Andrea Allievi
Senior Security Researcher

26/10/2013

Sirefef under microscope: The clever ZeroAccess evolves again!

Last month in our labs we dealt with some machines infected with new Sirefef virus (dropper dated September 2013). I personally reversed new droppers (even the July 2013 one) and finally wrote the remover for my Company. We have observed a lot of clever tricks exploited by this virus. Here in this analysis, we are going to describe the majority of them. Sirefef authors are the same as the old ZeroAccess rootkit that, even 2 years ago, employed very nasty strategies to infect a target system and remain undetected...

In the beginning was ZeroAccess

The first sample of ZeroAccess rootkit is dated in fall of the year 2009. In that year the infection was very innovative because of the AV killing feature and nasty rootkit behaviour used to hide its presence in the target system. I will not cover here all Zeroaccess evolutions: I would like to outline only the previous analysis completed by my colleague Marco Giuliani (Saferbytes CEO) and me:

- ZeroAccess – an advanced kernel mode rootkit
- ZeroAccess Gets Another Update
- ZeroAccess APC: My First blog post (an article in Italian language posted on my blog)

ZeroAccess rootkit has evolved over the years and lost its rootkit part. The current release is called Sirefef and it has become a very nasty virus.

First stages of decryption

The first dropper analyzed is dated July 2013 and is encrypted 3 times. The first decryption is quite easy to overcome: it does few memory transfers and finally calls decrypt routine. First decrypt routine employ some math and compare instructions to end its job. The brand-new decrypted buffer is a PE file that needs to be relocated. After the relocation and IAT resolution is done, the execution control is transferred to PE decrypted executable. This Pe doesn't do anything interesting: Starting from execution thread TEB, it resolves some Kernel32 API addresses (beginning from the classical *GetProcAddress*) and builds a simple IAT. Then it starts to decrypt, map and relocate a final PE file in its address space. Execution control is returned to latter PE file that is actually mapped in memory.

Here is where things become interesting. The last decrypted Pe begins execution, checking if it is a classical executable or a Dll. In the first case, it does the following:

1. Obtains weather the process is launched under a Wow64 environment (exploiting *ZwQueryInformationProcess* native API, with *ProcessWow64Information* parameter)
2. Detects if a debugger is attached to current process
   - If a debugger is attached, Sirefef employs a strange behavior: it loads "untfs.dll" library and tries to resolve the function with ordinal 0x2302. If it succeeds, it calls resolved function. A careful researcher can suddenly identify that something is suspect. If the reader analyses "untfs.dll" system file, he can realizes that this library doesn't export any function with ordinal 0x2302. We will see what happens here afterwards.
   - Otherwise, if a debugger is not present, Sirefef calls its main stage function (called by the authors "MainZaDbgFunc"). This latter one creates a Debug object and re-launches itself as a

debugged process (*CreateProcess* API with DEBUG_PROCESS flag). Then it creates a user-mode thread that will be used to communicate with the debugged process via a mailslot (named "*uewuyew**ID***" where "ID" represent Sirefef process ID formatted in 8 hex digits).
Finally the debug cycle begins. Sirefef now waits and processes each debug event generated by debugged process.

Go beneath protection: discovering real Sirefef code

At this stage we have two Sirefef processes born from the same image file. The first process is the debugger, which controls the second Sirefef process: the debugged one. The debugger cycle analyses every event that debugged generates.
When the debugged process is first created, the debugger receives CREATE_PROCESS_DEBUG event. The Sirefef code that manages this event first queries basic information about process (PEB, image file handle, and so on...). It queries each DLL loaded by the debugged process (LOAD_DLL_DEBUG event, the code that manages the latter event is very similar) and adds it to an internal list. If name of the module loaded by debugged process equals to "untfs.dll", then Sirefef jumps to a platform-dependent procedure that enables trap-flag in target debugged thread context. Afterwards, execution of the debugged process is resumed. The enabled trap flag causes the debugged to generate another event, after the original "untfs.dll" section object has been created: EXCEPTION_DEBUG. The code that manages this event first retrieves target thread list entry, and then checks for three kind of exception: DBG_CONTROL_C, STATUS_BREAKPOINT and STATUS_SINGLE_STEP.

