SECURITY RESEARCH THREAT LANDSCAPE AND ANALYSIS

Andromeda 2.7 Features

by **Neo Tan (/author/neo-tan)** | April 23, 2014 | Category: Security Research (/category/security-research)

[

This article originally appeared in Virus Bulletin

](http://www.virusbtn.com/virusbulletin/archive/2013/08/vb201308-Andromeda)

Recently, we found a new version of the Andromeda bot in the wild. This version has strengthened its self-defense mechanisms by utilizing more anti-debug/anti-VM tricks its predecessors. It also employs some novel methods for trying to keep its process hidden and running persistently. Moreover, its communication data structure and encr 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 injec 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 ar 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 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 rear 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 decompres 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. Figur gives an overview of the unpacking routine.

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 defense mechanisms.

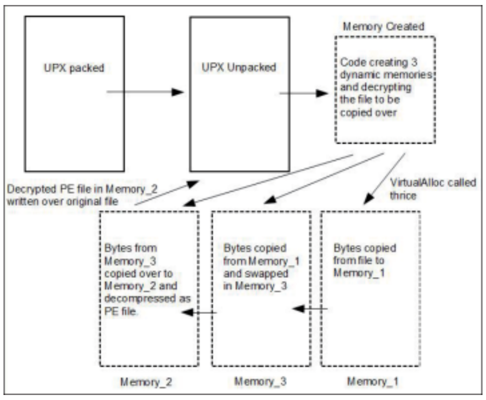

UPX packed
UPX Unpacked
Memory Created
Code creating 3 dynamic memories and decrypting the file to be copied over
Decrypted PE file in Memory_2 written over original file
VirtualAlloc called thrice
Bytes from Memory_3 copied over to Memory_2 and decompressed as PE file
Bytes copied from Memory_1 and swapped in Memory_3
Bytes copied from file to Memory_1
Memory_2
Memory_3
Memory_1

Figure 1: The unpacking process.

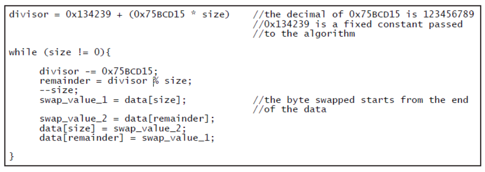
Anti-API hook

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.

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.

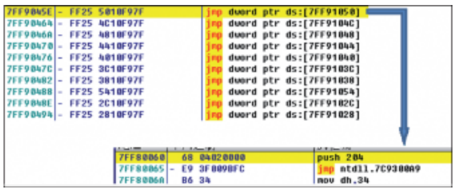


```
divisor = 0x134239 + (0x75BCD15 * size)    //the decimal of 0x75BCD15 is 123456789
                                           //0x134239 is a fixed constant passed
                                           //to the algorithm

while (size != 0){

      divisor -= 0x75BCD15;
      remainder = divisor % size;
      --size;
      swap_value_1 = data[size];           //the byte swapped starts from the end
                                           //of the data
      swap_value_2 = data[remainder];
      data[size] = swap_value_2;
      data[remainder] = swap_value_1;

}
```

Figure 2: Algorithm showing how the bytes were swapped.

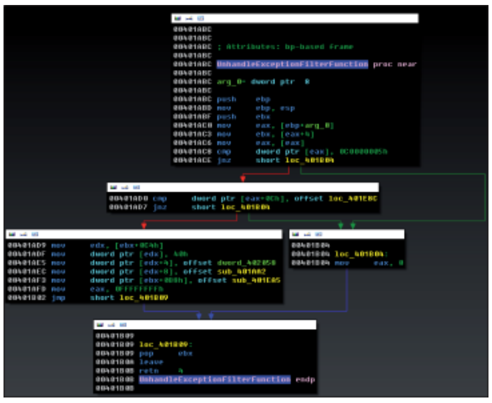
This function (Figure 4) gets the pExceptionPointers->ContextRecord (the seconc DWORD of arg_0) in order tov set the location of the real payload (sub_401EA5) t EIP (ebx+0B8h) upon return.

