VICUS BULLETIN Global threat landscape

A REVIEW OF THE EVOLUTION OF ANDROMEDA OVER THE YEARS BEFORE WE SAY GOODBYE!

Bahare Sabouri & He Xu Fortinet, Canada

Andromeda, also known as Gamaru and Wauchos, is a modular and HTTP-based botnet that was discovered in late 2011. From that point on, it managed to survive and continue hardening by evolving in different ways. In particular, the complexity of its loader and AV evasion methods increased repeatedly, and C&C communication changed between the different versions as well.

We deal with versions of this threat on a daily basis and we have collected a number of different variants. The botnet first came onto our tracking radar at version 2.06, and we have tracked the versions since then. In this paper we will describe the evolution of Andromeda from version 2.06 to 2.10 and demonstrate both how it has improved its loader to evade automatic analysis/detection and how the payload varies among the different versions.

This article could also be seen as a way to say 'goodbye' to the botnet: a takedown effort, followed by the arrest of the suspected botnet owner in December 2017, may mean we have seen the last of the botnet that has plagued Internet users for more than a decade.

OVERVIEW OF ANDROMEDA

The first Andromeda to be discovered was spotted in the wild in 2011, and the new 2.06 version followed quickly afterwards in early 2012. Not much is known about any earlier versions and it is possible they were never released into the wild.

The campaign continued to develop with versions 2.07, 2.08, 2.09 and 2.10. The latest known version, 2.10, was first seen in 2015 and may be the final version released: according to posts on underground forums, the development of the threat stopped around a year ago. Figure 1 shows a brief history of Andromeda.

Regardless of the version, Andromeda arrives on the target machine as a packed sample. Various packers have been used, from very famous packers such as UPX and SFX RAR to lesser known and even customized ones which are compiled in various languages such as Autoit, .Net and C++.

Unpacking the first layer of the sample reveals the loader, which is small both in terms of size (13KB to 20KB) and in the number of function calls it contains.

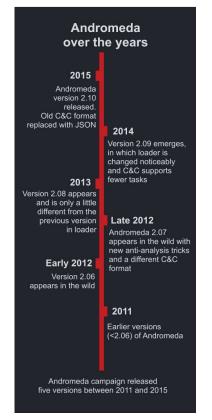


Figure 1: A brief history of Andromeda.

1961:00401929	test	eax, eax
1961:0040192B	jz	loc_401A2E API Hash
1961:00401931	push	86B0A95Ah
1961:00401936	push	[ebp+var_C]
_1961:00401939	call	resolveAddress_byHash
1961:0040193E	test	eax, eax
1961:00401940	jz	loc_401A0B
1961:00401946	push	104h
_1961:0040194B	push	[ebp+var_10]
_1961:0040194E	call	eax
1961:00401950	add	eax, [ebp+var_10]
_1961:00401953	mov	dword ptr [eax], 642E2A5Ch
1961:00401959	add	eax, 4
_1961:0040195C	mov	dword ptr [eax], 6C6Ch
_1961:00401962	push	75272948h
_1961:00401967	push	[ebp+var_C]
_1961:0040196A	call	resolveAddress_byHash
_1961:0040196F	mov	[ebp+var_158], eax
_1961:00401975	test	eax, eax
_1961:00401977	jz	loc_401A0B
_1961:0040197D	push	0C9EBD5CEh
_1961:00401982	push	[ebp+var_C]
_1961:00401985	call	resolveAddress_byHash
_1961:0040198A	mov	edx, eax
_1961:0040198C	test	eax, eax
_1961:0040198E	jz	short loc_401A0B
_1961:00401990	lea	eax, [ebp+var_152]
1961:00401996	push	eax
_1961:00401997	push	[ebp+var_10]
1961:0040199A	call	edx

Figure 2: Version 2.08 passes the hash as an immediate value to 'resolveAddress_byHash'.

LOADER

In all versions of Andromeda the loader avoids making direct calls to APIs. Instead, it incorporates hashes to find and call

the APIs via general purpose registers. Versions 2.06, 2.07 and 2.08 pass hash values as immediate values to a function and thus find the matching API name. Version 2.06 uses a custom hash function, while versions 2.07 and 2.08 use CRC32. Versions 2.09 and 2.10 have the same trivial custom hash function. Figure 3 shows the loader in version 2.09 handling an array of hash values.

