CHECKPOINT.COM

CHECK POINT RESEARCH

PUBLICATIONS TOOLS ABOUT US CONTACT US

SUBSCRIBE UNDER ATTACK?

Search IPS Protections, Malware Families, Applications and more...

SiliVaccine: Inside North Korea's Anti-Virus

May 1, 2018

By: Mark Lechtik and Michael Kajiloti

Revealed: In an exclusive piece of research, Check Point Researchers have carried out a revealing investigation into North Korea's home-grown anti-virus software, SiliVaccine. One of several interesting factors is that a key component of SiliVaccine's code is a 10-year-old copy of one of Trend Micro's, a Japanese company, software components.

Background

The journey began with our research team receiving a very rare sample of North Korea's SiliVaccine anti-virus software from Martyn Williams, a freelance journalist with a focus on North Korean technology.

On July 8th 2014 Mr. Williams had himself received the software as a link in a suspicious email sent by someone going by the name of 'Kang Yong Hak', who's mailbox has since been rendered unreachable. Upon taking a closer look, our team was able to uncover several interesting elements.

The strange email sent by 'Kang Yong Hak', supposedly a Japanese engineer, contained a link to a Dropbox-hosted zip file that held a copy of the SiliVaccine software, a Korean language readme file instructing how to use the software and a suspicious looking file posing as a patch for SiliVaccine.

Trend Micro's AV Scan Engine

After detailed forensic analysis of SiliVaccine's engine files, our team discovered exact matches of SiliVaccine and large chunks of anti-virus engine code belonging to Trend Micro, a completely separate Japan-based provider of cyber security solutions. Furthermore, this exact match coding had been well hidden by SiliVaccine's authors. With Trend Micro being a Japanese company, and Japan and North Korea enjoying no official diplomatic or political relationship, this is a surprising discovery.

Of course, the purpose of an anti-virus is to block all known malware signatures. However, a deeper investigation into SiliVaccine found that it was designed to overlook one particular signature, which ordinarily it would be expected to block, and which is blocked by the Trend Micro detection engine. While it is unclear what this signature actually is, what is clear is that the North Korean regime does not want to alert its users to it.

Bundled Malware

Also found to be included in the SiliVaccine software that Marytn received was the JAKU malware. This was not necessarily part of the anti-virus but could be targeted towards journalists like Mr. Williams.

In brief, JAKU is a highly resilient botnet forming malware that has infected around 19,000 victims, primarily by malicious BitTorrent file shares. It has however been seen to target and track more specific individual victims in both South Korea and Japan, including members of International Non-Governmental Organizations (NGOs), engineering companies, academics, scientists and government employees.

Our investigation found though that the JAKU file was signed with a certificate issued to a certain 'Ningbo Gaoxinqu zhidian Electric Power Technology Co., Ltd', the same company that was used to sign files by another well-known APT group, 'Dark Hotel'. Both JAKU and Dark Hotel are thought to be attributed to North Korean threat actors.

The Japanese Connection

As well as the initial email containing the copy of North Korea's anti-virus coming from a claimed Japanese sender, there were other connections with Japan found by our researchers.

During our investigation, we discovered the names of the companies that are thought to have authored SiliVaccine, two of which are PGI (Pyonyang Gwangmyong Information Technology) and STS Tech-Service.

Underlying these Japanese connections, however, is the non-relationship between Japan and North Korea, who are enemies with no official diplomatic relations.

STS Tech-Service seems to be a North Korean establishment, it has previously worked with other companies, including those by the names of 'Silver Star' and 'Magnolia', both of which are based in Japan.

Conclusion

This revealing exploration into SiliVaccine may well raise suspicions of authenticity and motives of the IT security products and operations of this Hermit Kingdom.

While attribution is always a difficult task in cyber security, there are many questions raised by our findings. What is clear, however, are the shady practices and questionable goals of SiliVaccine's creators and backers.

Below are the full technical details of the investigation.

———————————————————————————————————————

Contents:

Architecture and Overview

SiliVaccine has the structure of a rather classical Anti-Virus. It consists of both user and kernel mode components, all working together for the purpose of scanning files or memory against a given repository of static malware signatures (known as the pattern file). Generally, scans can be done on demand or in real time, each constituting a separate component to handle the scanning action accordingly. Either way, all are carried out by a single logic implemented in the core of the product, known as the engine.

The figure below gives a general overview of the various software elements that comprise SiliVaccine and their interactions, followed by an outline of each of the components and its purpose:

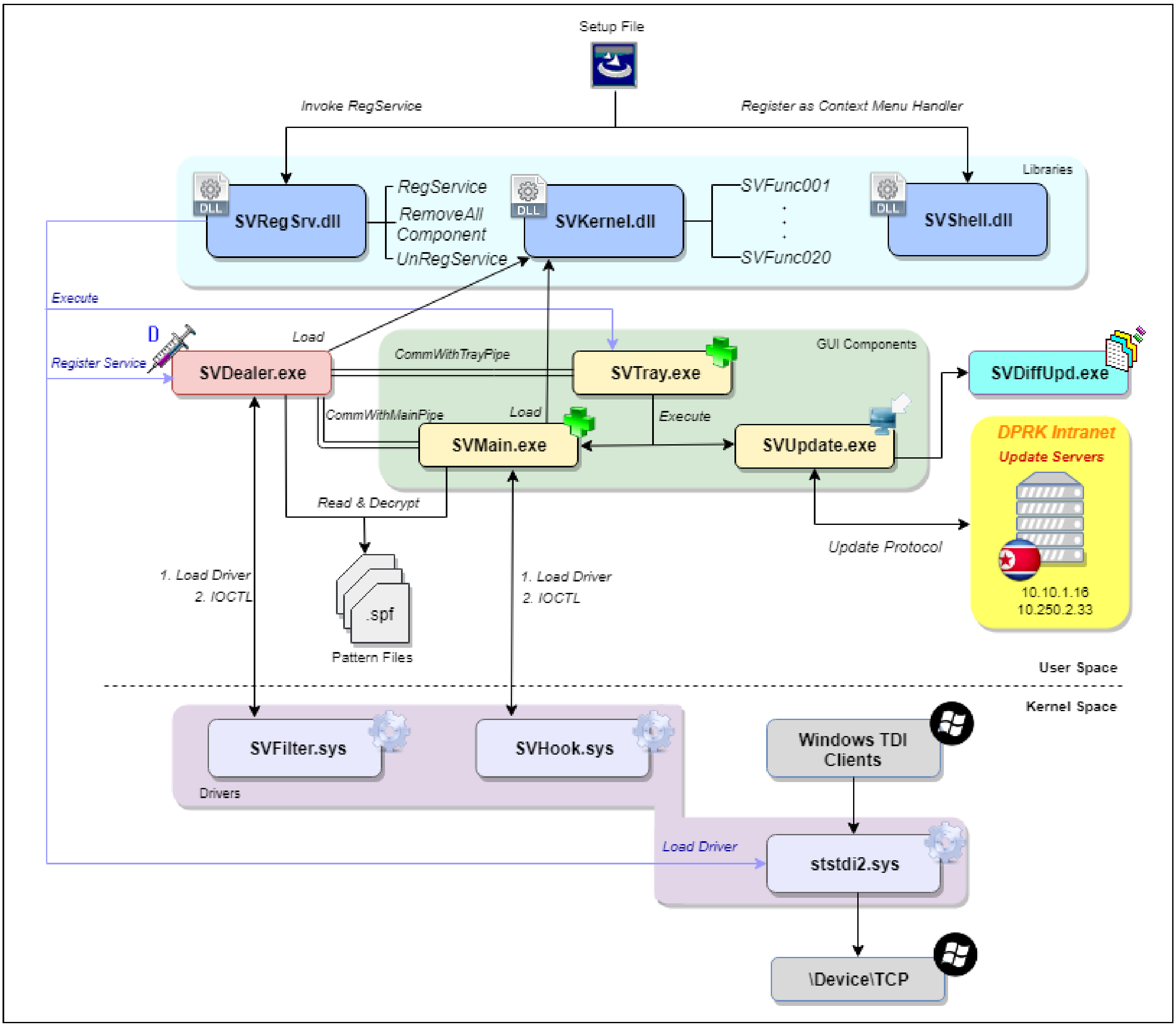


Figure 1: SiliVaccine's architecture.

Libraries

- SVKernel.dll – the file scanning engine, the most substantial and important component of the AV. It has the core functionalities of detecting a file type, parsing it and going over its sections to locate any malicious indicators. The latter come in the form of signatures, which are designated for the engine and describe various properties that identify malware families or samples. These are stored as a group of 34 files – ranging from *SVPatt00.spf* to *SVPatt33.spf,* and can be read and interpreted by the engine during run time. All of the functions given by the engine are exposed through 20 DLL exports (*SVFunc001 – SVFunc020)*, which are in turn used by the executables that conduct the scanning tasks. A deeper inspection of both the pattern files and the engine itself appears in the upcoming sections.
- SVRegSrv.dll – part of the installation package, which is in charge of deploying or discarding some of the AV's components. The DLL gives 3 exports (apart from *DllMain*), each one taking a different course of actions:
  - *RegService*: invoked by SiliVaccine's MSI installer and used to activate the main components of the software. This is done by executing *exe,* registering *SVDealer.exe* as a service and loading *ststdi2.sys* as a driver. Apart from this, it creates a directory for quarantined files and scan logs, as well as creates elementary registry values that serve all the other components of the AV (e.g. the installation path, found in the *AppPath* key under *HKLM\SOFTWARE\STS Tech-Service\SVaccine*).
  - *RemoveAllComponent*: removes all previously installed components, i.e. executables, DLLs, quarantine directory, scan logs and registry keys.
  - *UnRegService*: Finds all relevant active windows with the names 'SiliVaccine 4.0' , 'SiliVaccine 4.0 Update' and 'SiliVaccine 4.0 Patch' and closes them by posting a WM_CLOSE window message. Additionally, it looks for the tray application by its window class ('SVTRAY') and shuts it down, unloads the *sys* and *SVFilter.sys* drivers and deletes the *SVDealer.exe* service.

Interestingly, the last 2 functions are never called, since there is no uninstallation utility for the software present amongst the installed files.

- SVShell.dll – a COM class serving as a shell extension, which is used to register a SiliVaccine context menu handler (as described in the figure below). This handler allows the scanning of particular files upon clicking the right mouse button; however no actual action takes place when doing so.

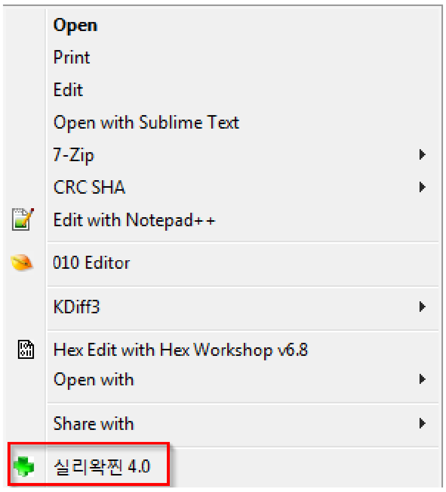


Figure 2: SiliVaccine's context menu entry.

Service

- SVDealer.exe – user mode component registered as a service that is responsible for carrying out real-time scanning tasks. For this purpose, it loads a file system filter driver named *SVFilter.sys*, which intercepts various file system activities and signals the *SVDealer* component so as to trigger it to scan the files and report back with a verdict. The scanning itself is done by calling the corresponding export from *SVKernel.dll* (*SVFunc018*). A deeper explanation is given in the upcoming sections.

GUI Components

- SVTray.exe – the most essential GUI component appearing as a tray icon, which runs on the system constantly. The menu corresponding to the tray application (opened by right-clicking the icon) allows opening either the *SVMain.exe* or *SVUpdate.exe* components, or activating\disabling the real-time protection feature.
- SVMain.exe – the main window through which the user interacts with the software, which allows conducting manual scans on particular files\directories or changing various setting like file whitelisting, update scheduling etc. Also allows for the opening of the *SVUpdate.exe* utility.