It also gets the ESP (ebx+0C4h) and then sets the two arguments which will be p 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_402(



```
7FF9045E - FF25 5010F97F    jmp dword ptr ds:[7FF91050]
7FF90464 - FF25 4C10F97F    jmp dword ptr ds:[7FF9104C]
7FF9046A - FF25 4810F97F    jmp dword ptr ds:[7FF91048]
7FF90470 - FF25 4410F97F    jmp dword ptr ds:[7FF91044]
7FF90476 - FF25 4010F97F    jmp dword ptr ds:[7FF91040]
7FF9047C - FF25 3C10F97F    jmp dword ptr ds:[7FF9103C]
7FF90482 - FF25 3810F97F    jmp dword ptr ds:[7FF91038]
7FF90488 - FF25 5410F97F    jmp dword ptr ds:[7FF91054]
7FF9048E - FF25 2C10F97F    jmp dword ptr ds:[7FF9102C]
7FF90494 - FF25 2810F97F    jmp dword ptr ds:[7FF91028]

7FF80060   68 04020000      push 204
7FF80065 - E9 3F0090FC      jmp ntdll.7C9300A9
7FF8006A   B6 34            mov dh,34
```

Figure 3: Anti-API hooking

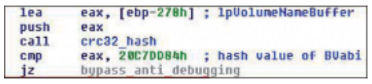


```
00401ABC
00401ABC
00401ABC ; Attributes: bp-based frame
00401ABC
00401ABC UnhandleExceptionFilterFunction proc near
00401ABC
00401ABC arg_0= dword ptr  8
00401ABC
00401ABC push    ebp
00401ABD mov     ebp, esp
00401ABF push    ebx
00401AC0 mov     eax, [ebp+arg_0]
00401AC3 mov     ebx, [eax+4]
00401AC6 mov     eax, [eax]
00401AC8 cmp     dword ptr [eax], 0C0000005h
00401ACE jnz     short loc_401B04

00401AD0 cmp     dword ptr [eax+0Ch], offset loc_401E8C
00401AD7 jnz     short loc_401B04

00401AD9 mov     edx, [ebx+0C4h]
00401ADF mov     dword ptr [edx], 40h
00401AE5 mov     dword ptr [edx+4], offset dword_402058
00401AEC mov     dword ptr [edx+8], offset sub_401AA2
00401AF3 mov     dword ptr [ebx+0B8h], offset sub_401EA5
00401AFD mov     eax, 0FFFFFFFFh
00401B02 jmp     short loc_401B09

00401B04
00401B04 loc_401B04:
00401B04 mov     eax, 0

00401B09
00401B09 loc_401B09:
00401B09 pop     ebx
00401B0A leave
00401B0B retn    4
00401B0B UnhandleExceptionFilterFunction endp
00401B0B
```

Figure 4: UnhandledExceptionFilter function.

Anti-VM and anti-forensics

The GetVolumeInformationA API is called on drive C: to get the name of the drive the bot calculates the CRC32 hash value of the name (Figure 5). If the hash value drive name matches 0x20C7DD84, it will bypass all the anti-debugging and anti-\ 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 could skip all the trouble when debugging his/her own program.



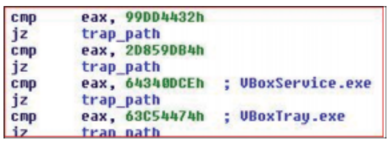
```
lea     eax, [ebp-278h] ; lpVolumeNameBuffer
push    eax
call    crc32_hash
cmp     eax, 20C7DD84h  ; hash value of BVabi
jz      bypass_anti_debugging
```

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. 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) a forensics tools (regmon.exe, filemon.exe, etc.), this indicates that the debugging process is inside a sandbox environment and being monitored.

1. 2.) 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.
2. 3.) Querying for a value in the systemcurrentcontrolsetservicesdiskenum registry to search for the presence of any virtual machine (Figure 7).

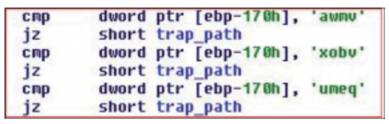


```
cmp     eax, 99DD4432h
jz      trap_path
cmp     eax, 2D859DB4h
jz      trap_path
cmp     eax, 64340DCEh  ; VBoxService.exe
jz      trap_path
cmp     eax, 63C54474h  ; VBoxTray.exe
jz      trap_path
```

Figure 6: Matching the process with CRC32 hash values.