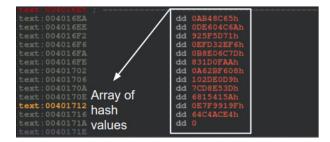


Figure 3: In version 2.09, the loader handles an array of hash values.

Version 2.10 also keeps an array of API hash values. The hash algorithm is a custom function and, in order to complicate static analysis further, the author incorporates opaque predicates, as shown in Figure 4.

MAIN STRUCTURE

The section in the loader that is used to evade virtual machines and, more generally, analysis, is similar in versions 2.06, 2.07 and 2.08. In these variants, the loader enumerates the processes running on the machine and compares them against a list of unwanted processes. In order to do this, the loader converts the name of each process to lowercase and then calculates its hash value.

The hash values are then compared against a hard-coded list of values. The same algorithm as is used to hash API names is used here. The hash algorithm in version 2.08 has an extra xor instruction (xor eax, 0E17176Fh). As shown in Figure 5, the newer versions have longer lists of unwanted processes.

Next, the bot takes advantage of registry artifacts and checks the registry value in the following key:

Key: HKLM\system\currentcontrolset\services\disk\enum ValueName: 0

Version 2.06 parses the value of the subkey for the presence of the substrings 'qemu', 'vbox' and 'wmwa'. Similarly, versions 2.07 and 2.08 check for 'qemu', 'vbox' and 'vmwa'. (It is likely that 'wmwa' was a bug in version 2.06 that was patched later.) Upon finding any of these strings, each version takes a different approach to redirect the flow of the code.

Before redirecting the code in versions 2.06 and 2.07, the sample designates another snippet of code that uses a technique known as 'time attack' in order to prevent further analysis. The malware acquires the timestamp counter (by calling rdtsc) twice and calculates the difference between the two. If the difference is less than 512ms, it proceeds to resolve imports and decrypt the payload. Otherwise, it leads to a dummy code, where the loader drops a copy of itself in %ALLUSERSPROFILE% and renames it to svchost.exe.

Following that, it creates an autorun registry for the dropped file as follows:

Key: HKLM\SOFTWARE\Microsoft\Windows\ CurrentVersion\Run ValueName: SunJavaUpdateSched



Figure 4: Opaque predicates in the version 2.10 loader make static anaylsis more difficult.