- Ctrl+C exception is used to pass calculated volume MD5 between the debugger and debugged process
- Software breakpoint exception is actually not used. Execution indeed is resumed with DBG_EXCEPTION_HANDLED code.
- Things gets interesting in "Single step" exception management code. We will see this in a while...

```
AnalyzeDbgException proc near           ; CODE XREF: MainZaDebuggerFunc+195↓p
                push    ebp
                mov     ebp, esp
                sub     esp, 0Ch
                push    ebx
                push    esi
                mov     esi, eax        ; ESI = Win32 DEBUG_EVENT structure
                xor     ebx, ebx
                cmp     [esi+5Ch], ebx
                jnz     short ProtectOk
                push    dword ptr [esi+0Ch] ; ExitStatus
                push    dword ptr [edi+18h] ; ProcessHandle
                call    ds:ZwTerminateProcess

ProtectOk:                              ; CODE XREF: AnalyzeDbgException+F↑j
                push    dword ptr [esi+8]
                mov     ecx, edi        ; ECX = ZA_CP_EVENT struct
                call    SearchThrByIdInList
                mov     [ebp+lpCurThrEntry], eax
                mov     eax, [esi+0Ch]  ; EAX = Exception code
                cmp     eax, DBG_CONTROL_C
                jz      CtrlCExc
                cmp     eax, STATUS_BREAKPOINT
                jz      BreakPointExc
                cmp     eax, STATUS_SINGLE_STEP
                jnz     ContinueDbg
                lea     esi, [edi+20h]
                cmp     [esi], ebx
                jz      ContinueDbg
                cmp     [ebp+lpCurThrEntry], ebx
                jz      short DecryptError
                lea     eax, [ebp+SectHandle] ; Here we are processing Single step exception
                push    eax             ; Extract encrypted PE and create memory section
                call    CreateMemSecAndDecryptPe
                test    eax, eax
                jz      short DecryptError
00000DBA 004019BA: AnalyzeDbgException+5E
```

s

Figure 2. Sirefef debugger code that process EXCEPTION_DEBUG event code

Single Step Management code

Single step exception management code begins with "*CreateMemSecAndDecryptPe*" procedure: it allocates a large buffer with *LocalAlloc* API, then it begins to decrypt 133 KB of data starting at offset 0xD0 of *_rdata* section. Encryption is again based on math operations (I haven't investigated too much on this encryption algorithm). "*CopyAndRemapPeInMemSect*" procedure exploits native APIs to create and map an unnamed memory section in current process address space, as big as the just allocated buffer. It copies the entire decrypted data (we will call this data the clean Sirefef PE file), perform IAT fixups, and finally it unmaps the latter memory section (but doesn't close it - section handle will be used as reference). At this stage **memory section of original "untfs.dll"** is **unmapped** from debugged process (exploiting `ZwUnmapViewOfSection` native API). Sirefef clean PE is then mapped at the **same virtual address** of old original library. This is one of the tricky features of Sirefef. Furthermore, Sirefef changes target debugged thread context, setting EAX register to 0x40000003. This value equals to `STATUS_IMAGE_NOT_AT_BASE` code. Why this behavior?

Here we are seeing one example of the large knowledge of the Sirefef authors about Nt architecture. Indeed, after we have reversed a big slice of NTDLL code, we saw that *LdrLoadDll* procedure, called every time an executable implicitly or explicitly (via *LoadLibrary(Ex)* API) loads a DLL, exploits *LdrpMapViewOfSection* to map the DLL section object just-created with the aid of *ZwCreateSection*. Our reader should know that every executable module loaded by Windows is shared between processes by "Section" objects. *LdrpMapViewOfSection* in turn uses ZwMapViewOfSection to do the actual map. The latter native interface every time is called to map a section object created with `SEC_IMAGE` attribute, if it

hasn't been able to map image section object at its preferred base address (and the PE image file buffer is not flagged with `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE`), it returns to the caller with the warning status code `STATUS_IMAGE_NOT_AT_BASE`. This behavior is implemented in Windows kernel: a large call chain starts from *NtMapViewOfSection* routine and ends at *MiMapViewOfImageSection.* The latter one returns a warning code to the caller every time it detects an image section not mapped at its preferred base address.

Now that we have understood what `STATUS_IMAGE_NOT_AT_BASE` means, one question suddenly arises: How this return code affects the *LdrpLoadDll* mapping routine? *LdrpFindOrMapDll* (internal procedure called from *LdrLoadDll*) checks the return address of *LdrpMapViewOfSection*, in case of an "Image not at base" warning status, code is diverted, and *LdrpRelocateImage* is called to do the actual relocation process. If a relocation table is found, the target PE buffer is processed and relocated, otherwise a `STATUS_CONFLICTING_ADDRESSES` error is returned and DLL load process will be broken (indeed it means that there are 2 modules that want to be loaded at the same addresses. The last one can't be relocated: Nt loader can't proceed).

```
                mov     [ebp+secSize], al
                lea     eax, [ebp+ViewSize]
                push    eax             ; ViewSize
                lea     eax, [ebp+ImageBase]
                push    eax             ; BaseAddress
                push    dword ptr [ebp+secSize] ; secSize
                push    [ebp+AllocationType] ; imageFullPath
                push    [ebp+String1.Buffer] ; imageName
                push    [ebp+SectionHandle] ; SectionHandle
                call    _LdrpMapViewOfSection@24 ; LdrpMapViewOfSection(x,x,x,x,x,x)
                mov     edi, eax
                xor     esi, esi
                cmp     edi, esi
                jl      MapError
                ......
                mov     esi, STATUS_IMAGE_MACHINE_TYPE_MISMATCH
                cmp     ebx, esi
                jz      MachineMismatch
                mov     eax, [ebp+pNtHdr]
                mov     ecx, 2000h
                test    [eax+16h], cx
                jz      short loc_77F20F42
                or      [ebp+var_8], 4
                cmp     ebx, STATUS_IMAGE_NOT_AT_BASE
                jz      ImageNotAtBase

                                        ; CODE XREF: LdrpFindOrMapDll(x,x,x,x,x,x)
                                        ; LdrpFindOrMapDll(x,x,x,x,x,x)+1E0F↑j ...
                mov     eax, _NtdllBaseTag
                push    78h             ; Size
                ......
ImageNotAtBase:                         ; CODE XREF: LdrpFindOrMapDll(x,x,x,>
                push    dword ptr [ebp+secSize] ; PVOID
                lea     ecx, [ebp+Destination]
                push    ecx             ; moduleFullPath
                push    eax             ; pNtHdr
                push    [ebp+ViewSize]  ; dwSize
                push    [ebp+ImageBase] ; ImageBase
                call    _LdrpRelocateImage@20 ; LdrpRelocateImage(x,x,x,x,x)
                mov     edi, eax
                test    edi, edi
                jge     RelocSuccess
                jmp     Error
0001EBD0 77EDF7D0: LdrpFindOrMapDll(x,x,x,x,x,x)-3F951
```

