes are copied to the second allocated memory, where the data is decompressed using the aPLib decompression library. The result is a PE file which is then written over the original file image, and the anti-debugging tricks are carried out from here. [Figure 1](#) gives an overview of the unpacking routine.

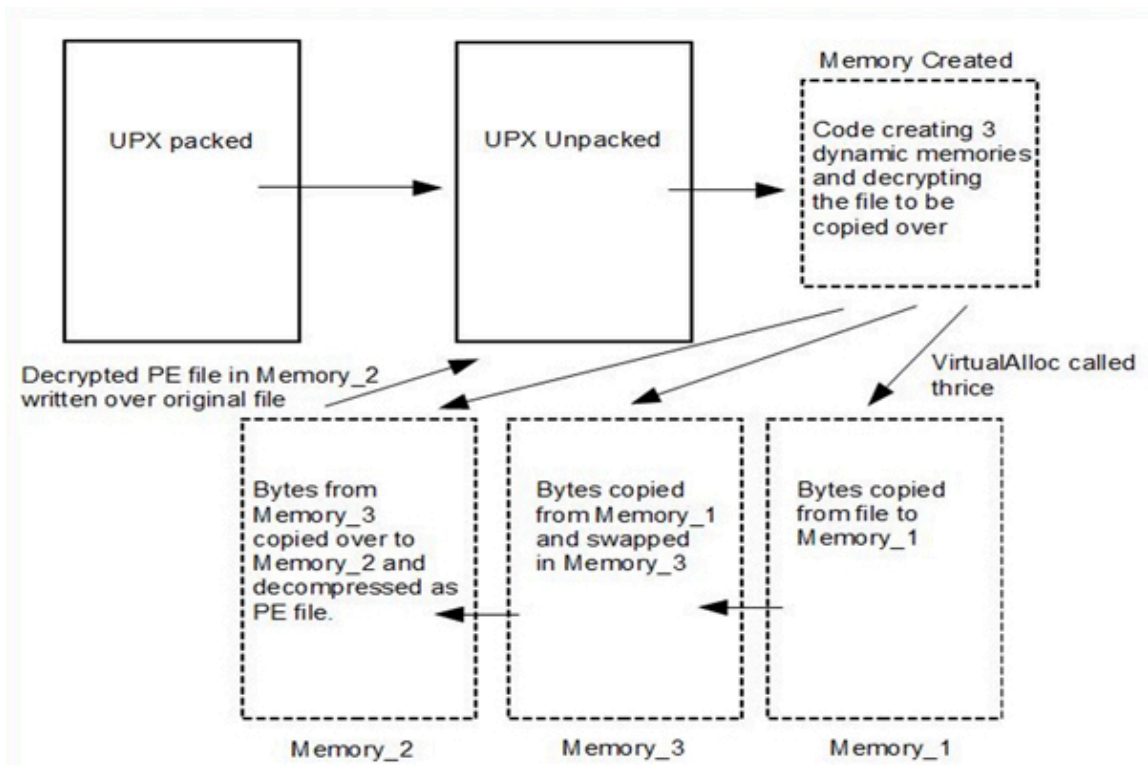


Figure 1. The unpacking process.

```

divisor = 0x134239 + (0x75BCD15 * size) //the decimal of 0x75BCD15 is 123456789
//0x134239 is a fixed constant passed
//to the algorithm

while (size != 0){
    divisor -= 0x75BCD15;
    remainder = divisor % size;
    --size;
    swap_value_1 = data[size]; //the byte swapped starts from the end
    swap_value_2 = data[remainder]; //of the data
    data[size] = swap_value_2;
    data[remainder] = swap_value_1;
}
    
```

Figure 2. Algorithm showing how the bytes were swapped.

## The way to the real routine

This version of Andromeda employs many anti debug/anti VM tricks, which result in the bot switching to a pre-set fake routine in order to prevent it from running in the VM environment, being debugged or monitored. The purpose is obvious: to prevent analysts from being able to access the real malicious routine. In the following sections, we'll take a detailed look at these defence mechanisms.

### Anti-API hook

The sample allocates another section of memory for its anti API hooking technique. The technique consists of storing the first instruction of the API to memory, followed by a jump to its second instruction in the DLL.

For example, in [Figure 3](#), memory location 0x7FF9045E stores the location of memory 0x7FF80060, which is where the first instruction of the API ntdll.RtlAllocateHeap is stored, followed by a jump to the second instruction in the DLL.