				call	Convert_Str_LowerCase
				lea	eax, [ebp-144h]
				push	eax
				call	CalcHash Process
				xor	eax, 0E17176Fh
				cmp	eax, 97CA535Dh
				jz	loc_401EF3
				cmp	eax, 23928ADBh
				jz	loc_401EF3
				cmp	eax, 6A231AA1h
				jz	loc_401EF3
				cmp	eax, 6DD2531Bh
			; 540	jz	loc_401EF3
		lea	eax, [ebp-144h]	cmp	eax, 3A8B8BE4h
		push	eax	jz	loc_401EF3
		call	Convert_Str_LowerCase	cmp	eax, 3A51FCA1h
		lea	eax, [ebp-144h]	jz	loc_401EF3
lea	eax, [ebp+var_150]	push	eax	cmp	eax, 55BEA691h
push	eax	call	CalcHash_Process	jz	loc_401EF3
call	Convert_Str_LowerCase	cmp	eax, 99DD4432h	cmp	eax, 32F5A99Ch
lea	eax, [ebp+var_150]	jz	loc_401E97	jz	loc_401EF3
push	eax	cmp	eax, 2D859DB4h	cmp	eax, 3351E744h
call	CalcHash_Process	jz	loc_401E97	jz	loc_401EF3
cmp	eax, 4CE5FD07h	cmp	eax, 64340DCEh	cmp	eax, 79B90798h
jz	loc_401767	jz	loc_401E97	jz	loc_401EF3
cmp	eax, 8181326Ch	cmp	eax, 63C54474h	cmp	eax, 0FD53FE32h
jz	loc_401767	jz	loc_401E97	jz	loc_401EF3
cmp	eax, 31E233AFh	cmp	eax, 349C9C8Bh	cmp	eax, 23A97A00h
jz	loc_401767	jz	loc_401E97	jz	loc_401EF3
cmp	eax, 91D47DF6h	cmp	eax, 3446EBCEh	cmp	eax, 0ADC6152Bh
jz	loc_401767	jz	loc_401E97	jz	loc_401EF3
cmp	eax, 0E8CDDC54h	cmp	eax, 5BA9B1FEh	cmp	eax, 1365FAFEh
jz	loc_401767	jz	loc_401E97	jz	loc_401EF3
cmp	eax, 8C6D6Ch	cmp	eax, 3CE2BEF3h	cmp	eax, 98847CD1h loc_401EF3
jz	loc_401767 eax, 0A8D0BA0Eh	jz	loc_401E97	jz	eax, 299BC837h
cmp		cmp	eax, 3D46F02Bh	cmp	loc_401EF3
jz	loc_401767 eax, 0A4EF3C0Eh	jz	1oc_401E97	jz cmp	eax, 35E8EFEAh
cmp jz	loc_401767	cmp	eax, 77AE10F7h loc_401E97	jz	loc_401EF3
cmp	eax, 5CD7BA5Eh	jz	eax, 0F344E95Dh	cmp	eax, 632434B6h
jz	loc_401767	cmp jz	loc_401E97	jz	loc_401EF3
lea	eax, [ebp+var_174]	lea	eax, [ebp-168h]	lea	eax, [ebp-168h]
push	eax, [ebp+var_1/4]	push	eax, [ebp-100h]	push	eax [cop-room]
push	[ebp+var_4C]	push	dword ptr [ebp-40h]	push	dword ptr [ebp-40h]
call	[ebp+var_10]	call	dword ptr [ebp-10h]	call	dword ptr [ebp-24h]
test	eax, eax	test	eax, eax	test	eax, eax
jnz	loc_401508	jnz	loc 401C65	jnz	loc_401CD2
		J		-	

Figure 5: From left to right: version 2.06, 2.07 and 2.08 hard-coded hash values correspond to the list of unwanted processes.

2.06	2.07	2.08
0x4CE5FD07: vmwareuser.exe	0x99DD4432: vmwareuser.exe	0x97CA535D: vmwareuser.exe
0x8181326C: vmwareservice.exe	0x2D859DB4: vmwareservice.exe	0x23928ADB: vmwareservice.exe
0x31E233AF: vboxservice.exe	0x64340DCE: vboxservice.exe	0x6A231AA1: vboxservice.exe
0x91D47DF6: vboxtray.exe	0x63C54474: vboxtray.exe	0x6DD2531B: vboxtray.exe
0xE8CDDC54: sandboxiedcomlaunch.exe	0x349C9C8B: sandboxiedcomlaunch.exe	0x3A8B8BE4: sandboxiedcomlaunch.exe
0x8C6D6C: sandboxierpcss.exe	0x3446EBCE: sandboxierpcss.exe	0x3A51FCA1: sandoxierpcss.exe
0x0A8D0BA0E: procmon.exe	0x5BA9B1FE: procmon.exe	0x55BEA691: procmon.exe
0x0A4EF3C0E: wireshark.exe	0x3CE2BEF3: regmon.exe	0x32F5A99C: regmon.exe
0x5CD7BA5E: netmon.exe	0x3D46F02B: filemon.exe	0x3351E744: filemon.exe
	0x77AE10F7: wireshark.exe	0x79B90798: wireshark.exe
	0x0F344E95D: netmon.exe	0x0FD53FE32: netmon.exe
		0x23A97A00: prl_tools_service.exe
		0x0ADC6152B: prl_tools.exe
		0x1365FAFE: prl_cc.exe
		0x98847CD1: sharedintapp.exe
		0x299BC837: vmtoolsd.exe
		0x35E8EFEA: vmsrvc.exe
		0x632434B6: vmusrvc.exe

Table 1: Corresponding process to each hash.