Figure 3. A snap of LdrpFindOrMapDll code. The reader can see that return code of LdrpMamViewOfSection is checked, and if it equals to STATUS_IMAGE_NOT_AT_BASE it proceeds to relocate PE buffer

Sirefef authors knew this fact, and saved developing time transferring “relocation efforts” to the Windows kernel code. At this stage, the debugged process is resumed and the execution controls is finally transferred to Sirefef pure clean code (in debugged process). From now on, the Debugger is not involved in the infection...

Sirefef pure clean code

In our analysis, we are going to speak about the infection dated September 2013, not the July one, because there are already some available analysis on Web. I don’t want to repeat what others security researchers have already done.
We will start from the Sirefef September clean code entry point. For this release, the authors have changed the infection logic. While in the previous release they replaced two COM objects (CD Burning ShellFolder and Windows WMI Helper Dll) with the aim to auto-load infection every time the workstation boots, now they use a more standard way: one Windows service and one link to the classical “Run” registry key.

Sirefef clean code is a small routine that does basic things: first of all, it retrieves some basic information like the current OS version, whether the process is under Wow64 environment and current startup command line. If no command line argument is found, it means that Sirefef has to be installed. *InstallSirefef* routine starts creating its setup event named “`{0C5AB9CD-2F90-6754-8374-21D4DAB28CC1}`”. If an instance of setup event already exists, the process is immediately terminated with *ExitProcess* API (it means that another installation is still in progress). Otherwise it calls *InjectZaDll* to inject a Sirefef dll to “explorer.exe”. *InjectZaDll* locates and extracts the right Sirefef compressed DLL (aPlib compression type), based on the current Os platform (AMD64 versus standard x86), in a buffer large enough to contain it. The memory buffer content containing DLL is subsequently processed (IAT is partially resolved, only for those modules already loaded in the target process), relocated and copied in a section object mapped even in the target address space. Finally, the section object is unmapped from Sirefef process (only a copy mapped in the target process remains) and a thread is crated in the target process address space exploiting *RtlCreateUserThread* native interface. This just-born thread begins execution in Sirefef DLL code entry point. We will take a glance at what this DLL does afterwards...

Let’s now return to Sirefef main process setup routine...
“*CreateGoogleEnv*” routine does the actual setup process: first, it enables *SeRestorePrivilege* in the current process token and builds a particular security descriptor that has “Anonymous logon” as owner and a DACL that allows every access type except `FILE_LIST_DIRECTORY` to “everyone” group. It then queries Local AppData directory reading its location from registry (“HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders” key) and starts to build the Sirefef main path (local user security context). Here we see another clever trick employed by Sirefef authors. We know indeed that the native layer of Windows operation system can operate correctly with every Unicode character, while user Win32 layer do not. Sirefef builds 2 paths based on the setup phase:

The first one is the local user path, and is used to store those files that run in a security context of current user. The path is formatted as follows:

“%localappdata%\Google\Desktop\Install\{generated GUID}\<3 randUchr>\
<3 randUchr>\< RTL Chr – 2 randUChrs>\{generated GUID}\”

Where:

- “%localappdata%” is the Local user application data directory (“C:\Users\Smith\AppData” for example)

- "{generated GUID}" is a Sirefef generated directory name based on system volume MD5, like "{0031a25b-264f-8e00-2b10-0595f10fed4c}" for example. We don't describe here the algorithm used for its generation because it is the same used in previous versions of this virus
- "randUchr" denotes a Unicode character that is outside the standard ASCII table, like "☠" for example. Indeed a Unicode char is represented as a 16 bit word value, and it could be 65536 different characters, many of those are not supported by Win32 layer of operation system
- "<RTL Chr>" is a special unicode character that is used by languages written in a right to left direction (like Arabic). In this way the text that follows this special symbol is written starting from the right position, ending in the left position...