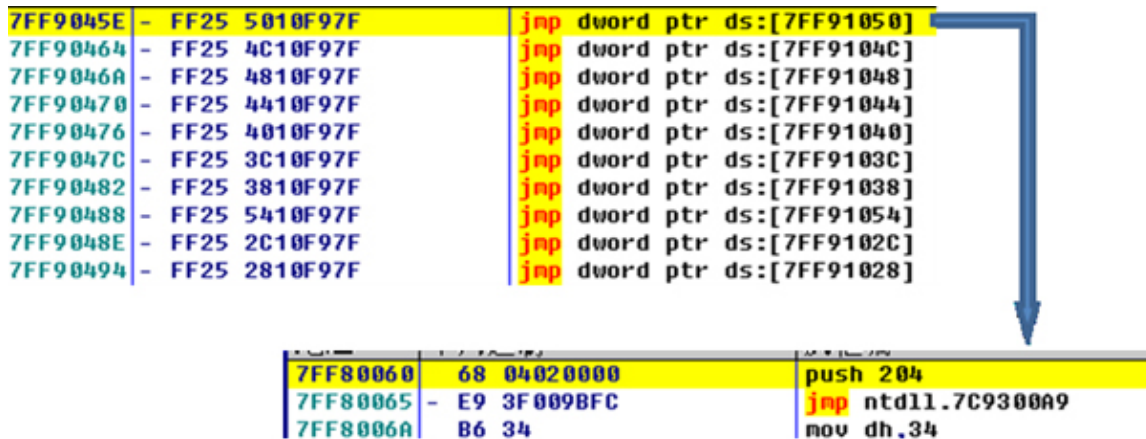


Figure 3. Anti-API hooking.

### Customized exception handler

A pointer to a handler function is passed to the SetUnhandledExceptionFilter API. The handler is called when an access violation error is intentionally created by the sample when it tries to write into the file's PE header. The code in the handler is only executed if the process is not being debugged.

This function ([Figure 4](#)) gets the pExceptionPointers >ContextRecord (the second DWORD of arg\_0) in order to set the location of the real payload (sub\_401EA5) to the EIP (ebx+0B8h) upon return. It also gets the ESP (ebx+0C4h) and then sets the two arguments which will be passed to the payload function: arg0 to dword\_402058 and arg1 to sub\_401AA2. Dword\_402058 points to the encrypted code and sub\_401AA2 points to another decryption routine which will be injected by the code decrypted from dword\_402058.

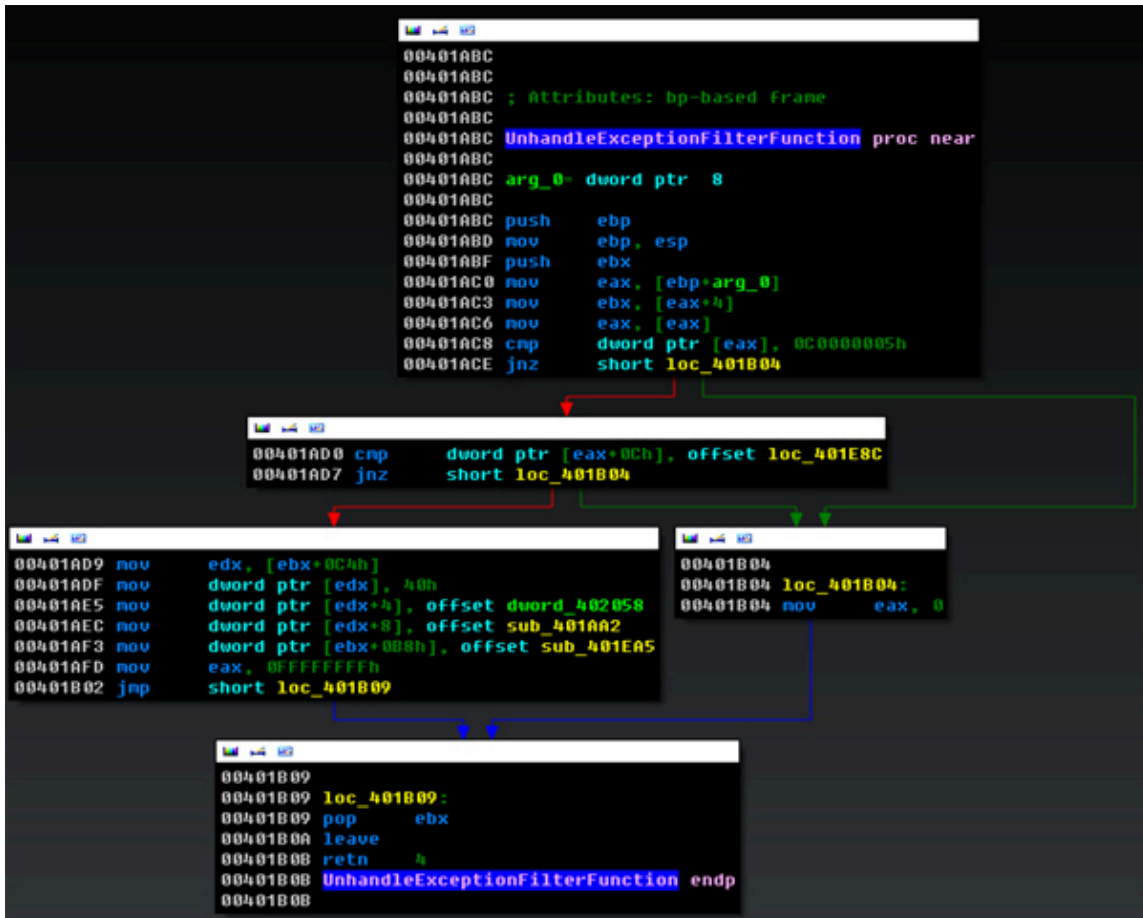


Figure 4. UnhandledExceptionFilter function.