Figures 3 & 4: SVMain GUI and SVTray menu.

- SVUpdate.exe – an update utility which contacts intranet servers within the DPRK so as to poll for either new versions of the binary executables or pattern files.
  The servers contacted can be hardcoded, in which case
  they are 10.10.1.16 \10.250.2.33, or manually configured in the *SVMain* component. The communication protocol between the AV client and the update server is based on HTTP, with some peculiar additions such as the '*User-Dealer'* field, which appears as an addition to the common '*User Agent*' field:

```
GET /silivaccineetc/?8a8f9b9e8b9ad0bc9091919a9c8b969091ad9a8e8a9a8c8b HTTP/1.1
Accept: */*
User-Dealer: SVUpdate
User-Agent: SVUpdate
Host: 10.10.1.16
```

Figure 5: Custom field appended to HTTP header during update request.

Each received update file is checked for integrity using a custom diffing tool called *SVDiffUpd.exe*.

Named Pipes

- CommWithMainPipe – used to pass the process ID of *SVMain to SVDealer*, which in turn passes it to the *SVFilter* driver via an IOCTL. The driver registers this number as a global variable (along with *SVDealer*'s PID). Upon access to any file on the system, the filter driver checks the PID of the associated process, and in case it's that of an AV component (that is – either *SVMain* or *SVDealer*), the access is permitted.
- CommWithTrayPipe – used to notify the tray application of a newly attached removable media. For this, *SVDealer* generates a bitmask of currently available drives with the use of the *GetLogicalDrives* function, and monitors it for changes. Once a change is detected, *SVDealer* issues an IOCTL to attach the file system filter driver to new drive, sets a global event named *RemovableMediaInsertEvent* to be signaled and passes the drive index as a bitmask to *SVTray* via the named pipe. In turn, the tray application, which monitors for the pipe data indefinitely, sets a global event named *MemoryScanEvent* to be signaled, and issues the new drive index bitmask as window message to the '*SVTRAY'* window.

Drivers

- SVFilter.sys – file system filter driver which communicates with the real-time protection service (*SVDealer.exe*). Some actions on the file system are intercepted so as to pass control to the aforementioned process for a prior scan. This will determine if the action should or should not take place, based on the scan verdict.
- SVHook.sys – a driver that interacts with *SVMain* upon a memory scan, in which case it provides various details related to the scanned process that are accessible solely through kernel mode.
- ststdi2.sys – a TDI driver which intercepts all TCP packets sent by system TDI clients. The driver logs all connections in a data structure and allows another entity to query it via IOCTLs.

It's noteworthy that some of the components mentioned above (namely: *SVKernel*, *SVDealer* and all of the GUI components) are protected with Themida and Unopix, well-known and effective packers. While the usage of Themida is quite common in commercial software, in this case it's somewhat puzzling; the reason for this being that it's hard to break and provides very solid mitigations against reverse engineering. Considering the fact that SiliVaccine has no competitors in the North Korean market, it is not clear why this software has to be so protected. The following section may provide a possible explanation for using it.

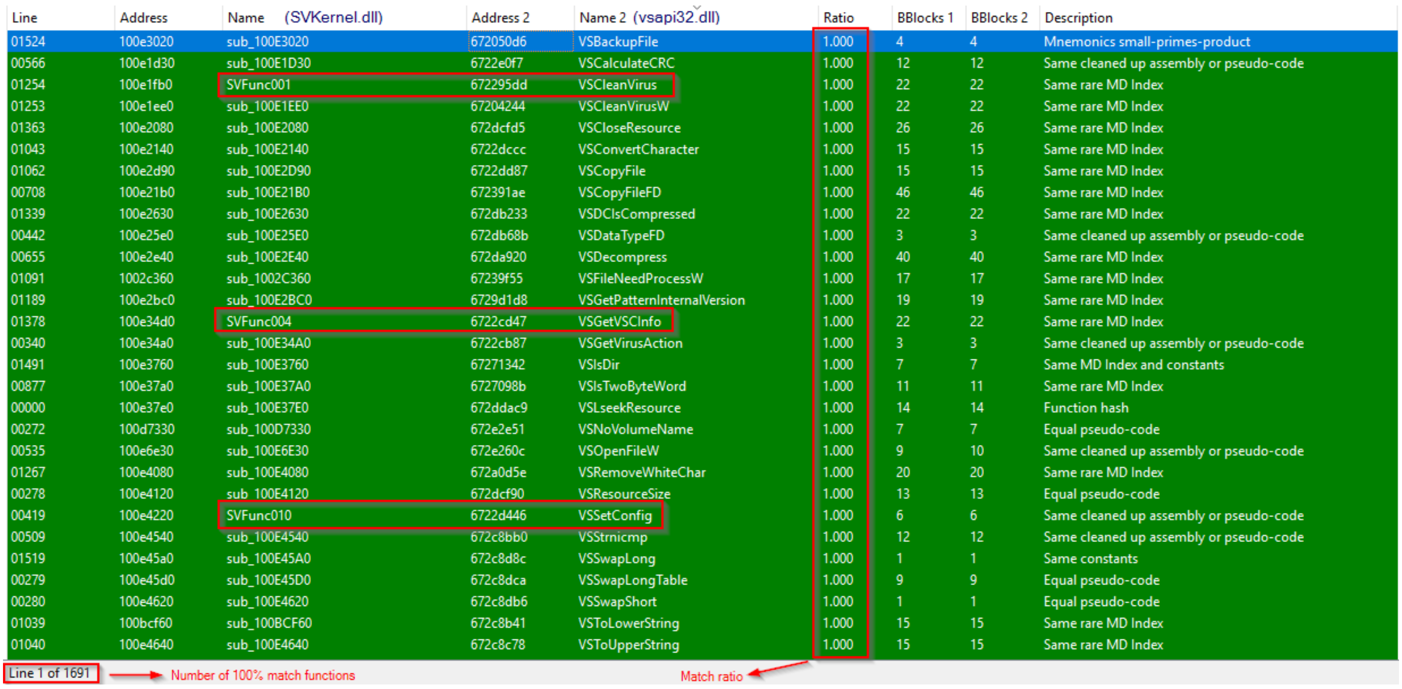
File Scanning Engine

One of the core elements of SiliVaccine is the scanning engine. This is in fact a DLL named *SVKernel.dll* which is responsible for carrying out a scan against a file or process memory so as to determine whether it's malicious or not. In the former case it will also provide the detection name (i.e. the name of the signature triggered for that file), based on a repository of signatures referred to as the pattern file. This is probably the most important component of this software, and is being loaded by other utilities of the AV that invoke the scans themselves.