<pre>.text:00401746 .text:00401746 .text:00401748 .text:00401749 .text:00401740 .text:0040174C .text:00401745 .text:00401753 .text:00401753</pre>		rdtsc push rdtsc pop sub cmp jnb	; start+1E3fj eax edx eax, edx eax, 200h short loc_401767
.text:00401755 .text:00401755 .text:0040175B .text:00401761	loc_401755:	lea mov lea	; CODE XREF: start+ACfj eax, dword_401778 [ebp+var_188], eax eax, dummyCode

Figure 6: Timestamp analysis to detect the debugger.

.text:00401B84 1	.oc_401B84:	; CODE XREF: sub_401B62+71j
.text:00401B84	push	8007h
.text:00401B89	call	dword ptr [ebp-10h]
.text:00401B8C	push	offset custom_exception_handler
.text:00401B91	call	dword ptr [ebp-SetUnhandledExceptionFilter]
.text:00401B94	mov	ebx, large fs:30h
.text:00401B9B	mov	ebx, [ebx+0Ch]
.text:00401B9E	mov	ebx, [ebx+0Ch]

Figure 7: Bot creates a custom exception handler in version 2.07.

1961:00401BEE mov eax, large fs:30h 1961:00401BF4 mov eax, [eax+0Ch]	_1961:00401BD7 _1961:00401BD8 _1961:00401BDD _1961:00401BDF _1961:00401BE5 _1961:00401BEA _1961:00401BEC	push call jz push push call	eax sub_401746 eax, eax loc_401EFE offset custom_exception_handler 1 eax ; RtlAddVectoredExceptionHandler
_1961:00401BF7 mov eax, [eax+0Ch] 1961:00401BFA mov esi, [eax+28h]	_1961:00401BEE _1961:00401BF4 _1961:00401BF7	mov mov mov	eax, large fs:30h eax, [eax+0Ch] eax, [eax+0Ch]

Figure 8: Bot adds a custom exception handler into VEH in version 2.08.

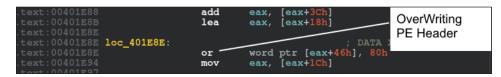


Figure 9: Overwriting the PE header raises an exception.

00280000	00041000	\Device\Har \Device\Har \Device\Har		MAP MAP MAP	-R -R -R	-R -R
00320000	00006000	\Device\Har loader.exe		MAP	-R	-R
00401000	00005000	".text"	Executable code	IMG IMG	-R	ERWC-
7C800000	00001000	kernel32.dl	Read-only initialized data	IMG	-R -R	ERWC- ERWC-
	00084000 00005000		Executable code Initialized data	IMG IMG	ER -RW	ERWC-
7C88A000 7C8F0000	00066000 00006000		Resources Base relocations	IMG IMG	-R -R	ERWC- ERWC-

Figure 10: The PE header only has read rights.

Eventually, waiting for a command in an infinite loop, it sniffs port 8000. A received command will then be run in the command window.

As part of its evolution, version 2.07 implements a custom exception handler using a call to SetUnhandledExceptionFilter. Similarly, version 2.08 calls RtlAddVectoredExceptionHandler and adds the custom handler as the first handler into the vectored exception handler chain (VEH), as shown in Figures 7 and 8. If the malware finds any of the substrings in the retrieved registry, it runs a function that causes an access violation. The access violation is created intentionally when the sample tries to overwrite the DLL characteristics in the PE header which only has read rights, as shown in Figures 9 and 10.

In this case, if the sample is not being debugged, control is passed immediately to the custom handler. The custom exception handler decrypts a piece of code that will be injected into another process later (Figure 11).

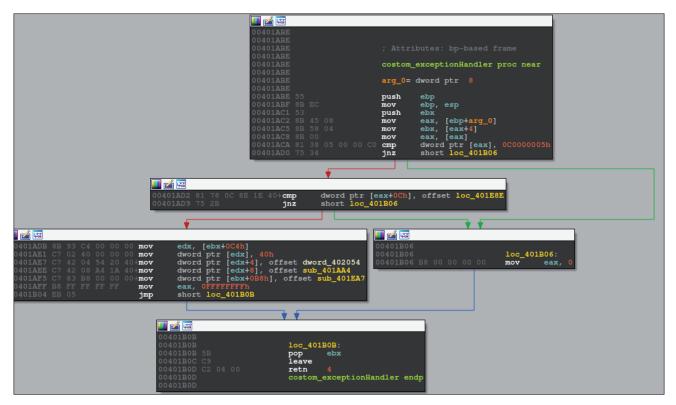


Figure 11: Custom exception handler.