The second path becomes the Sirefef main working directory, and is generated in the second phase of the setup procedure. It is formatted like the first one with a slight variation:

"%ProgramFiles%\Google\Desktop\Install\{generated GUID}\
<3 space chars>\...\<randUChr - RTL Chr – randUChr>\{generated GUID}\"

As the reader can see, the main Sirefef path uses forbidden characters: directory named "..." is not recognized by Win32 layer. As a matter of fact, if a user tries to open that path with Windows Explorer, it receives an error message like the following one:

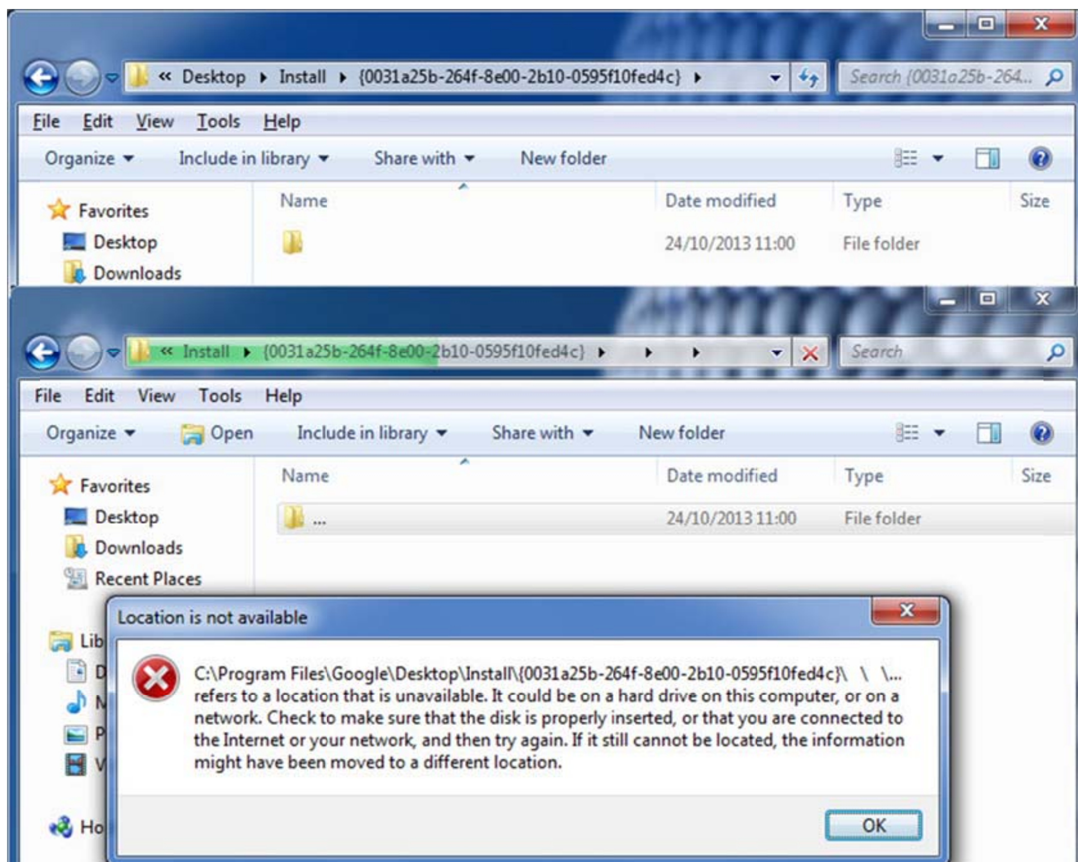


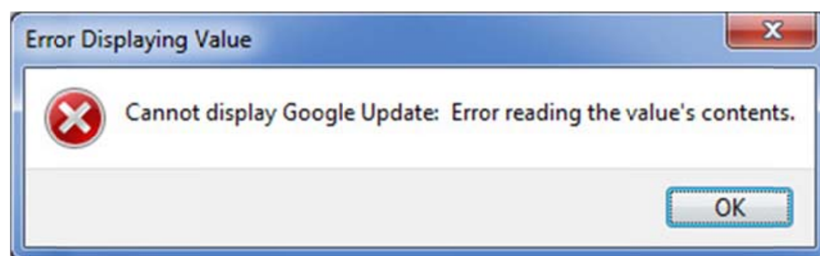
Figure 4. Error message received if a user tries to open Sirefef main path

Furthermore, applications like "Windows Command prompt" have some issues dealing with RTL char. The last directory name of Sirefef working paths is protected by the anonymous Security descriptor built at the start of setup function. In this way a user mode application can't easily open the last directory of Sirefef path.
After the "*CreateGoogleEnv*" procedure has created working path, it starts to generate Sirefef files. In particular, it copies itself in a file named "*GoogleUpdate.exe*", and generates its configuration file named

"@". Finally, it calls "*AddZaToSystemStartup*" or "*CreateZaService*" routine depending on the security context (local user, or system) to assure that it will be loaded at every system startup. First time, the function exploited is *"AddZaToSystemStartup"* (local user security context).