As mentioned in the previous section, the engine binary (as well as several others) comes packed with Themida 2.x and Unopix 0.94, both efficient software protectors that allow virtualizing parts of the code and providing various reverse engineering mitigations. This makes the resulting installed binaries fairly hard to analyze. Nonetheless, we were able to unpack the samples as almost no part of the code was virtualized and very few protection features were enabled in the first place.

Looking at the unpacked binaries, we spotted strings that appeared in another file found on the internet named *vsapi32.dll.* As it turns out, this is a proprietary file scanning engine written by **Trend Micro**, a Japanese cyber security vendor manufacturing a range of AV solutions. Their engine serves the exact same purpose as the one found in SiliVaccine, and appeared highly similar to it.

In order to estimate the similarity between the two files we conducted binary diffing. Much to our surprise there is a great amount of code shared between them. In fact, as many as 1,691 functions had a 100% match based on very strong criteria (e.g. same function hashes, identical pseudo-code etc.). This finding is outlined in the following figure, which shows some of the results of the diffing process.



Line	Address	Name (SVKernel.dll)	Address 2	Name 2 (vsapi32.dll)	Ratio	BBlocks 1	BBlocks 2	Description
01524	100e3020	sub_100E3020	672050d6	VSBackupFile	1.000	4	4	Mnemonics small-primes-product
00566	100e1d30	sub_100E1D30	6722e0f7	VSCalculateCRC	1.000	12	12	Same cleaned up assembly or pseudo-code
01254	100e1fb0	SVFunc001	672295dd	VSCleanVirus	1.000	22	22	Same rare MD Index
01253	100e1ee0	sub_100E1EE0	67204244	VSCleanVirusW	1.000	22	22	Same rare MD Index
01363	100e2080	sub_100E2080	672dcfd5	VSCloseResource	1.000	26	26	Same rare MD Index
01043	100e2140	sub_100E2140	6722dccc	VSConvertCharacter	1.000	15	15	Same rare MD Index
01062	100e2d90	sub_100E2D90	6722dd87	VSCopyFile	1.000	15	15	Same rare MD Index
00708	100e21b0	sub_100E21B0	672391ae	VSCopyFileFD	1.000	46	46	Same rare MD Index
01339	100e2630	sub_100E2630	672db233	VSDClsCompressed	1.000	22	22	Same rare MD Index
00442	100e25e0	sub_100E25E0	672db68b	VSDataTypeFD	1.000	3	3	Same cleaned up assembly or pseudo-code
00655	100e2e40	sub_100E2E40	672da920	VSDecompress	1.000	40	40	Same rare MD Index
01091	1002c360	sub_1002C360	67239f55	VSFileNeedProcessW	1.000	17	17	Same rare MD Index
01189	100e2bc0	sub_100E2BC0	6729d1d8	VSGetPatternInternalVersion	1.000	19	19	Same rare MD Index
01378	100e34d0	SVFunc004	6722cd47	VSGetVSCInfo	1.000	22	22	Same rare MD Index
00340	100e34a0	sub_100E34A0	6722cb87	VSGetVirusAction	1.000	3	3	Same cleaned up assembly or pseudo-code
01491	100e3760	sub_100E3760	67271342	VSIsDir	1.000	7	7	Same MD Index and constants
00877	100e37a0	sub_100E37A0	6727098b	VSIsTwoByteWord	1.000	11	11	Same rare MD Index
00000	100e37e0	sub_100E37E0	672ddac9	VSLseekResource	1.000	14	14	Function hash
00272	100d7330	sub_100D7330	672e2e51	VSNoVolumeName	1.000	7	7	Equal pseudo-code
00535	100e6e30	sub_100E6E30	672e260c	VSOpenFileW	1.000	9	10	Same cleaned up assembly or pseudo-code
01267	100e4080	sub_100E4080	672a0d5e	VSRemoveWhiteChar	1.000	20	20	Same rare MD Index
00278	100e4120	sub_100E4120	672dcf90	VSResourceSize	1.000	13	13	Equal pseudo-code
00419	100e4220	SVFunc010	6722d446	VSSetConfig	1.000	6	6	Same cleaned up assembly or pseudo-code
00509	100e4540	sub_100E4540	672c8bb0	VSStrnicmp	1.000	12	12	Same cleaned up assembly or pseudo-code
01519	100e45a0	sub_100E45A0	672c8d8c	VSSwapLong	1.000	1	1	Same constants
00279	100e45d0	sub_100E45D0	672c8dca	VSSwapLongTable	1.000	9	9	Equal pseudo-code
00280	100e4620	sub_100E4620	672c8db6	VSSwapShort	1.000	1	1	Equal pseudo-code
01039	100bcf60	sub_100BCF60	672c8b41	VSToLowerString	1.000	15	15	Same rare MD Index
01040	100e4640	sub_100E4640	672c8c78	VSToUpperString	1.000	15	15	Same rare MD Index

Figure 6: Results of a binary diff between *SVKernel.dll* and *vsapi32.dll.*

From the figure above it's also evident that some of *SVKernel*'s export functions have an exact match to ones exported by *vsapi32*. As a matter of fact, at least 17 *SVKernel* exports are either precisely the same as ones in *vsapi32*, or are wrappers around such functions. The following is a table that shows the mapping we resolved between the two engines export functions, as well as few other ones that are written particularly for SiliVaccine:

SiliVaccine Function	Trend Micro Function	Description
SVFunc001	VSRemoveVirusW	Copies an infected file to a temp folder and attempts to clean it based on its format.
SVFunc002	VSDecompressFile	Decompresses a file.
SVFunc003	VSGetPatternPath	Sets the path for the pattern file in a scan configuration struct.
SVFunc004	VSGetVSCInfo	Initializes a scan information struct based on an internal configuration struct.
SVFunc005	VSInit	Initializes sub-structs and fields within the scan configuration struct.
SVFunc006	VSQuit	Carries out various cleanup actions of files and memory upon completion of a scan task.
SVFunc007	VSReadPatternInFile	Retrieves a particular pattern within some offset in the pattern file.
SVFunc008	VSSetCharacterEnvType	Sets a global that describes the character encoding used throughout the execution of the scan (e.g. for writing the log file).
SVFunc009	VSSetConfFlag	Updates a flag in the scan configuration struct.
SVFunc010	VSSetConfig	Updates several flags in the scan configuration struct.
SVFunc011	Unknown	Unknown
SVFunc012	calls VSSetConfFlag with particular arguments	-
SVFunc013	VSSetLogFilePath	Sets the path for the scan log file in a scan configuration struct.
SVFunc014	calls VSSetConfFlag with particular arguments	-
SVFunc015	calls VSSetConfFlag with particular arguments	-
SVFunc016	VSSetProcessFileCallBackFunc	Sets a callback function which is called upon scan completion. The callback address is passed as an argument and set in a scan configuration struct.
SVFunc017	calls VSSetConfFlag with particular arguments	-
SVFunc018	VSVirusScanFileW	Initializes a scan task for a particular filename, passed as argument.
SVFunc019 (SiliVaccine Proprietary Function)	-	Reads and decrypts the pattern file (based on the *.spf* files).
SVFunc020 (SiliVaccine Proprietary Function)	-	Unmaps and closes the pattern file handle.

The following figures demonstrate the similarity of code in several key export functions. The first one corresponds to a function that sets all the parameters required for a scan task. Although the logic is practically the same in the two functions, there are some variations in the utilized internal data structures, which are evident through the allocation size for the corresponding structs. This may suggest that some adaptations were made to the structs so as to suite the SiliVaccine engine implementation.

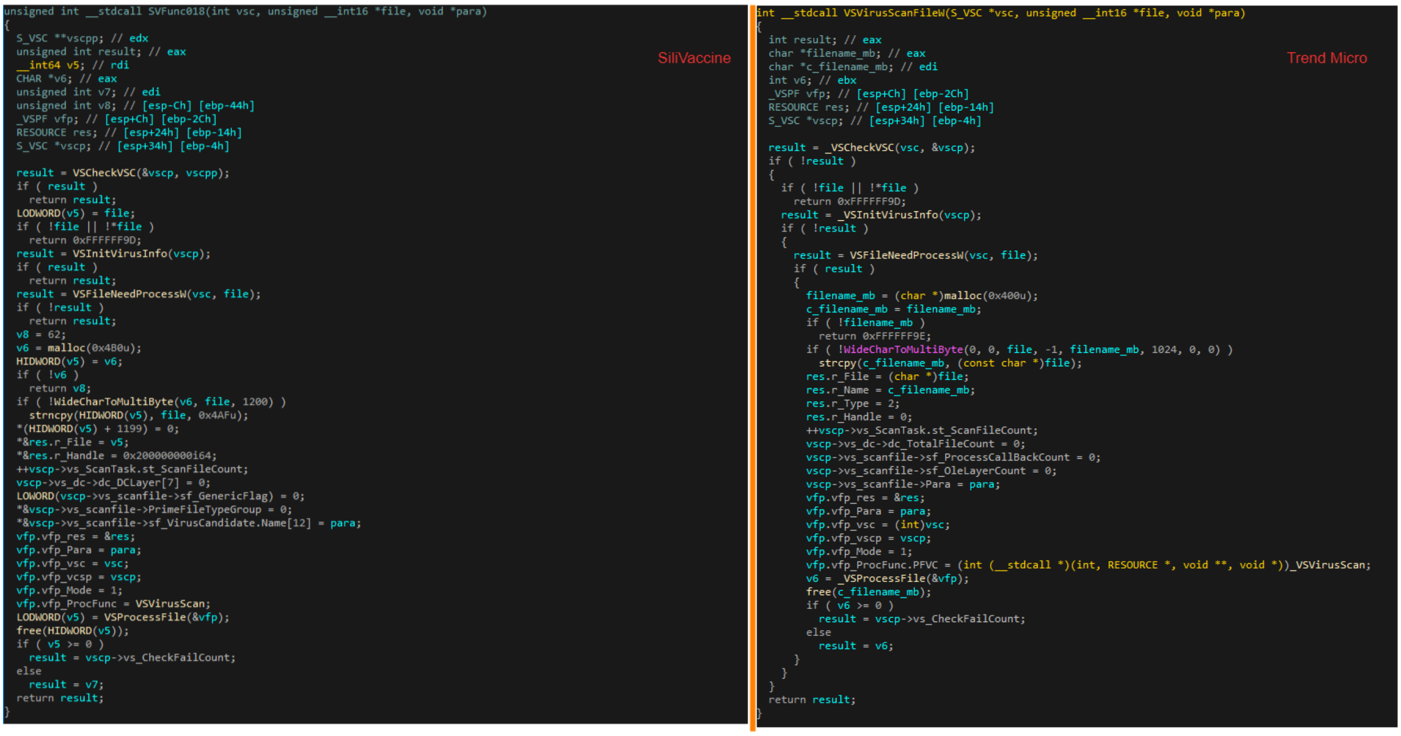


Figure 7: Comparison between the virus scan function in *SVKernel.dll* and *vsapi32.dll.*

The next comparison is of a function that initializes fields in an internal configuration struct used throughout a scan. The particular field updated here is one corresponding to an ID given to the scan's log file. An important difference can be witnessed between the functions – the SiliVaccine version uses inline versions of the *memset* and *memcpy* functions, while the Trend Micro engine actually calls the libc functions. This function inlining happens in a few similar places across the code, and suggests that possibly the Trend Micro source code was recompiled with an optimization that wasn't used in the original engine.