(Click [here](#) to view a larger version of Figure 4.)

### Anti-VM and anti-forensics

The GetVolumeInformationA API is called on drive C:\ to get the name of the drive. Then the bot calculates the CRC32 hash value of the name (Figure 5). If the hash value of the drive name matches 0x20C7DD84, it will bypass all the anti-debugging and anti-VM checks and invoke the exception directly. When the CRC32 hash is reversed, one possible result is 'BVabi'. This could be the name of the author's C drive, so that he/she could skip all the trouble when debugging his/her own program.

```

lea    eax, [ebp-278h] ; lpVolumeNameBuffer
push   eax
call   crc32_hash
cmp    eax, 20C7DD84h ; hash value of BVabi
jz     bypass anti debugging
    
```

Figure 5. Checking if the drive name's CRC32 value is 0x20C7DD84.

If the hash value of the drive name doesn't match, the following anti-debug/anti-VM tricks are employed:

1. Iterating through process names and computing their CRC32 hash values: if a hash value matches any of those on a list of hash values of VM processes (Figure 6) and forensics tools (regmon.exe, filemon.exe, etc.), this

indicates that the debugging process is inside a sandbox environment and being monitored.

```

cmp     eax, 99DD4432h
jz      trap_path
cmp     eax, 2D859DB4h
jz      trap_path
cmp     eax, 64340DCEh ; VBoxService.exe
jz      trap_path
cmp     eax, 63C54474h ; VBoxTray.exe
jz      trap_path

```

**Figure 6. Matching the process with CRC32 hash values.**

- Trying to load the libraries guard32.dll and sbiedll.dll, which belong to *Comodo* and *Sandboxie* respectively. If the libraries can be loaded successfully, this indicates that the debugging process is inside a sandbox environment.
- Querying for a value in the system\currentcontrolset\services\disk\enum registry to search for the presence of any virtual machine ([Figure 7](#)).

```

cmp     dword ptr [ebp-170h], 'awmv'
jz      short trap_path
cmp     dword ptr [ebp-170h], 'xobv'
jz      short trap_path
cmp     dword ptr [ebp-170h], 'umeq'
jz      short trap_path

```

**Figure 7. Querying for virtual machines**

- Calling the opcode rdtsc, which returns the processor time stamp. When first called, it saves it in edx, and the second time it saves it in eax. The registers are subtracted and if the result of the tickcount is more than 0x200h, this indicates that the process is being debugged.

If the bot does detect the presence of either a debugger or a virtual machine, it decrypts the dummy code. This code copies itself under %alluserprofiles% as svchost.exe with hidden system file attributes. It then writes itself in the registry HKLM\Software\Microsoft\Windows\CurrentVersion\Run as SunJavaUpdateSched. A socket is then created to listen actively, but no connection has been made previously.

### Data structure of encrypted routine

As mentioned, the bot will decrypt the code as the next routine, whether dummy code or a useful routine. The encrypted code of the file is contained within a specific structure that the file uses when carrying out its decryption routine. In this sample, there are three sets of encrypted code which represent three different routines. One routine contains dummy code that is decrypted only when the sample is being debugged or run in a virtual machine. The second routine contains code that injects itself into another process, whereon the third routine is decrypted in that process. The data structure is shown in [Figure 8](#).

```

typedef struct _DATA_STRUCTURE{
    DWORD Key[4];           //0x00 Key used in RC4
    DWORD Code_Length;     //0x10 length of encrypted code
    DWORD Hash;           //0x14 crc32 hash of encrypted code
    DWORD Memory_Size;    //0x18 size of memory to allocate
    DWORD Start_Offset;   //0x1C start of code in memory
    DWORD Lib_Offset;     //0x20 location of DLL and API
                        //hash values
    DWORD Memory_API_Hook; //0x24 size of memory for
                        //anti-API hooking
}

```

**Figure 8. Data structure used by the bot.**

The encrypted data, which is located at 0x28h after the structure, is decrypted using RC4. The key used is a fixed length of 0x10h and is located at the beginning of the structure. The decrypted code is further decompressed into allocated memory using the aPLib decompression library.

## Twin malicious injected processes

The bot will inject its core code into two processes after successfully bypassing all the anti-debug/anti-VM tricks. First, let's see how the malicious code is injected into processes before we shed light on how the two injected processes interact with each other.