_1961:00401BEE	xor	ecx, ecx
_1961:00401BF0	push	ecx
_1961:00401BF1	push	ecx
_1961:00401BF2	push	ecx
_1961:00401BF3	push	ecx
_1961:00401BF4	push	ecx
_1961:00401BF5	push	200h
_1961:00401BFA	lea	eax, [ebp-36Ch]
_1961:00401C00	push	eax
_1961:00401C01	lea	eax, [ebp-36Ch]
_1961:00401C07	push	eax
_1961:00401C08	call	dword ptr [ebp-GetVolumeInformationA]
_1961:00401C0B	lea	eax, [ebp-36Ch]
_1961:00401C11	push	eax
_1961:00401C12	call	calc_crc32
_1961:00401C17	cmp	eax, 20C7DD84h
_1961:00401C1C	jz	skip_antiVmChecks

Figure 12: Drive C checksum is calculated and compared to 0x20C7DD84.

Versions 2.07 and 2.08 share another feature that controls whether the loader bypasses anti-VM and anti-debugging procedures. The loader calls GetVolumeInformationA on the 'C:\' drive and acquires the drive name. Next, it calculates the CRC32 of the drive name and compares it against a hard-coded value, 0x20C7DD84 (Figure 12). If they match, it bypasses the anti-forensics checks and proceeds directly to invoke the exception. The author probably used this technique to test the bot in his/her virtual machine without the need to go through the anti-VM/anti-analysis features.

Versions 2.09 and 2.10 evade debugging and analysis by implementing the same idea as previous versions, but this

time in the payload. Eventually, in all versions, the loader injects the payload into a remote process using a process hollowing technique and runs it in memory.

PAYLOAD

As mentioned, the payloads of versions 2.09 and 2.10 start with some anti-VM tricks, despite the earlier versions having taken care of this in the loader. Like the older versions, they check for a list of blacklisted processes in case the machine is compromised. The number of blacklisted processes in version 2.09 is exactly the same as in 2.08, whereas it increases to

	F	
	d	d 2D859DB4h
	d	d 64340DCEh
proc_name_crc32 dd 99DD4432h	d	d 63C54474h
dword_7FF902AC dd 2D859DB4h	d	d 349C9C8Bh
dd 64340DCEh	d	d 3446EBCEh
dd 63C54474h	d	d 5BA9B1FEh
dd 349C9C8Bh	d	d 3CE2BEF3h
dd 3446EBCEh	d	d 3D46F02Bh
dd 5BA9B1FEh	d	d 77AE10F7h
dd 3CE2BEF3h	d	d OF344E95Dh
dd 3D46F02Bh	d	d 2DBE6D6Fh
dd 77AE10F7h	d	d 0A3D10244h
dd 0F344E95Dh	d	d 1D72ED91h
dd 2DBE6D6Fh	d	d 96936BBEh
dd 0A3D10244h	. d	d 278CDF58h
dd 1D72ED91h	d	d 3BFFF885h
dd 96936BBEh	d	d 6D3323D9h
dd 278CDF58h	d	d 0D2EFC6C4h
dd 3BFFF885h	d	d 0DE1BACD2h
dd 6D3323D9h	d	d 3044F7D4h
db 0	d	b 0
db 0	d	b 0
db 0	d	b 2 dup(0)
db 0		

proc_name_crc32 dd 99DD4432h

Figure 13: The number of blacklisted processes increases in version 2.10.

21 processes in version 2.10 (see Figure 13). Like versions 2.07 and 2.08, versions 2.09 and 2.10 calculate the CRC32 of the process name. However, instead of implementing the algorithm, they call RtlComputeCrc32 directly. If the bot finds any of the target processes, it runs a snippet of code to sleep for one minute in an infinite loop in order to evade detection.