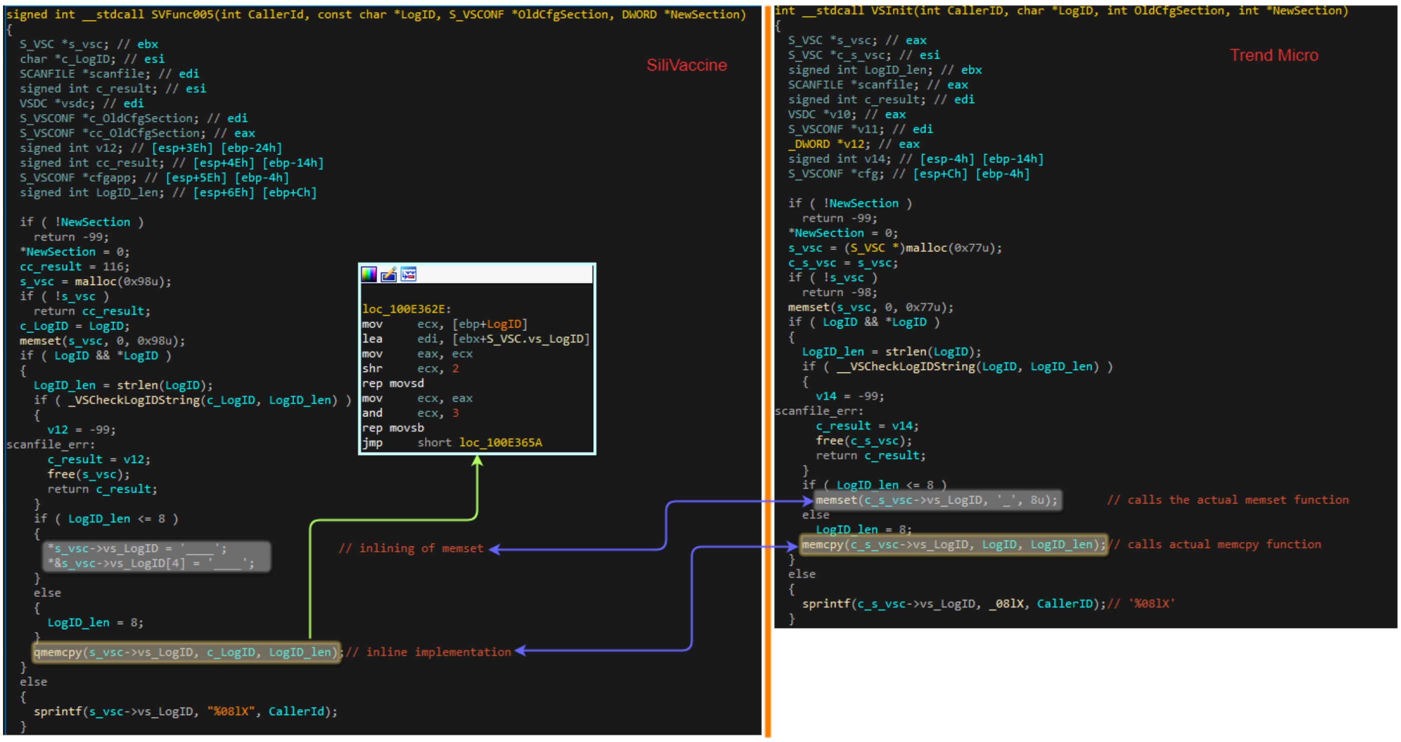


Figure 8: Comparison between proprietary data structure initialization functions in *SVKernel.dll* and *vsapi32.dll.*

Much similar to this, we can observe *SVFunc004* which initializes an internal scan information structure. In this case, we witness function inlining once again – this time to the *memcpy* and *strcpy* functions. More importantly, if we look at the data put into the struct, we can see that one of the fields contains the actual version of the engine, which is hardcoded in the binary. Thus, we can infer that the Trend Micro engine leveraged by SiliVaccine's authors is 8.910-1002. While information on this version is available online, it is pretty rare and harder to find than its successor (version 8.95) and predecessor (version 8.87). We can also tell that this engine version was released in August 2008, and was still deployed in the SiliVaccine version we were examining, which is known to be from 2013.

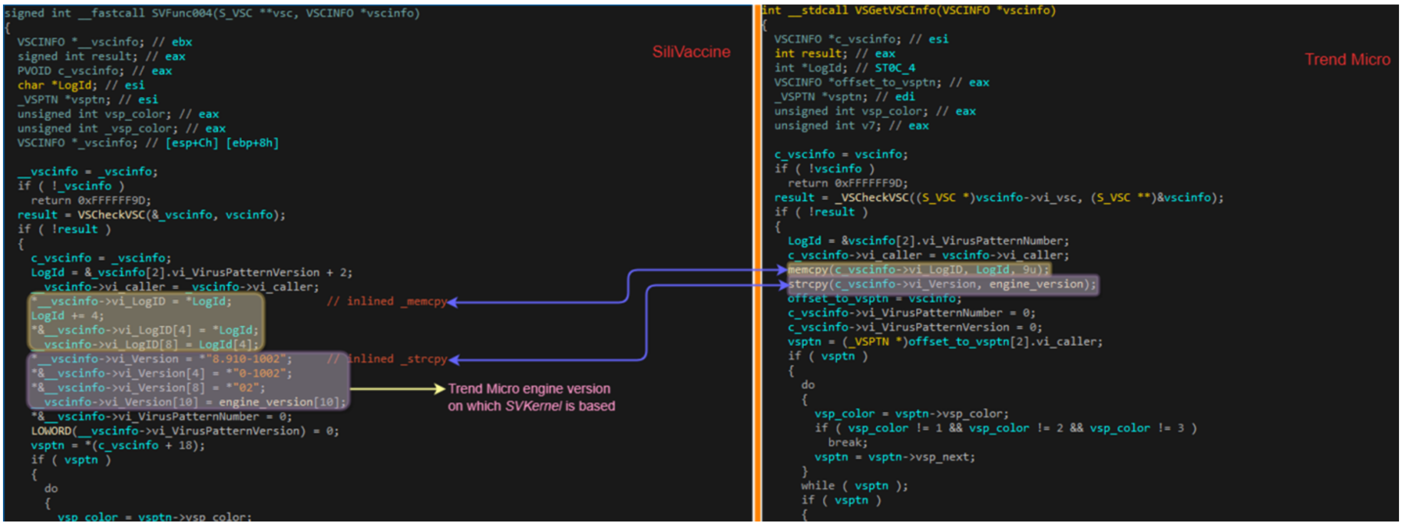


Figure 9: Comparison of scan info struct initialization functions in *SVKernel.dll* and *vsapi32.dll.*