### Code injection routine

The bot calls the GetVolumeInformation API on C:\, to get the VolumeSerialNumber. It then checks whether the environment variable 'svch' has already been created [1]. If it has, then it will inject itself into svchost.exe. If the environment variable is not present, it will set the environment variable 'src' to point to its own file path and then inject into msixec.exe. This suggests that the bot injects its code into two different processes at different instances. We shall see why in the next section.

It then gets the Windows directory. Before injection, the bot needs to find the location of these files (svchost.exe, msixec.exe) in the Windows directory. Thus, it calls ZwQueryInformationProcess and accordingly concatenates the process name with \System32 for 32-bit and \SysWOW64 for 64-bit systems.

The injection process involves several steps:

1. As with the previous versions, the malware calls CreateFile to get the handle of the file it wants to inject. It then gets its section handle by calling ZwCreateSection, which is used by ZwMapViewOfSection to get the image of the file in memory. From this image, it extracts the size of image and the address of the entry point from the PE header.
2. A memory address with the same size as that of the image of the file it wants to inject is created with page\_execute\_readwrite access. Then the image of the file is copied over to this memory address.
3. Another memory address is created with the same size as that of the image of the original bot file, also with page\_execute\_readwrite access. The original file is then copied over to this new memory address.

4. A suspended process of the file to be injected is created. The memory address containing the original file is unmapped. ZwMapViewOfSection is called with the bot's file handle and the process handle (acquired from creating the suspended file process). So now the injected file's process handle has a map view of the botnet file. Before it calls ResumeThread to resume the process, it changes the entry point of the injected file to point to its code, which it has modified as follows:

```
push <address of botnet code to jump to>
ret
```

### Twin process interaction

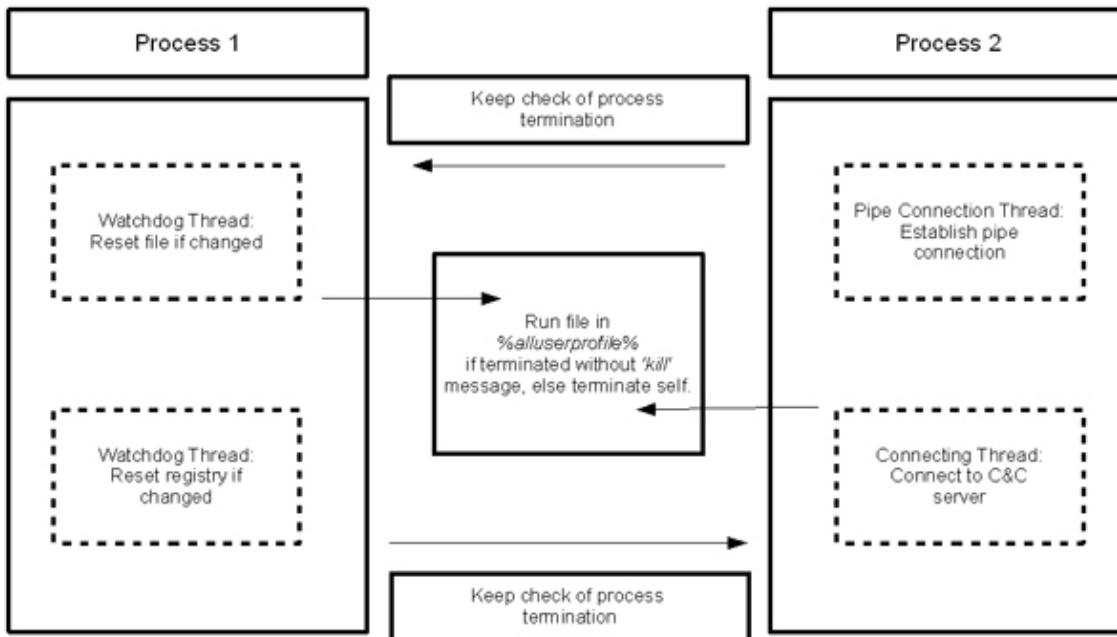
The code that is injected into the process decrypts more code into memory using the methods described in the previous section. This final decrypted code is the commencement of the botnet's payload. In this version, Andromeda displays some new techniques in its execution.

First, it modifies the registry entry

HKLM\system\currentcontrolset\services\sharedaccess\parameters\firewallpolicy\standardprofile\authorizedapplications\list to the value of %s\*:Generic Host Process, which points to the path of the current process. This is done to allow the process to bypass the firewall.

Next, it tries to determine whether the environment variable 'svch' has been set. If it has, it means that another instance of the file has been run. If it has not been set, then the malware has yet to inject itself into the other process.

The creation of two processes is important for the bot. One process is used to make sure that the copy of the bot which will be created in %alluserprofile% is always present and that the registry entries have not been modified. The second process is used for connecting to the C&C server and executing instructions based on the messages received. Additionally, the two processes communicate with each other through an instance of creating a pipe connection. It is this connection that enables either process to check that both instances of the bot are always running or to terminate the processes in the event of an update or installation. The analysis of this part has been divided into Process 1 and Process 2, so as to better understand the communication between the two processes ([Figure 9](#)).