1. 4.) 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 %alluserprofi les% as svchost.exe with hidden system fi le attributes. It then writes itself in the registry HKLMSoftwareMicrosoftWindowsCurrentVersionRun as SunJavaUpdateSched. A socket is then created to listen actively, but no connection has been made previously.



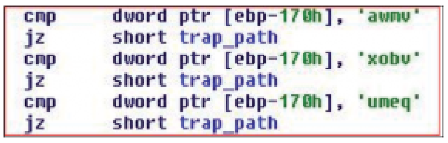
```
cmp     dword ptr [ebp-170h], 'awmv'
jz      short trap_path
cmp     dword ptr [ebp-170h], 'xobv'
jz      short trap_path
cmp     dword ptr [ebp-170h], 'umeq'
jz      short trap_path
```

Figure 7: Querying for virtual machines.

.

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 struc 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 conta 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 proce whereby the third routine is decrypted in that process. The data structure is shown in Figure 8.



```
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 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 st 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 pro 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'[1] has already been created. 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 msiexec.exe. This suggests that the bot injects its code into two different processes at different instances. We shall see why in the next section.

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

It then gets the Windows directory. Before injection, the bot needs to find the location of these files (svchost.exe, msiexec.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. 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 header.
2. 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. 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. 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 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. Bef 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 commer of the botnet's payload. In this version, Andromeda displays some new techniques in its execution.

First, it modifies the registry entry HKLMsystem currentcontrolsetservicessharedaccessparameters firewallpolicystandardprofileauthorizedapplicationslist 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 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 prese 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 receivec Additionally, the two processes communicate with each other through the instance of creating a pipe connection. It is this connection that enables either process to check 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 Process 2, so as to better understand the communication between the two processes (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.

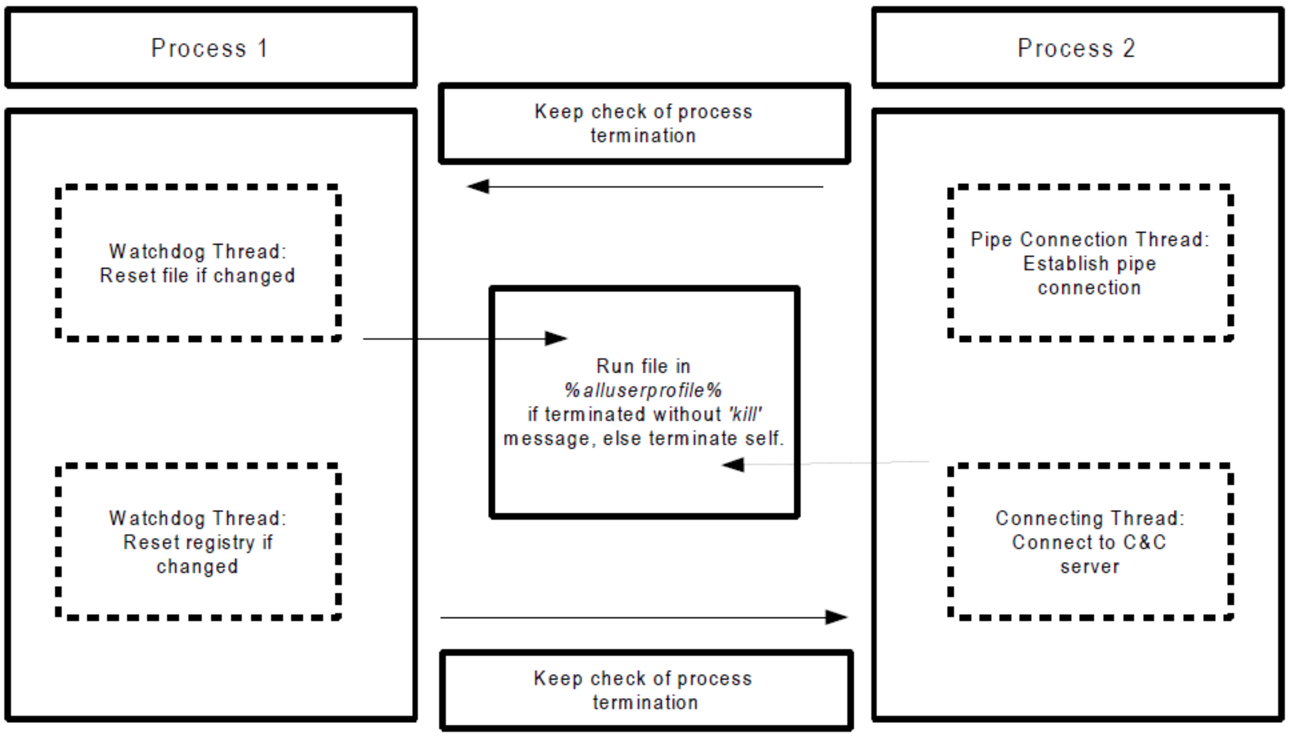