System reboot survival strategies

AddZaToSystemStartup is a simple procedure used by Sirefef to add itself in current user "Run" registry key. A new registry value is created with native APIs and its data is set to "Google Update.exe" executable (found in Sirefef local application data path) with a ">" argument as the command line. Noteworthy is that Sirefef fakes the name of its registry value: its name is "Google Update" terminated with a NULL char and 2 unicode characters (first is a RTL char, second is a random one). Exploited *ZwSetValueKey* native API accepts a unicode string that contains value name. The unicode string used by Sirefef has a length attribute value as big as the sum of string size, NULL char and trailing characters. In this way, if the user of infected machine tries to open Windows Registry Editor to navigate to infected key, it will receive the following message:



This is because Windows registry editor uses standard Win32 APIs. In particular *RegQueryValue(Ex)* Win32 function, accepts only NULL terminated strings. When the application enumerate registry values located in "Run" key, it receives only one part of Sirefef value name: "Google Update" NULL terminated string. Afterwards, when *RegQueryValue(Ex)* is called to query value data, it tries to convert the NULL terminated value name in a Unicode string structure, and it misses last characters. As a matter of fact, the internal *ZwQueryValueKey* call fails because it cannot locate a non-existing value name. This generates the error message.

After current user setup process is terminated, *CreateGoogleEnv* procedure closes all handles and exits. Execution flow returns to the *InstallSirefef* routine. It checks whether the current user setup phase has been completed with success, and, if so, it proceeds to the next phase: System setup. *"IsCurZaOrOldInstalled"* routine is called to verify if some old version of ZeroAccess is active in the target system. If so, it stops the setup process. Otherwise it tries to enable "Debug" privilege in the current process token (needed for "System" account impersonation). If it doesn't succeed, the "*SearchDebugPrivInToken*" routine opens and analyses process token. In particular, it looks if the linked token (used in UAC environments) has Debug privilege. If the user is an administrator and is under an UAC environment, Sirefef tries to bypass UAC exploiting the classical Adobe Flash installer trick (already described in few articles). If instead user is not an administrator, the setup process is ended. In this way Sirefef can hit even normal users (does the reader remember user setup process?).

After admin privileges have been acquired, the *"AdjustPrivsAndInstall"* routine enables *SeDebugPrivilege* and impersonates System Token with a precise algorithm:

1. Obtains processes and threads list with *ZwQuerySystemInformation* native API
2. For each process in the list it looks at the first that has *ThreadCount* and *InheritedFromProcessId* members not equal to 0. This process is always "*smss.exe*" that runs in a System security context.

3. Impersonates "*smss.exe*" security token, and enables the following privileges (exploiting *RtlAdjustPrivilege* Ntdll native Apis): SeMachineAccountPrivilege, SeTakeOwnershipPrivilege, SeRestorePrivilege, SeDebugPrivilege, SeSystemProfilePrivilege, SeSecurityPrivilege

```
AdjustPrivsAndInstall proc near

Enabled         = byte ptr -1

                push    ecx
                push    ecx
                push    esi
                lea     eax, [esp+0Ch+Enabled]
                push    eax             ; Enabled
                push    0               ; CurrentThread
                push    1               ; bEnabled
                push    14h             ; privIdx
                call    ds:RtlAdjustPrivilege ; Enable Debug privilege
                                        ; in current process token
                test    eax, eax
                jl      short Exit
                call    SearchAndImpersonateSystemThr
                test    eax, eax
                jz      short Exit
                push    14h
                call    SckIoFunc1
                call    DestroyAvProtections
                push    0
                call    InjectZaDll     ; Inject ZA DLL in "Services.exe" process
                mov     esi, eax
                test    esi, esi
                jz      short Error
```

Figure 6. A snap of "AdjustPrivsAndInstall" routine. The reader can see the enabling of Debug privilege, System Process token impersonation and the destroy protection function call

At this stage Sirefef thread runs in the "System" account security context, the most powerful account for a Nt environment. If the impersonation worked well, the "*DestroyAvProtections*" routine is called: its job is to terminate each AV process and to delete some System services like:

- mpssvc - Windows Firewall
- SharedAccess - Internet Connection Sharing (ICS)
- RemoteAccess - Routing and Remote Access
- PolicyAgent - IPsec Policy Agent
- Iphlpsvc - IP Helper
- Wscsvc - Security Center
- PcaSvc - Program Compatibility Assistant Service
- Bfe - Base Filtering Engine

Furthermore, another peculiarity of this function is that it makes "Windows Defender" and "Microsoft Security essentials" Security suites useless: for each file, it applies a security descriptor (exploiting *ZwSetSecurityObject* native interface) that has "System" account as owner and **an empty DACL** (no access to anyone), and a reparse point targeting "c:\windows\system32\config" file (exploiting "*ZwFsControlFile*" native API). In this manner, the user has no chance to open any file belonging to one of those security suites.

In this way Sirefef will be sure that it can safely control the target OS without any obstacles. The same Dll used in first setup phase is extracted and mapped in "*services.exe*" OS process and finally "*CreateGoogleEnv*" is called again, this time for phase 2 of the setup process, executed in a System Security context (setup phase number is passed to each functions as a parameter).

"*CreateGoogleEnv"* creates the Sirefef system working path (this time path is located in "Program files" directory) in the same way as seen before. It then copies and creates Virus files in it, and finally calls *CreateZaService* to create fake virus service.