**Figure 9. The flow of communication between the two bot processes.**

(Click [here](#) to view a larger version of Figure 9.)

**Process 1 (installation routine and watchdogs)**

This part of code is executed when the environment variable ‘svch’ has not been found. The bot tries to connect to the pipe name, which is ‘kill’ xor’ed by the VolumeSerialNumber. If it can connect, then the bot terminates the other process. This thread is created as a check to terminate the other bot process in the event of an installation.

It then tries to get the environment variable ‘src’, which was created before injection. The value contains the path from which the original file was run. It uses this path to create a copy of the original file before deleting it, and saves it in %alluserprofile% with a random filename.

Next, the bot wants to enable the file to autorun, so it saves the path of the file in %alluserprofile% in the registry. At first, it tries to access the subkey \software\microsoft\windows\currentversion\Policies\Explorer\Run in registry HKLM. If it is unsuccessful, it accesses the subkey \software\microsoft\windows nt\currentversion\windows in HKCU. The registry that it accesses successfully is the one that is used throughout for any modifications (explained in pseudo code in [Figure 10](#)). Once it has accessed the registry, it sets the security key of the registry to KEY\_ALL\_ACCESS. The security key is obtained by passing the string ‘D:(A;;KA;;;WD)’ to the ConvertStringSecurityDescriptorToSecurityDescriptorA API, which converts it to a security key. Once it has set the security key, it saves the path of the new file to the registry under the value of VolumeSerialNumber (for HKLM) or Load (for HKCU). The original file in the old path is deleted and the environment variable ‘src’ is set, pointing to 0.

```

If 'HKLM\software\microsoft\windows\currentversion\Policies\Explorer\Run' is
accessible
- set the Security Key to KEY_ALL_ACCESS
- set the VolumeSerialNumber as key to the file created in %alluserprofile%
Else access 'HKCU\software\microsoft\windows nt\currentversion\windows'
- set the Security key to KEY_ALL_ACCESS
- set the key 'Load' to the file created in %alluserprofile%
    
```

**Figure 10. Pseudo code of registry chosen.**

After this, the bot creates two watchdog threads which are primarily used to keep re-setting the file and the registry entries if they have been modified. The first thread checks if any modification has been made to the filename in %alluserprofile%, or if it has been deleted. Then it creates the file again with the same filename. It accomplishes this by first saving the file to the buffer by calling ReadFile. Then it calls the FindFirstChangeNotificationW API, whose handle will retrieve the changes made to the filename. If the handle is 0xFFFFFFFF, then no changes have been made, and it enters a loop. If a change has been notified, then it creates the file again with the same filename, and writes the contents of the file back from the buffer created by ReadFile.

The second thread checks if any changes have been made to a value in the registry. If a change has been made, then it resets the registry security key and the value in the registry. Notification of changes made to the registry is set by calling RegNotifyChangeKeyValue.

The bot then creates two environment variables – ‘ppid’, pointing to its process ID, and ‘svch’ with the value of 1. It then runs the file that has been created in %alluserprofile%. After running the file, it tries to connect to the pipe ‘kill’ xor’ed by the VolumeSerialNumber. Since the value of svch has been set to 1, the second process will create a thread that creates the named pipe connection and executes a second thread to connect to the C&C server. When the first process can connect successfully to the pipe connection created by the second process, it resets the environment variables ‘svch’ and ‘ppid’ to 0.

### **Process 2 (core routine)**

When the bot is run in another process, it sets the environment variable ‘svch’ to 0. A thread is created that creates a named pipe. If a connection is established, the thread reads the bytes that are written from the other process. If the message is ‘kill’ xor’ed by VolumeSerialNumber, the process terminates. However, if the message is ‘gpid’, then it sends its current process ID to the first process. This information is used by the old process to access information about the new process when the new process terminates. When the new process terminates, the old process checks the handle of the process. If the message is ‘kill’ xor’ed by VolumeSerialNumber, then the old process terminates. This check is made when the bot wants to update itself and hence has to make sure that the watchdog threads have been terminated. Otherwise, the old process terminates the new process and runs the file in %alluserprofile% again.

After the new process has created its thread to connect to the C&C server, it will get the ‘ppid’ environment variable. This variable contains the process ID of the old process. Like the old process, it uses this information to access when the old process terminates. And if the message is ‘kill’ xor’ed by VolumeSerialNumber, then the new process terminates. This check is performed when an installation is taking place. Otherwise, the new process runs the file in %alluserprofile% and terminates itself.

[Figure 11](#) shows how the process IDs are used by the processes.