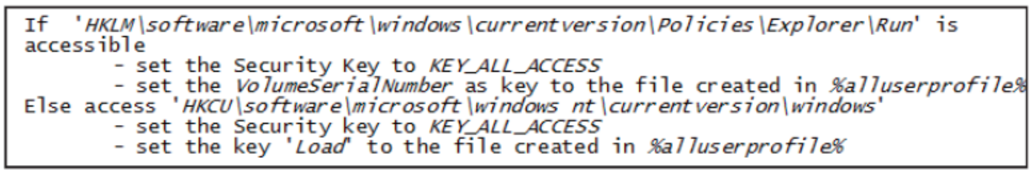

Process 1
Process 2
Keep check of process termination
Watchdog Thread: Reset file if changed
Pipe Connection Thread: Establish pipe connection
Run file in %alluserprofile% if terminated without 'kill' message, else terminate self.
Watchdog Thread: Reset registry if changed
Connecting Thread: Connect to C&C server
Keep check of process termination

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

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 cr 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 softwaremicrosoft windowscurrentversionPoliciesExplorerRun in registry HKLM. If it is unsuccessful, it accesses the subkeysoftwaremicrosoftwindows ntcurrentversionwindows in HKCU. Th 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 s 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 regis 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 che 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 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 hanc 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 cor 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 %alluserprof 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 thre 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 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 threa 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 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. Wh 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 cl 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 p and runs the file in %alluserprofile% again.

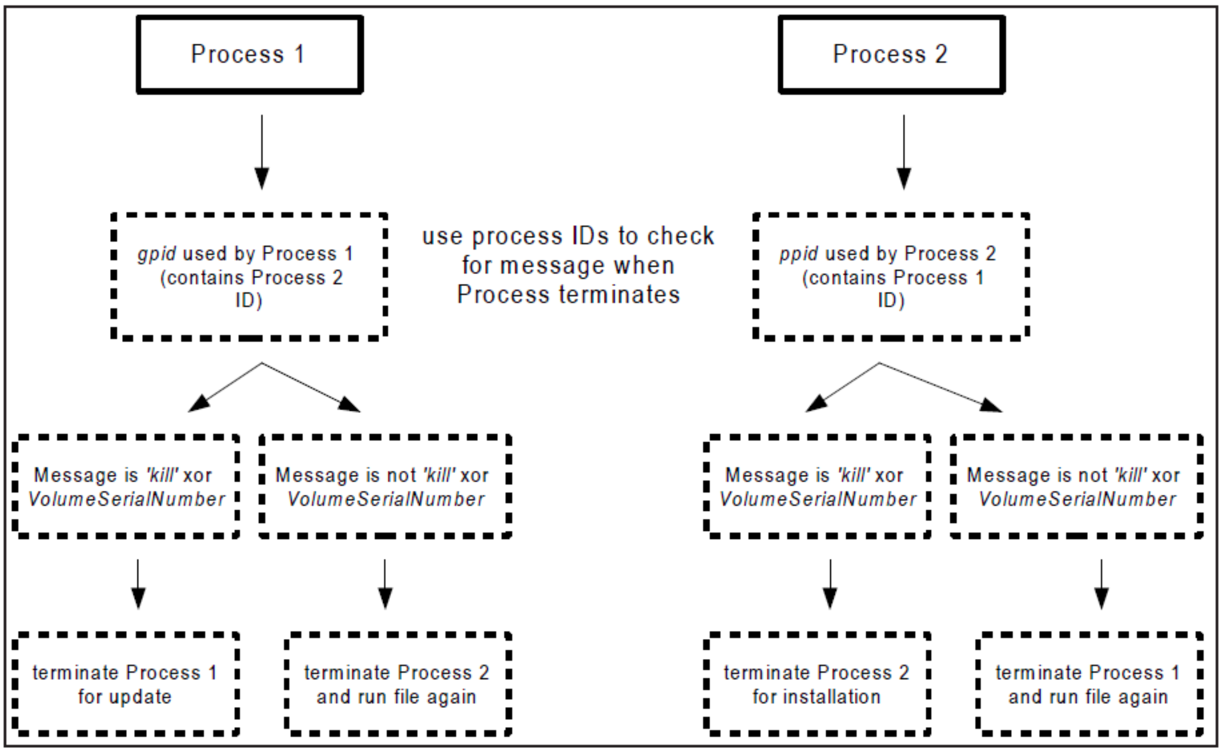
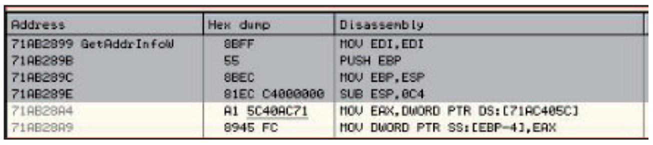

Process 1
Process 2
gpid used by Process 1 (contains Process 2 ID)
use process IDs to check for message when Process terminates
ppid used by Process 2 (contains Process 1 ID)
Message is *'kill'* xor *VolumeSerialNumber*
Message is not *'kill'* xor *VolumeSerialNumber*
Message is *'kill'* xor *VolumeSerialNumber*
Message is not *'kill'* xor *VolumeSerialNumber*
terminate Process 1 for update
terminate Process 2 and run file again
terminate Process 2 for installation
terminate Process 1 and run file again

Figure 11: Process IDs used by the processes.

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 proce 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 term 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.

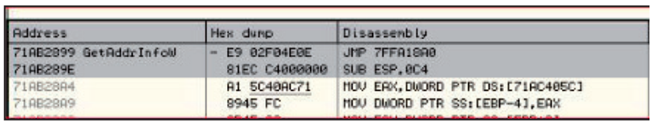
The second thread created by the new process carries out some further code injection. It first resolves winhttp.dll APIs using the anti-API hooking technique and also inline hooks three APIs: ws2_32.GetAddrInfoW (Figures 12 and 13), ntdll.ZwMapViewOfSection and ntdll.ZwUnmapViewOfSection. The control flow of the