The "CreateZaService" procedure will create Sirefef fake service. The term "fake" is used here to point out that a target service is created without the aid of the Service control manager **but** only using Ntdll native registry APIs. The service main key, named "<RTL>etadpug", and presented to infected users as "gupdate", thanks to its "right to left" char placed in the first position, is created in "HKLM\System\CurrentControlSet". Another clever trick is now exploited. *CreateZaService* also creates a "Parameters" subkey in ~~a~~the same strange way as seen before: its subkey name unicode string is built as "Parameters", NULL char, RTL char + 1 rand char.

```
0041B964  5C 00 53 00 65 00 72 00  76 00 69 00 63 00 65 00  \.S.e.r.v.i.c.e.
0041B974  73 00 5C 00 2E 20 65 00  74 00 61 00 64 00 70 00  s.\.. e.t.a.d.p.
0041B984  75 00 67 00 00 00 00 00  78 00 7A 00 10 B9 41 00  u.g.....x.z..¦A.
0041B994  50 00 61 00 72 00 61 00  6D 00 65 00 74 00 65 00  P.a.r.a.m.e.t.e.
0041B9A4  72 00 73 00 00 00 2E 20  64 27 00 00 1A 00 1C 00  r.s..... d'......
0041B9B4  94 B9 41 00 54 00 79 00  70 00 65 00 00 00 00 00  ö¦A.T.y.p.e.....
0041B9C4  08 00 0A 00 B8 B9 41 00  53 00 74 00 61 00 72 00  ....©¦A.S.t.a.r.
```

Figure 7. Dump of Sirefef service strings. Reader can see service name (highlighted in orange color) and "Parameters" key Unicode strings

The handle to "Parameters" key is then closed (it was created only as an hurdle). *CreateZaService* ~~then~~ set up the following values in main service key: Start, Type, ErrorControl, ObjectName, Description, DisplayName, and finally ImagePath. The service image path is set to "*Google Update.exe*" main executable located in Sirefef system working path, with a trailing "<" character used as an argument. The strange format of "Parameters" key renders every attempt to destroy Sirefef service useless. Every Win32 application (Windows SC utility, Registry editor, and so on…) that tries to destroy the service, will encounter the same sort of error we have already seen for "Run" current user key.

At this stage, the Setup process is **done**. All work is now transferred to Sirefef DLL injected in "explorer.exe" process (for current logged user security context), and in "services.exe" (for System security context).The dropper process sleeps for 1 second and then ends with a call to *ExitThread* API.

Sirefef Injected DLL

The Dll extracted from Sirefef clean code (aPlib compressed) is platform dependent. Sirefef indeed includes two versions: 32 bit and 64 bit. It begins its existence in a victim process. IAT is partially resolved because dropper has resolved only its dependent modules that are actually loaded in victim process. As a matter of fact, Dll entry point starts to resolve entire IAT, even for those modules not loaded in victim address space. It exploits native loader interfaces like *LdrLoadDll* and *LdrGetProcedureAddress*. In this way, it loads even those Dll that are actually not mapped in victim address space. When IAT is totally resolved, Sirefef Dll installs a vectored exception handler via RtlAddVectoredExceptionHandler, loads "lz32.dll" module and then **employs a new trick:** the objective here is to create a new thread to execute code from the infected module. This could be detected by AVs if start address of the thread is outside every loaded module registered in process PEB's loader data. Sirefef writes a trampoline in "lz32.dll" space, and finally it queues a work item in the victim process thread pool. At the end current thread is terminated with *RtlExitUserThread* .

When the Work item starts execution, Sirefef regains control of the victim process. Work item first checks whether it's executed under a user security context, and if so, it checks if "actioncenter" or

“wscntfy” modules are loaded. These modules are responsible for showing Action center Security related information to user. If one or both are found, Sirefef hooks their imported “*Shell_NotifyIconW*” API. New Sirefef *Shell_NotifyIcon* procedure transform each “Message” passed as first parameter in **NIM_DELETE**. The results are that each notify icon requested by the Action center is deleted and icon is not showed.

```
za_Shell_NotifyIconW proc near          ; DATA XREF: HookShellNotifyIconW+AB↓o

lpdata           = dword ptr  8

                 push    [esp+lpdata]
                 push    2               ; NIM_DELETE
                 call    g_lpOrgShellNotifyIconW ; CALL original
                 retn    8               ; Shell_NotifyIconW(NIM_DELETE, lpdata)
za_Shell_NotifyIconW endp               ; and return
```

Figure 8. Sirefef simple new “Shell_NotifyIconW” function

Sirefef main path (local user path or system path depending on security context) is retrieved with the same algorithm seen at the beginning. Then the work item code creates one of Sirefef main events based on current security context:

- System security context, event name will be: “`{A3D35150-6823-4462-8C6E-7417FF841D77}`”
- Local user security context, event name will be: “`{A3D35150-6823-4462-8C6E-7417FF841D78}`”

If all worked fine, Sirefef work item creates main DLL execution thread and then ends.

Main Dll thread does a lot of things. It starts waiting for the ending of the Setup process: it continuously opens Sirefef setup event, and, if exists, it closes its handle and sleeps for 4 seconds. The process is always repeated unless the same event will not exists anymore (*ZwOpenEvent* routine returns `STATUS_OBJECT_NAME_NOT_FOUND`). Sirefef Dll thread also opens its main System security context event and, if found it, repeats exactly the previous procedure: in this way the system security context Dll code and the current user security context Dll are synchronized.
At this stage Dll thread code calls “*CreateGacIniAndModifyWinsock*” procedure. This is a very important function because, if executed in Server systems, it generates a lot of problems (I have indeed called this feature “The Sirefef Server Hell”).