Finally, if we look at scan cleanup function at *SVFunc006*, which removes various AV file & memory artifacts, we see a key difference between Trend Micro and SiliVaccine's engines. If we look at the *SVKernel* version of the function, we can spot a call to a native function used to unload and cleanup a legacy SiliVaccine driver named *SVIO.sys*. This driver doesn't exist in this version of the AV, and has nothing to do with *vsapi32* nor Trend Micro.

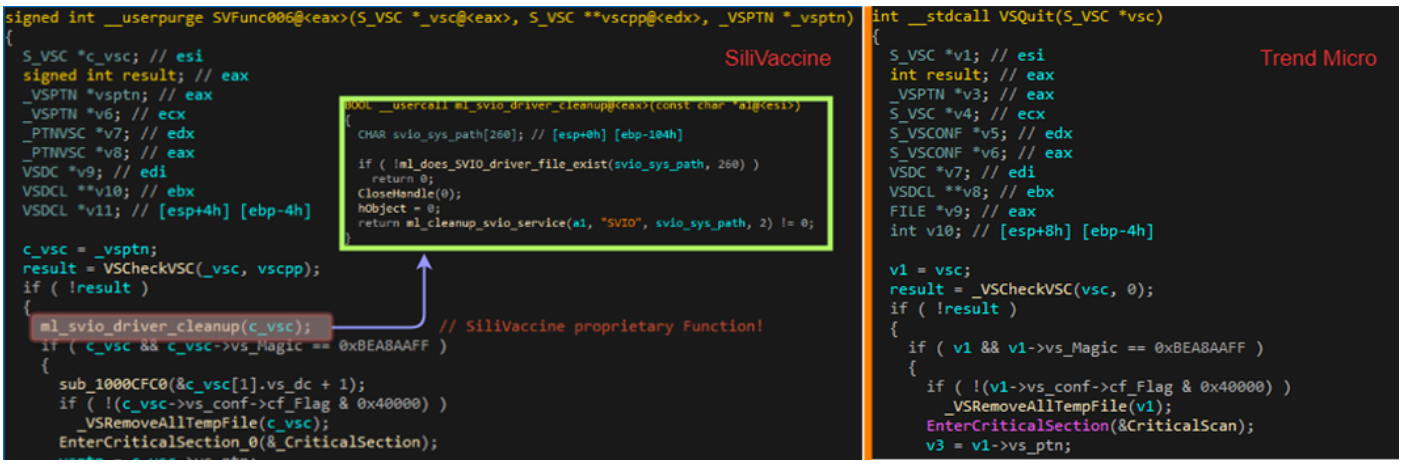


Figure 10: Comparison of scan info struct initialization functions in *SVKernel.dll* and *vsapi32.dll.*

Trend Micro's Response

At this point, it is pertinent to mention that we reached out to notify Trend Micro of their detection engine being used in SiliVaccine. They responded very promptly and were highly cooperative with our team. Their response was as follows:

"Trend Micro is aware of the research by Check Point on the 'SiliVaccine' North Korean anti-virus product, and Check Point has provided us with a copy of the software for verification. While we are unable to confirm the source or authenticity of that copy, it apparently incorporates a module based on a 10+ year-old version of the widely distributed Trend Micro scan engine used by a variety of our products. Trend Micro has never done business in or with North Korea. We are confident that any such usage of the module is entirely unlicensed and illegal, and we have seen no evidence that source code was involved. The scan engine version at issue is quite old and has been widely incorporated in commercial products from Trend Micro and third party security products through various OEM deals over the years, so the specific means by which it may have been obtained by the creators of SiliVaccine is unknown. Trend Micro takes a strong stance against software piracy, however legal recourse in this case would not be productive. We do not believe that the infringing use at issue poses any material risk to our customers."

Trend Micro's indication that a widely licensed library was misappropriated may be behind SiliVaccine's use of a 10+ year-old version of their scan engine is backed up by an additional analysis our team made of an older version of SiliVaccine, too. This suggests that this is not a one-time occurrence.

Pattern Files

SiliVaccine's malware signatures are stored in a series of files called Pattern files. A Deeper look into these files reveals they are in fact a Trend Micro's pattern file (*lpt$vpn* file), encrypted and divided into 2MB chunks. This is hardly surprising due to fact that SiliVaccine uses Trend Micro's scanning engine.

SVPatt00.spf	2/12/2018 12:36 PM	SPF File	2,049 KB
SVPatt01.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt02.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt03.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt04.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt05.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt06.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt07.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt08.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt09.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt10.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt11.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt12.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt13.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt14.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt15.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt16.spf	2/12/2018 12:36 PM	SPF File	2,048 KB
SVPatt17.spf	2/12/2018 12:36 PM	SPF File	2,048 KB

Figure 11: Listing of files that comprise the overall signature repository used by SiliVaccine, known as the pattern file.