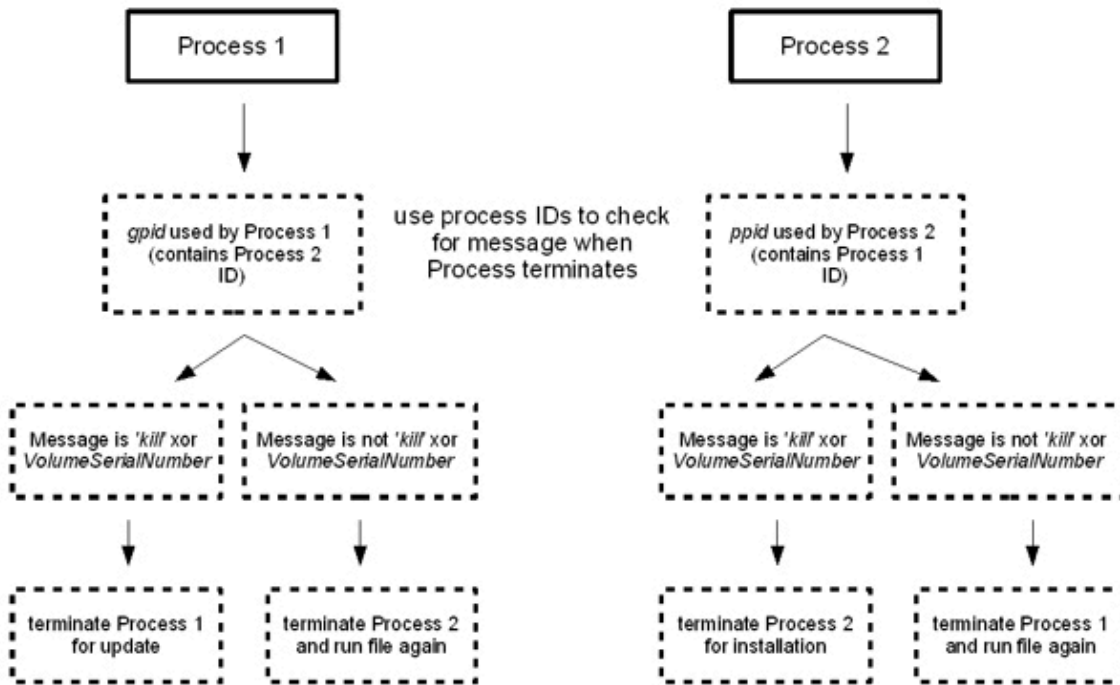


Figure 11. Process IDs used by the processes.

The second thread created by the new process carries out some further code injection. It first resolves winhttp.dll APIs using the anti-API hooking technique and also inline hooks three APIs: ws2\_32.GetAddrInfoW (Figure 12 and Figure 13), ntdll.ZwMapViewOfSection and ntdll.ZwUnmapViewOfSection. The control flow of the APIs is redirected by inserting a jump to the malicious function. Before writing to the API, it calls VirtualProtect. After the bytes have been written, it calls FlushInstructionCache so that the changes take effect immediately.

Address	Hex dump	Disassembly	Comment
71AB2899	GetAddrInfoW	8BFF	MOV EDI,EDI
71AB289B	55		PUSH EBP
71AB289C	8BEC		MOV EBP,ESP
71AB289E	81EC C4000000		SUB ESP,0C4
71AB28A4	A1 5C40AC71		MOV EAX,DWORD PTR DS:[71AC405C]
71AB28A9	8945 FC		MOV DWORD PTR SS:[EBP-4],EAX

Figure 12. Before inline hooking GetAddrInfoW.

Address	Hex dump	Disassembly	Comment
71AB2899	GetAddrInfoW	- E9 02F04E0E	JMP 7FFA10A0
71AB289E	81EC C4000000		SUB ESP,0C4
71AB28A4	A1 5C40AC71		MOV EAX,DWORD PTR DS:[71AC405C]
71AB28A9	8945 FC		MOV DWORD PTR SS:[EBP-4],EAX

Figure 13. After inline hooking GetAddrInfoW.

It then calls QueueUserAPC, which creates an asynchronous procedure call object. This object points to the code which decrypts some encrypted strings using RC4 decryption (Figure 14). These encrypted strings are the domains it intends to connect to. Before each decrypted string, it inserts the DWORD 0x6C727501 xor'ed by VolumeSerialNumber, which is ASCII for URL. This magic DWORD is used when it calls the RtlWalkHeap API to retrieve the domain names from the heap.