```
Address                 Hex dump          Disassembly
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
```

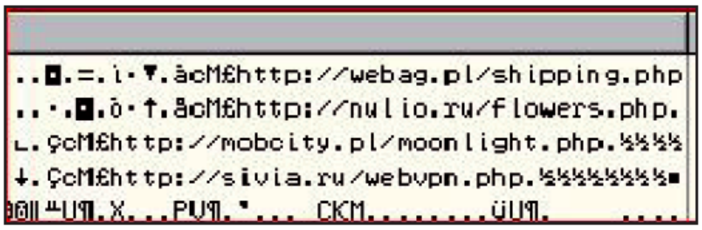
APIs is redirected by inserting a jump to the malicious function. Before writing to the API, it calls VirtualProtect. After the bytes have been written, it calls FlushInstructionCache so that the changes take effect immediately.

Figure 12: Before inline hooking GetAddrInfoW.



```
Address                  Hex dump          Disassembly
71AB2899 GetAddrInfoW  - E9 02F04E0E       JMP 7FFA18A0
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.



```
..■.=.ì·▼.åcM£http://webag.pl/shipping.php
..·.■.ò·↑.åcM£http://nulio.ru/flowers.php.
∟.çcM£http://mobcity.pl/moonlight.php.½½½½
↓.ÇcM£http://sivia.ru/webvpn.php.½½½½½½½■
```

Figure 14: The decrypted domain names.