If 'HKLM\software\policies' contains the registry key 'is_not_vm' and the key is VolumeSerialNumber, version 2.10 bypasses these checks. This behaviour is comparable to that in versions 2.07 and 2.08 where the bot checked the checksum of the root drive.

EVOLUTION OF C&C

The main aim of Andromeda's payload is to steal the infected system's information, talk to the command-and-control (C&C) server, and download and install additional malware onto the system. In order to do this, it initiates a sophisticated

Version	Action Request	Task Report
2.06	id:%lulbid:%lulbv:%lulsv:%lulpa:%lulla:%lular:%lu	id:%lultid:%lulresult:%lu
2.07	id:%lulbid:%lulbv:%lulos:%lulla:%lulrg:%lu	id:%lultid:%lulres:%lu
2.08	id:%lulbid:%lulbv:%lulos:%lulla:%lulrg:%lu	id:%lultid:%lulres:%lu
2.09	id:%lulbid:%lulos:%lulla:%lulrg:%lu	id:%lultid:%lulerr:%lulw32:%lu
2.10	{"id":%lu,"bid":%lu,"os":%lu,"la":%lu,"rg":%lu} {"id":%lu,"bid":%lu,"os":%lu,"la":%lu,"rg":%lu,"bb":%lu}	{"id":%lu,"tid":%lu,"err":%lu,"w32":%lu}

Table 2: Evolution of the message formats.

Action Request		Task Report	
Tag	Information	Tag	Information
id	Volume serial number of victim machine	id	Volume serial number of victim machine
bid	Bot ID, a hard-coded DWORD in payload	tid	Task ID provided by server
bv	Bot version	res/result/err	Flag indicating if task is successful
ра	Flag indicating whether OS is 32-bit or 64-bit	w32	System error code, returned by RtlGetLastWin32Error
la	Local IP address acquired from sockaddr structure		
ar/rg	Flag indicating if the process runs in the administrator group		
sv/os	Version of the victim operating system		
bb	Flag indicating if victim system uses a Russian, Ukrainian, Belarusian or Kazakh keyboard		

Table 3: Definition of tags.

6

command-and-control channel with the server. Each version of Andromeda uses a different format for the message and the report that it sends to the server.

As shown in Table 2, each version has two message formats, both sent as HTTP POST requests: Action Request and Task Report. Action Request contains the information exfiltrated from the compromised system; the bot sends it to the server after encryption. Task Report, as the name implies, provides a report about the accomplished task.

The Action Request format shares some essential tags among all versions, such as 'id' and 'bid', while some other tags are version-specific, such as 'ar' in version 2.06 and 'bb' in version 2.10. It is only the last version of the bot that uses JSON format to communicate with the C&C server.

Table 3 describes the role of each tag in the format.

We believe that 'bid' is used to represent build ID and, interestingly, in some versions, like 2.06 and 2.10, it indicates a date in the format YYYYMMDD, as can be seen in Figure 14. In other instances, this tag represents a hard-coded random number. The latest observed 'bid' in version 2.10 is 22 May 2017, which suggests that development stopped then.

After version 2.08, 'bv', which indicates the bot version, is removed from the request message. However, in the two latest versions, there remains a clue as to the bot version, which is a hard-coded xor key. This xor key is used in five different places in version 2.09 and twice in version 2.10. In all cases, it xors the 'id' and will be further manipulated to be used as the file name or registry value (see Figures 15 and 16).

When the message is prepared for the required information, in all versions except the most recent one, the string is encrypted in two steps. The first step uses a 20-byte hard-coded RC4 key and the second step uses base64 encoding. Version 2.10 encrypts the message only using the RC4 algorithm. After posting the message to the server, the bot receives a message from the server. The bot validates the message by calculating its CRC32 hash excluding the first DWORD, which serves as a checksum. If the hash equals this excluded DWORD, it proceeds to decrypt the message using the 'id' value as the RC4 key.

Next, it decodes the base64 string and obtains a plain text message. Received messages have the following structure:

```
struct RecvBlock {
    uint8_t cmd_id;
    uint32_t tid;
    char cmd_param[];
};
```

According to the communicated cmd_id, the bot carries out a designated command which could be any number from the following: 1, 2, 3, 4, 5, 6, 9. In versions prior to 2.09, the bot is capable of performing all seven tasks. But in versions 2.09 and 2.10, it discards commands 4 and 5.