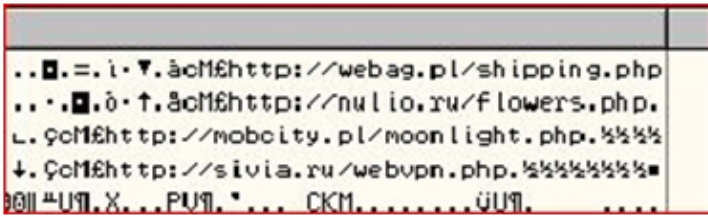
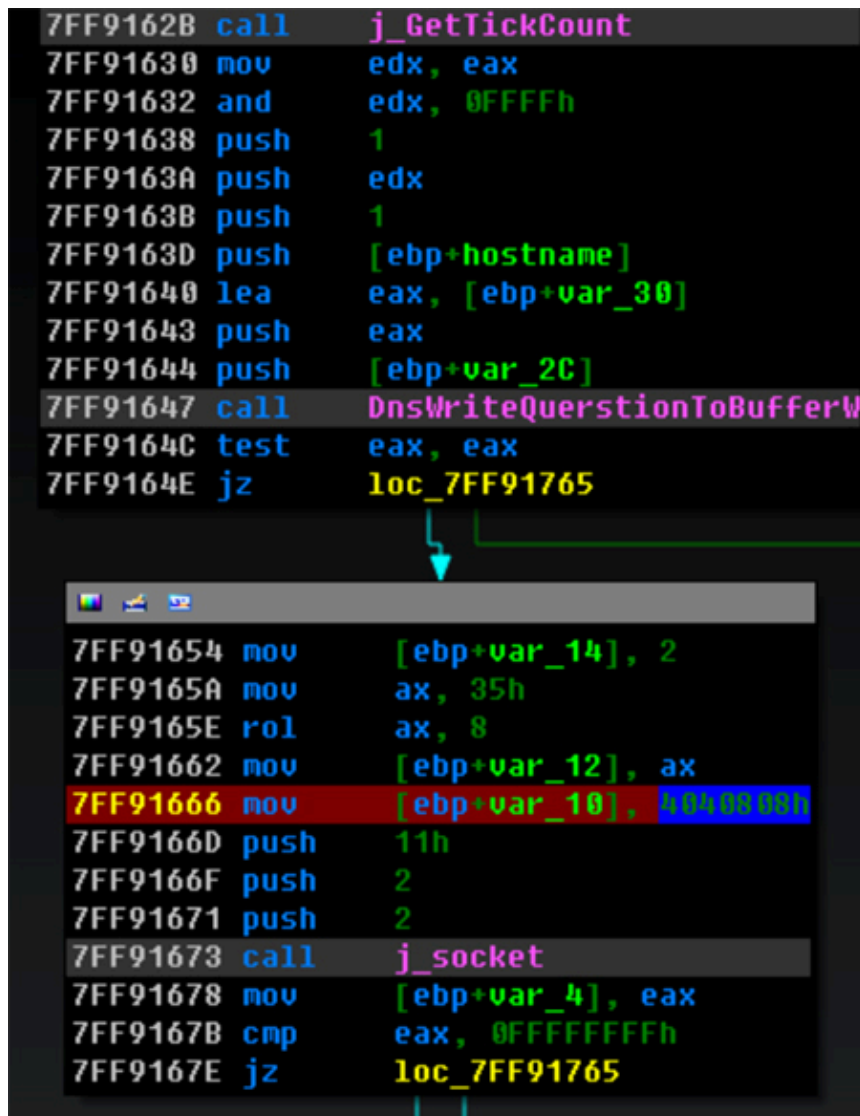


Figure 14. The decrypted domain names (now offline).

## Communication protocol, encryption algorithm

### Create connections

The hooked GetAddrInfoW API performs a DNS query for the input host name from Google DNS server 8.8.4.4 (Figure 15) using a randomly generated query identifier. It then returns the query result or '127.0.0.1' if the DNS query fails. The DNS record received is then used for querying the C&C domain name. It does this to avoid any application level DNS server redirection. The hooked ZwMapViewOfSection and ZwUnmapViewOfSection APIs will be used later to map/unmap the plug in image downloaded from the C&C server.



**Figure 15. Hard-coded Google DNS server IP in the GetAddrInfoW hooked function.**

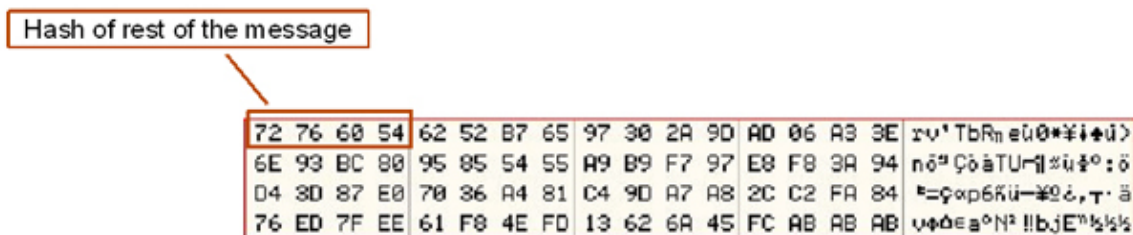
**Communication protocol and encryption algorithm**

Before establishing a connection, the bot prepares the message to be sent to the C&C server. It uses the following format:

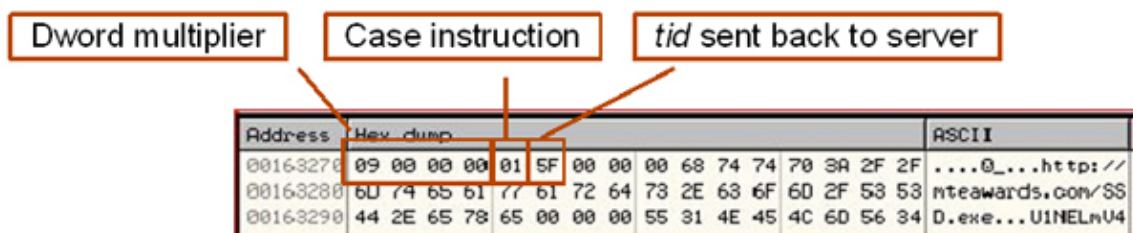
```
id:%lu|bid:%lu|bv:%lu|os:%lu|la:%lu|rg:%lu
```

- id is the VolumeSerialNumber, which is used as an RC4 key to decrypt the message received
- bid is a hard coded DWORD used for the communication
- bv is the version of the botnet (in this case it is 2.7)
- os is the version of the current operating system
- la is the socket name byte swapped
- rg is set to 1 if the process is in the Administrator group, otherwise it is 0 (Figure 16).

This string is encrypted using RC4 with a hard-coded key of length 0x20 and is further encrypted using base64. The message is then sent to the server. Once a message is received, the bot calculates the CRC32 hash of the message without including the first DWORD (Figure 16). If the calculated hash matches the first DWORD, the message is valid. Later it is decrypted using RC4 with the VolumeSerialNumber as the key. After the RC4 decryption the message is in the format gn([base64-encoded string]). This used to be just the base64 encoded string, but for some reason the author decided not to make the server backward compatible with the older bot versions. Then it decodes the base64 string inside the brackets to get the message in plain text (Figure 17).



**Figure 16. First DWORD of message received containing the CRC32 hash value.**



**Figure 17. Message received from the server.**

The first DWORD of the message is used as a multiplier to multiply a value in a fixed offset. The DWORD in that offset is used as an interval to delay calling the thread again to establish another connection. The next byte indicates what

action to carry out – there are seven options:

- Case 1 (download EXE):

Connect to the domain decrypted from the message to download an EXE file. Save the file to the %tmp% location with a random name and run the process.

- Case 2 (load plug-ins):

Connect to the domain decrypted from the message, install and load plug ins. The plug ins are decrypted by RC4 using the same key of length 0x20h.

- Case 3 (update case):

Connect to the domain to get the update EXE file. If a filename of VolumeSerialNumber is present in the registry, then save the PE file to the %tmp% location with a random name; else save it to the current location with the name of the file as VolumeSerialNumber. The file in %tmp% is run, while the current process terminates. It also sends the message 'kill' xor'ed by VolumeSerialNumber to terminate the older process.

- Case 4 (download DLL):

Connect to the domain and save the DLL file to the %alluserprofile% location. The file is saved as a .dat file with a random name and loaded from a specified export function. The registry is modified so it can be auto-loaded by the bot.

- Case 5 (delete DLLs):

Delete and uninstall all the DLLs loaded and installed in Case 4.

- Case 6 (delete plug-ins):

Uninstall all the plug-ins loaded in Case 3.

- Case 7 (uninstall bot):

Suspend all threads and uninstall the bot.

After executing the action based on which instruction it received, another message is sent to the server to notify it that the action has been completed:

```
id:%lu|tid:%lu|res:%lu
```

- id is the VolumeSerialNumber
- tid is the next byte (task id) after the byte displaying the case number in the message received
- res is the result of whether or not the task was carried out successfully.

Once the message has been sent, the thread exits and waits for the delay interval period to pass before it reconnects to the server to receive additional instructions.

## Conclusion

This new version of the Andromeda bot has demonstrated its tenacity by executing code that ensures every instance of its process is kept running and by employing more anti debug/anti VM tricks than its previous version. However, it is still possible to bypass all those tricks once we have complete knowledge of its executing procedures. Moreover, we could easily block its communication data after addressing the decryption performance issue.

## Bibliography

[1] All the environment variables used in this version of Andromeda are encrypted using xor on the VolumeSerialNumber, which the file acquires by calling GetVolumeInformationA on drive C:\. The bot employs this technique as a way of specifying its status in the machine. 'svch' is a flag if the process is injected into svchost.exe; 'src' stores the location of the file; 'ppid' stores the first process ID; 'gpId' stores the second process ID.

[2] Tan, N. Andromeda Botnet. Virus Bulletin, June 2012, pp.5–11.

---

Source: <https://www.virusbulletin.com/virusbulletin/2013/08/andromeda-2-7-features>