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

Communication Protocol, Encryption Algorithm

Create connections

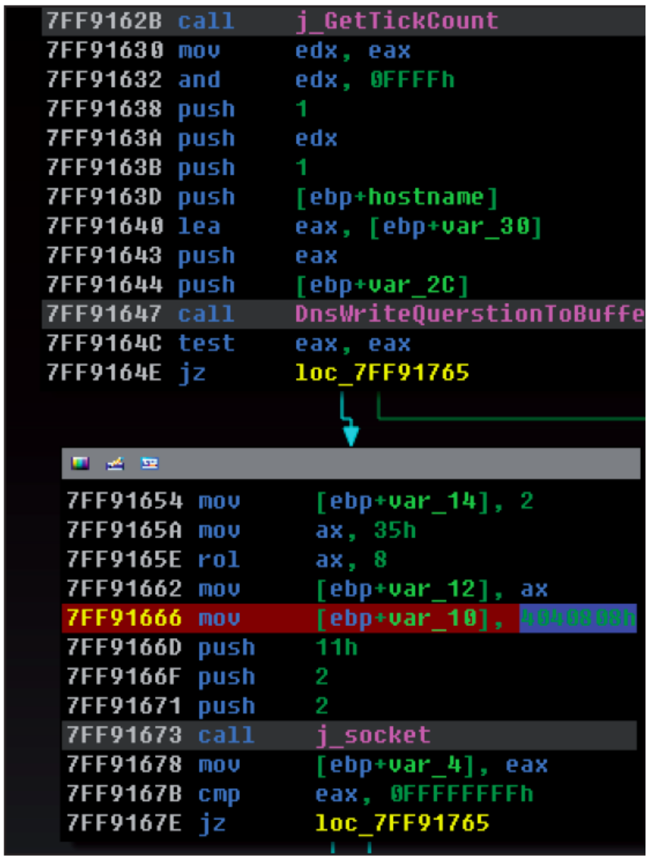
The hooked GetAddrInfoW API performs a DNS query for the input host name from Google DNS server 8.8.4.4 (Figure 15) using a randomly generated query identifier. It then returns the query result or '127.0.0.1' if the DNS query fails. The DNS record received is then used for querying the C&C domain name. 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.

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
```

1. id is the VolumeSerialNumber, which is used as an RC4 key to decrypt the message received

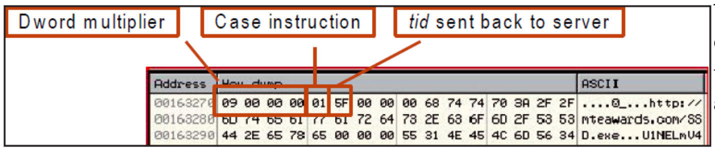


```
7FF9162B call    j_GetTickCount
7FF91630 mov     edx, eax
7FF91632 and     edx, 0FFFFh
7FF91638 push    1
7FF9163A push    edx
7FF9163B push    1
7FF9163D push    [ebp+hostname]
7FF91640 lea     eax, [ebp+var_30]
7FF91643 push    eax
7FF91644 push    [ebp+var_2C]
7FF91647 call    DnsWriteQuerstionToBuffe
7FF9164C test    eax, eax
7FF9164E jz      loc_7FF91765

7FF91654 mov     [ebp+var_14], 2
7FF9165A mov     ax, 35h
7FF9165E rol     ax, 8
7FF91662 mov     [ebp+var_12], ax
7FF91666 mov     [ebp+var_10], 40400808h
7FF9166D push    11h
7FF9166F push    2
7FF91671 push    2
7FF91673 call    j_socket
7FF91678 mov     [ebp+var_4], eax
7FF9167B cmp     eax, 0FFFFFFFFh
7FF9167E jz      loc_7FF91765
```

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

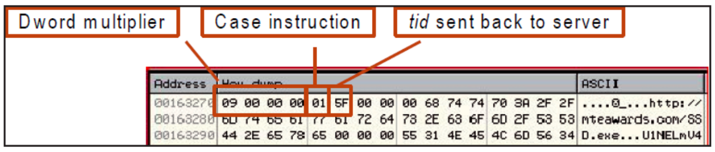
1. bid is a hard-coded DWORD used for the communication
2. bv is the version of the botnet (in this case it is 2.7)
3. os is version of the current operating system
4. la is the socket named byte swapped
5. rg is set to 1 if the process is in the Administrator group, otherwise it is 0 (Figure 16).

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 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 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 us 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 string inside the brackets to get the message in plain text (Figure 17).