In Table 4 we take a look at each task and describe it further using static analysis of the code.

It is interesting to note that the cmd_id value changes a little in versions 2.09 and 2.10. As a result, the bot first downloads the plug-in and later finds three plug-in exports, aStart,



Figure 14: 'bid' value in version 2.10.

mov xor push call call mov	; CODE XREF: DropCog ; DropCopy_Autorun_' eax, '0209' eax saveDate encrypt_loop edi, eax	push push xor mov call mov	ds:pHeap01 eax, '0210' ds:Seed, eax ebx ; SetFileAttributesW [esp+58h+var_48], 32h
push	edi		; CODE XREF: sub_4095D
lea	eax, [esi+ebx*2]	call	sub_4092EE
push	offset aMsS_exe ; "\\ms%s.exe"	mov	[esp+58h+var_3C], eax
push	eax	cmp	eax, ebp
call	ds:wsprintfW	jz	loc_4099D3
add	esp, OCh	push	eax
push	edi	push	offset aMsS_exe ; "ms%s.exe"
add	ebx, eax	push	esi ; pszPath
call	freeheap	call	j_PathFindFileNameW
mov	edi, FILE_ATTRIBUTE_NORMAL	push	eax

Figure 15: The bot version is represented as a hard-coded xor key and used as a file name.



Figure 16: The bot version is represented as a hard-coded xor key and used in registries.

cmd_id	Task type	Description
1	Download EXE	Using the domain provided in the command_parameter, the bot downloads an exe, saves it in the temp folder with a random name, and executes it.
2	Install plug-in	Using the domain provided in the command_parameter, the bot installs and loads plug-ins.
3	Update bot	Using the domain provided in the command_parameter, the bot gets the exe file to update itself. If a file named 'Volume Serial Number' exists in the registry, the bot drops the update in the temp folder and gives it a random name. Otherwise, the file is dropped in the current directory. This task is followed by cmd_id=9, which kills the older bot.
4	Install DLL	Using the domain provided in the command_parameter, the bot downloads a DLL into the %alluserprofile% folder with a random name and .dat extension.
5	Delete DLLs	The DLL loaded in cmd_id=4 is uninstalled.
6	Delete plug-ins	The plug-ins loaded in cmd_id=3 are uninstalled.
9	Kill bot	All threads are suspended and the bot is uninstalled.

Table 4: The seven command IDs and their tasks.

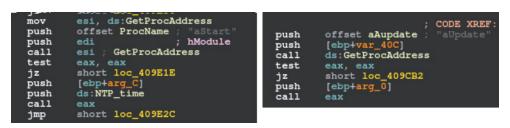


Figure 17: The payload also searches for plug-in exports aStart and aUpdate.

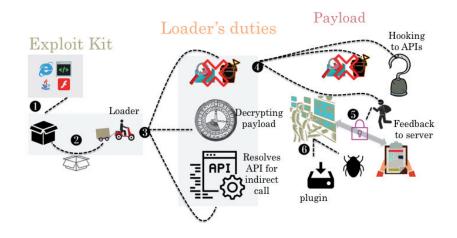


Figure 18: Andromeda at a glance.

aUpdate and aReport, using a call to the GetProcAddress API (Figure 17).

To summarize, Andromeda normally spreads via exploit kits located on compromised websites. The primary sample is packed and drops the loader after the unpacking stage. In the earlier versions of the bot the loader contains anti-VM and anti-analysis tricks. In all versions, the loader decrypts the payload and resolves APIs for indirect calls in the payload. As a result, using an anti-API hooking trick, the loader saves the first instruction of the API call into memory and jumps to the second instruction.

In the last two versions of the bot (2.09 and 2.10) the payload contains anti-VM and anti-analysis features. In version 2.07 and later versions, the payload leverages an inline hooking technique and hooks selected APIs. For example, in versions 2.07 and 2.08 the bot hooks GetAddrInfoW, ZwMapViewOfSection and ZwUnmapViewOfSection; in version 2.09 it hooks GetAddrInfoW and NtOpenSection; and in version 2.10 it hooks GetAddrInfoW and NtMapViewOfSection. In all versions, the bot steals information from the compromised system, sends the information to the server (after encryption), and waits for a command from the server.