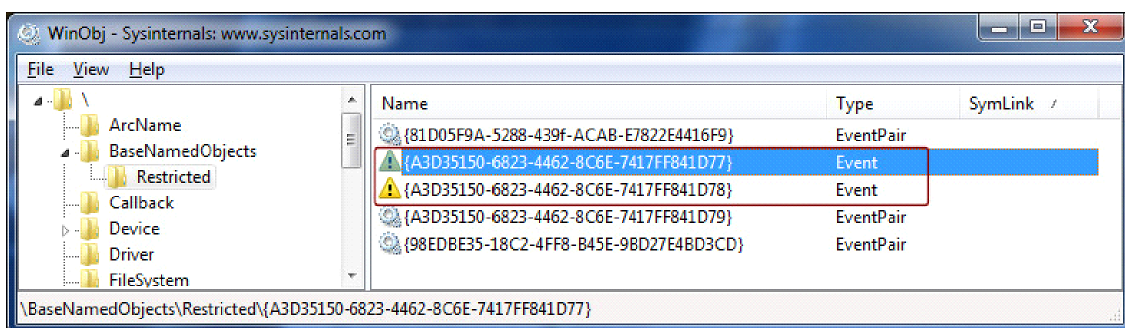


Figure 9. Sirefef main synch events: system security context event and local user security context event

CreateGacIniAndModifyWinsock and the “Sirefef Server Hell”

This function is a very nasty one. It begins to search whether it runs under System security context, and if not, it tries to impersonate the System token. If it succeeds, it calls “*CreateAssemblyDirs*” to verify and create (if needed) “GAC” and “GAC_MSIL” directories under “%systemroot%\Assembly”. The original “desktop.ini” file located in these 2 folders is deleted. *ZaModifyWinsock* function code is now executed: it opens “HKLM\SYSTEM\CurrentControlSet\Services\WinSock2\Parameters” Winsock key and launches

Winsock catalogs modification routine. This routine infects Winsock; and is called 2 times: one for Namespace catalogs, and another one for Protocol catalogs.
ModifyWinSockSubkey function indeed accepts two parameters: name of the registry value where to obtain the name of current catalog and address of the parsing procedure. The parsing procedure instead requires three parameters:

1. Total number of entries in current catalog;
2. Name of the entry key format string ("Catalog_Entries\\%012u");
3. Handle to catalog root registry key

PROTOCOL CATALOGS INFECTION
ModifyWinSockSubkey is first called with the following parameters: "Current_Protocol_Catalog" registry value name and Protocol Catalog parsing procedure. It first retrieves the name of current protocol catalog and opens the corresponding "Parameters" sub-key (in our test system sub-key was named "Protocol_Catalog9"). Then it queries the "Num_Catalog_Entries" value from last opened key, with the aim to find the right number of entries in current protocol catalog.

Sirefef protocol catalog parsing procedure then starts execution. For every entry in current protocol catalog, it does the following:

1. Reads data of "*PackedCatalogItem*" value and checks if data is a binary type and its size is bigger or equal to 0x128 bytes
2. Checks if GUID of catalog entry, found at offset 0x118 of read data, equals to a well-known GUID:
   - Ipv4 catalog entry - `{E70F1AA0-AB8B-11CF-A38C-00805f48A192}`
   - Ipv6 catalog entry - `{F9EAB0C0-26D4-11D0-BFBB-00AA006C34E4}`
   - NetBios catalog entry - `{8D5F1830-C273-11CF-C895-00805F48A192}`
   - QoS catalog entry - `{9D60A9E0-337A-11D0-88BD-0000C082E69A}`
   - Unknown catalog entry - `{9FC48064-7298-43E4-BDB7-181F2089792A}`
3. If GUID matches, it zeroes out the first 0x100 bytes of read binary data and copies "mswsock.dll" ANSI string inside it (starting at offset 0). Otherwise, it moves to next entry

NAMESPACE CATALOGS INFECTION
ModifyWinSockSubkey is called the second time with the following parameters:
"*Current_NameSpace_Catalog*" registry value name and Namespace Catalog parsing procedure.
As before, it first retrieves the name of current namespace catalog and opens the corresponding "Parameters" sub-key (in our test system sub-key was named "NameSpace_Catalog5"). As usual it retrieved the exact number of catalog entries and finally calls the namespace parsing procedure. For every entry in the current namespace catalog it does the following:

1. Reads data of "ProviderId" registry value and checks if data is binary type and its size is equal to 0x10 (size of a GUID). This value contains current namespace entry GUID.
2. Checks if read data equals to one the following GUIDs:
   - Tcp/Ip namespace provider GUID - `{22059D40-7E9E-11CF-5AAE-00AA00A7112B}`
   - NLA (Network Location Awareness) namespace provider GUID - `{6642243A-3BA8-4AA6-A5BA-2E0BD71FDD83}`
3. If current entry GUID matches, Sirefef substitutes "LibraryPath" value data of current entry with "mswsock.dll" string.