```
Address  Hex dump                                          ASCII
00163270 09 00 00 00 01 5F 00 00 00 68 74 74 70 3A 2F 2F  ....@_...http://
00163280 6D 74 65 61 77 61 72 64 73 2E 63 6F 6D 2F 53 53  mteawards.com/SS
00163290 44 2E 65 78 65 00 00 00 55 31 4E 45 4C 6D 56 34  D.exe...U1NELmV4
```

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



```
Address  Hex dump                                          ASCII
00163270 09 00 00 00 01 5F 00 00 00 68 74 74 70 3A 2F 2F  ....@_...http://
00163280 6D 74 65 61 77 61 72 64 73 2E 63 6F 6D 2F 53 53  mteawards.com/SS
00163290 44 2E 65 78 65 00 00 00 55 31 4E 45 4C 6D 56 34  D.exe...U1NELmV4
```

Figure 17: Message received from the server

The first DWORD of the message is used as a multiplier to multiply a value in a fix offset. The DWORD in that offset is used as an interval to delay calling the thread to establish another connection. The next byte indicates what action to carry out are seven options:

1. Case 1 (download EXE): Connect to the domain decrypted from the message download an EXE file. Save the file to the %tmp% location with a random nam and run the process.
2. 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 k of length 0x20h.

1. Case 3 (update case): Connect to the domain to get the update EXE file. If a file name of VolumeSerialNumber is present in theregistry, then save the PE file to the %tm 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.
2. 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 fr a specified export function. The registry is modified so it can be auto-loaded by the bot.
3. Case 5 (delete DLLs): Delete and uninstall all the DLLs loaded and installed in Case 4.
4. Case 6 (delete plug-ins): Uninstall all the plug-ins loaded in Case 3.
5. 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
```

1. id is the VolumeSerialNumber
2. tid is the next byte (task id) after the byte displaying the case number in the message received
3. 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.

Conclusions

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 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. Morec could easily block its communication data after addressing the decryption performance issue.

by **Neo Tan (/author/neo-tan)** | April 23, 2014 | Category: Security Research (/category/security-research)

9 36 0 Google + 0

Tags: security (/tag/security) | botnet (/tag/botnet) | malware (/tag/malware-1) | ddos (/tag/ddos-1) | Virus bulletin (/tag/virus-bulletin-2) | andromeda botnet (/tag/andromed botnet)

0 Comments Fortinet Blog 1 L

Recommend Share Sort by

Start the discussion…

Be the first to comment.

Subscribe Add Disqus to your site Privacy DIS

FortiGuard Labs on the Web

Twitter (http://www.twitter.com/fortiguardlabs) Facebook (http://www.faceb

LinkedIn (http://www.linkedin.com/groups?gid=1321377&trk=hb_side_g) Youtube (http://www.youtub

Monthly Archives

September 2015 (/2015/09)	2
August 2015 (/2015/08)	17
July 2015 (/2015/07)	22
June 2015 (/2015/06)	18
May 2015 (/2015/05)	16
April 2015 (/2015/04)	34
March 2015 (/2015/03)	17
February 2015 (/2015/02)	11
January 2015 (/2015/01)	16
December 2014 (/2014/12)	7
November 2014 (/2014/11)	19
October 2014 (/2014/10)	16
September 2014 (/2014/09)	11
August 2014 (/2014/08)	11
July 2014 (/2014/07)	20
June 2014 (/2014/06)	21
May 2014 (/2014/05)	19
April 2014 (/2014/04)	16
March 2014 (/2014/03)	20
February 2014 (/2014/02)	15
January 2014 (/2014/01)	25
December 2013 (/2013/12)	10
November 2013 (/2013/11)	15
October 2013 (/2013/10)	19
September 2013 (/2013/09)	19
August 2013 (/2013/08)	14