Upon receiving a command from the server, the bot acts accordingly, installing plug-ins and downloading other

malware. Finally, the bot sends a report about its mission to the server.

SIDE NOTE

It has been a while since the last version of Andromeda was released. We have been waiting a long time for a new variant to emerge, but *Reuters* reported recently:

'National police in Belarus, working with the U.S. Federal Bureau of Investigation, said they had arrested a citizen of Belarus on suspicion of selling malicious software who they described as administrator of the Andromeda network.' [3]

Based on that, we can tentatively call this the end of the Andromeda era, and conclude that there won't be any further releases.

CONCLUSION

From 2011 to 2015, Andromeda kept analysts busy with its compelling features and functionality, and it remains among the most prevalent malware families today. Over the course of four years, five major versions were released, each new version being more complex than its predecessor. This guaranteed that Andromeda remained a sophisticated threat. A flexible C&C provided a wide range of functionality and efficiency, increasing the power of the threat by installing various modules. Meanwhile, it integrated several RC4 keys to encrypt data for C&C communications, thus making detection a significantly more complex challenge. Fortunately, however, analysts have become sufficiently familiar with Andromeda's ecosystem over the years to learn how to navigate all of its challenges.

REFERENCES

- Tan, N. Andromeda 2.7 features. Fortinet blog.
 23 April 2014. https://blog.fortinet.com/2014/04/23/ andromeda-2-7-features.
- [2] Xu, H. A good look at the Andromeda botnet. Virus Bulletin. May 2013. https://www.virusbulletin.com/ virusbulletin/2013/05/good-look-andromeda-botnet.
- [3] Sterling, T.; Auchard, E. Belarus arrests suspected ringleader of global cyber crime network. Reuters.
 5 December 2017. https://ca.reuters.com/article/ technologyNews/idCAKBN1DZ1VY-OCATC.
- [4] Xu, H. Cracked Andromeda 2.06 spreads bitcoinn miner. Fortinet blog. 7 January 2015. https://blog.fortinet.com/2015/01/07/crackedandromeda-2-06-spreads-bitcoin-miner.

SAMPLE INFORMATION

Version 2.06

MD5: 73564f834fd0f61c8b5d67b1dae19209 **SHA256:** 4ad4752a0dcaf3bb7dd3d03778a149ef1cf6a8237b2 1abcb525b9176c003ac3a

Fortinet detection name: W32/Kryptik.AFJS!tr

Version 2.07

MD5: d7c00d17e7a36987a359d77db4568df0 **SHA256:** 44950952892d394e5cbe9dcc7a0db0135a21027a0b f937ed371bb6b8565ff678

Fortinet detection name: W32/Injector.ZVR!tr

Version 2.08

MD5: b4d37eff59a820d9be2db1ac23fe056e **SHA256:** 92d25f2feb6ca7b3e0d921ace8560160e1bfccb0bee b6b27f914a5930a33e316 **Fortinet detection name:** W32/Tepfer.ASYP!tr.pws

Version 2.09

MD5: 3f2762d18c1abc67e21a7f9ad4fa67fd **SHA256:** 2f44d884c9d358130050a6d4f89248a314b6c02d40 b5c3206e86ddb834e928f6

Fortinet detection name: W32/BLDZ!tr

Version 2.10

MD5: fb0a6857c15a1f596494a28c3cf7379d

SHA256: 73802eaa46b603575216fb212bcc18c895f4c03b47c 9706cde85368c0334e0cd

Fortinet detection name: W32/Malicious_Behavior.VEX

Editor: Martijn Grooten

Head of Testing: Peter Karsai

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Ionut Räileanu, Chris Stock

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

© 2018 Virus Bulletin Ltd, The Pentagon, Abingdon Science Park, Abingdon, Oxfordshire OX14 3YP, England Tel: +44 (0)1235 555139 Email: editor@virusbulletin.com Web: https://www.virusbulletin.com/