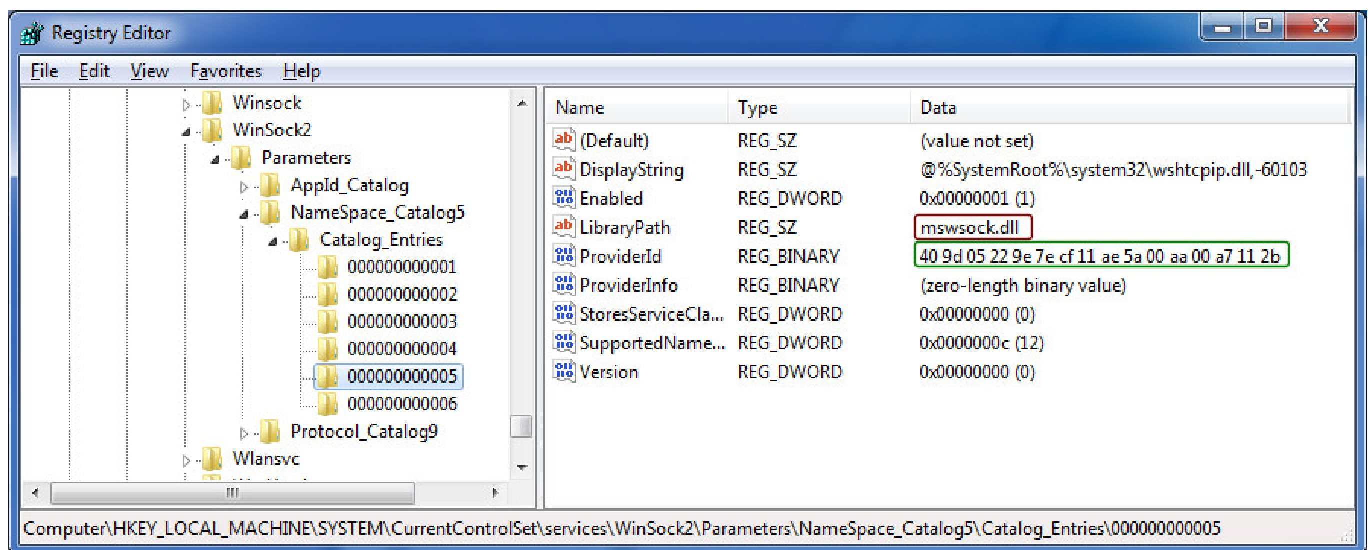


Figure 10. Snap of windows Registry editor showing infected Winsock namespace catalogs

WHY MSWSOCK.DLL?
At this stage *ZaModifyWinsock* function ends and returns execution control to *CreateGacIniAndModifyWinsock*. The reader is maybe wondering why Sirefef has done this strange behavior… Indeed "mswsock.dll" is a clean Windows library, right?

Here we are seeing the last clever Sirefef trick: *CreateAndMapFakeWSock* function is the **key**. It extracts and writes a fake "Winsock Dll" (included uncompressed in Sirefef Dll file, stored in its ".rdata" section) in "%systemroot%\assembly\GAC\desktop.ini" file (does the reader remember that original "desktop.ini" file was deleted before, right?). It then creates a file section object named "mswsock.dll" and places it in "KnowDlls" object manager directory. As the reader already knows, this directory includes memory sections of every system known library. Windows loader, with the help of this directory, when maps a known Dll in a target process, it avoids reading its file from solid storage, and saves a lot of time.

Windows loader, from now on, when is called to map original "mswsock.dll" library in a particular process address space, will find a ready memory section object of that library, and maps that one instead of the original one. This faked library has an export table containing every original "mswsock" exported entry, all forwarded to the original library, except for "WSPStartup". Sirefef *WSPStartup* function implement entire faked dll logic.
Due to limited available time, I have not reversed entire faked "mswsock" library, but I know for sure that this library acts as a filter and causes a lot of problems to all listening ports of a Server systems (like Windows Server 2008 and 2012). We actually don't know why, but If a curious reader would like to investigate this fact, he can write me a mail (address andrea.allievi@saferbytes.it).

I have built a short video that highlights this problem on a Windows Server 2008 R2 system:
http://www.andrea-allievi.com/files/Sirefef_2013_Server_Hell.avi

CONCLUSIONS

In this analysis, we have placed Sirefef under a microscope. We have seen that the people behind this infection are very smart and have large knowledge of the Windows Operation systems internal architecture.

Due to limited resources and available time, I have not ended the entire reverse engineering and program comprehension of this malware. If the reader is interested in it or if he has some questions or opinions, just send me a message at my personal mailbox: info@andrea-allievi.com. Maybe if we have many requests, I can continue this job on an eventual part 2.

Furthermore, I would like to thank my company, especially Marco Giuliani, for having sponsored this work making the analysis possible, as well as KernelMode.info community, for providing me a lot of useful information and fresh droppers. You are great, guys! ☺

Last revision: 27/10/2013

Andrea Allievi
Senior Security Researcher