July 2013 (/2013/07)	**14**
June 2013 (/2013/06)	**2**
April 2013 (/2013/04)	**1**
March 2013 (/2013/03)	**12**
February 2013 (/2013/02)	**11**
January 2013 (/2013/01)	**12**
December 2012 (/2012/12)	**8**
November 2012 (/2012/11)	**7**
October 2012 (/2012/10)	**4**
September 2012 (/2012/09)	**6**
August 2012 (/2012/08)	**7**
July 2012 (/2012/07)	**62**
June 2012 (/2012/06)	**17**
May 2012 (/2012/05)	**14**
April 2012 (/2012/04)	**15**
March 2012 (/2012/03)	**14**
February 2012 (/2012/02)	**11**
January 2012 (/2012/01)	**6**
December 2011 (/2011/12)	**4**
November 2011 (/2011/11)	**6**
October 2011 (/2011/10)	**11**
September 2011 (/2011/09)	**2**
August 2011 (/2011/08)	**2**
July 2011 (/2011/07)	**4**
June 2011 (/2011/06)	**6**
May 2011 (/2011/05)	**6**
April 2011 (/2011/04)	**5**
March 2011 (/2011/03)	**7**
February 2011 (/2011/02)	**5**
January 2011 (/2011/01)	**7**
December 2010 (/2010/12)	**8**
November 2010 (/2010/11)	**11**
October 2010 (/2010/10)	**3**
September 2010 (/2010/09)	**8**
August 2010 (/2010/08)	**4**
July 2010 (/2010/07)	**9**
June 2010 (/2010/06)	**9**
May 2010 (/2010/05)	**9**
April 2010 (/2010/04)	**6**
March 2010 (/2010/03)	**8**
February 2010 (/2010/02)	**6**
January 2010 (/2010/01)	**9**
December 2009 (/2009/12)	**8**

November 2009 (/2009/11)	6
October 2009 (/2009/10)	6
September 2009 (/2009/09)	8
August 2009 (/2009/08)	5
July 2009 (/2009/07)	8
June 2009 (/2009/06)	7
May 2009 (/2009/05)	4
April 2009 (/2009/04)	7
March 2009 (/2009/03)	9
February 2009 (/2009/02)	4
January 2009 (/2009/01)	1

Corporate

About Fortinet (http://fortinet.com/aboutus/aboutus.html)

Investor Relations (http://investor.fortinet.com/)

Careers (http://jobs.fortinet.com/)

Press Room (http://fortinet.com/press_releases/press.html)

Partners (http://fortinet.com/partners/index.html)

Global Offices (http://fortinet.com/aboutus/locations.html)

Fortinet Blog (http://blog.fortinet.com/)

Fortinet in the News (http://fortinet.com/aboutus/media/news.html)

Events (http://fortinet.com/events/index.html)

Contact Us (http://fortinet.com/contact_us/index.html)

How to Buy

Find a Reseller (http://fortinet.com/partners/reseller_locator/locator.html)

FortiPartner Program (http://fortinet.com/partners/partner_program/fpp.html)

Try & Buy (http://fortinet.com/how_to_buy/try_and_buy.html)

Fortinet Store (https://store.fortinet.com)

Products

Product Family (http://fortinet.com/products/index.html)

Certifications (http://fortinet.com/aboutus/fortinet_advantages/certifications.html)

Awards (http://fortinet.com/aboutus/fortinet_advantages/awards.html)

Video Library (http://video.fortinet.com/)

Service & Support

FortiCare Support (http://fortinet.com/support/forticare_support/index.html)

Support Helpdesk (https://support.fortinet.com/)

FortiGuard Center (http://fortiguard.com/)

Fortinet Blog (http://blog.fortinet.com)

(http://www.facebook.com/fortinet)

(http://www.twitter.com/fortinet)

(http://www.youtube.com/user/SecureNetworks)

(http://www.linkedin.com/company/fortinet)

(http://fortinet.com/rss.xml