The pattern files are encrypted with what seems like a custom encryption protocol that also utilizes a slightly modified SHA1 hashing algorithm. While understanding the encryption algorithm itself is quite a challenge, it's also not really necessary, as the decrypted pattern file can simply be dumped from memory after the decryption:

```
00 00 00 00 00 00 54 52 4F 4A 5F 33 31 33 36 00   ......TROJ_3136.
00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 53 49 42 59 4C 4C 45 2E 38 35   ......SIBYLLE.85
33 2D 31 00 00 00 00 01 00 00 00 00 00 00 00 00   3-1.............
00 00 00 00 00 00 53 45 43 54 30 00 00 00 00 00   ......SECT0.....
00 00 00 00 00 00 00 FA 19 00 00 00 00 00 00 00   .......ú........
00 00 00 00 00 00 50 53 5F 4D 50 43 2E 35 31 37   ......PS_MPC.517
00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 49 4E 43 48 5F 48 49 47 48 2E   ......INCH_HIGH.
35 33 34 2D 43 00 00 01 00 00 00 00 00 00 00 00   534-C...........
00 00 00 00 00 00 4D 45 47 41 44 45 56 49 4C 2E   ......MEGADEVIL.
36 36 35 2D 4F 00 00 01 00 00 00 00 00 00 00 00   665-O...........
00 00 00 00 00 00 4D 45 47 41 44 45 56 49 4C 2E   ......MEGADEVIL.
36 36 35 00 00 00 00 01 00 00 00 00 00 00 00 00   665.............
00 00 00 00 00 00 37 54 48 53 4F 4E 2E 32 38 34   ......7THSON.284
2E 44 00 00 00 00 00 01 00 00 00 00 00 00 00 00   .D..............
00 00 00 00 00 00 5A 45 52 4F 5F 48 55 4E 2E 34   ......ZERO_HUN.4
31 31 00 00 00 00 00 01 00 00 00 00 00 00 00 00   11..............
00 00 00 00 00 00 54 52 4F 4A 5F 42 56 5F 44 41   ......TROJ_BV_DA
4B 55 4D 41 00 00 00 01 00 00 00 00 00 00 00 00   KUMA............
```

Figure 12: Part of a memory dump containing the decrypted pattern file.

One entertaining fact regarding the decryption process is the decryption 'key'.

```
  strcpy(key, "voxjsdkaghghk");                // pattern decryption key
  id_of_pattern_file = 0;
  mk_init_pattern_decryption_globals();
  if ( !initalize_pattern_decryption_pads(key, 0x34124E5D, 0) )
  {
    mk_HeapFree_wrapper(pattern_file_raw);
LABEL_17:
    mk_HeapFree_wrapper(parsed_pattern_file);
    return 0;
  }
```

The key seems to be a combination of random English letters. However, if typed on a Korean-English keyboard when the language is set to 'Korean', you would get the Korean phrase that translates literally to 'Pattern encryption'.

The function *SVFunc019* exported by SVKernel is responsible for reading, decrypting and parsing the pattern files. It does so by reading each file, decrypting it, and appending it to a global buffer where the complete decrypted buffer is stored:

```
sprintf((int)&current_pattern_chunk, "%sSVPatt%.2d.spf", prefix, 0);
for ( hFile = CreateFileA(&current_pattern_chunk, 0x80000000, 1u, 0, 3u, 0x80u, 0);
      hFile != (HANDLE)-1;
      hFile = CreateFileA(&current_pattern_chunk, 0x80000000, 1u, 0, 3u, 0x80u, 0) )
{
  file_size = GetFileSize(hFile, 0);
  if ( !id_of_pattern_chunk )
  {
    file_size -= 0x40;
    SetFilePointer(hFile, 0x40, 0, 0);
  }
  ReadFile(hFile, pattern_file_raw, file_size, &NumberOfBytesRead, 0);// read the pattern file
  CloseHandle(hFile);
  memset(decrypted_pattern_chunk, 0, 0x500000u);
  mk_decrypt_pattern_file((int *)decrypted_pattern_chunk, (int *)pattern_file_raw, file_size);
  qmemcpy(&mk_g_decrypted_pattern_file[bytes_mapped], decrypted_pattern_chunk, 4 * (file_size >> 2));
  decrypted_pattern_file_end_offset = &mk_g_decrypted_pattern_file[4 * (file_size >> 2) + bytes_mapped];
  bytes_mapped += file_size;
  ++id_of_pattern_chunk;
  qmemcpy(decrypted_pattern_file_end_offset, &decrypted_pattern_chunk[4 * (file_size >> 2)], file_size & 3);
  sprintf((int)&current_pattern_chunk, "%sSVPatt%.2d.spf", prefix, id_of_pattern_chunk);
}
```

Figure 13: Pattern file loading and decryption code.

SVMain and SVDealer call the proprietary *SVFunc019* function in order to load and decrypt the pattern file, and from there they use Trend Micro's code (*SVFunc007 – VSReadPatternInFile*) in order to parse and load the signatures themselves.

Detection Names Renaming Scheme

As previously described, SiliVaccine uses Trend-Micro's pattern files, which contain Trend Micro's signature names. However, these names are never revealed to the end user, as it performs internal renaming of detected malware names, essentially converting them from Trend Micro's format into its own "proprietary" format. This functionality is handled by dedicated functions in the *SVMain & SVDealer* modules, set as a callback by calling the *SVFunc016* (*VSSetProcessFileCallBackFunc*) export in *SVKernel*, and triggered right after each file scan. If the scan detected a malware, the matching detection name is taken as it appears in the pattern file and converted into a custom format as described below. Throughout the program only the modified name format is referenced, as described further in the whitelisting section.

The renaming method works as follows:

1. The detection name from the pattern file (as reported by the *SVKernel* scan) is split into parts by searching for the following delimiters: "-", "_", "."
2. The first part (the prefix), which usually specifies the malware type/category, is replaced according to the following table.

Trend Micro Prefix	SiliVaccine Prefix
PE	W32
WORM	Wrm
BKDR	Bkd
Cryp	Crp
TROJ	Trj
TSPY	Spy
Possible	Poss
Html	Htm

Other prefixes are used as-is. The prefix replacement is shown in the following screenshot:

```
    if ( ml_stricmp_wrapper_(v5, "TSPY") )
    {
      if ( ml_stricmp_wrapper_(*(unsigned __int8 **)detection_info_struct->detection_names, "Possible") )
      {
        result = (unsigned __int8 **)ml_stricmp_wrapper_(
                                       *(unsigned __int8 **)detection_info_struct->detection_names,
                                       "Html");
        if ( !result )
        {
          *detection_info_struct->detection_names = (char **)heap_alloc_or_realloc(
                                                             *(LPVOID *)detection_info_struct->detection_names,
                                                             4u);
          result = (unsigned __int8 **)*detection_info_struct->detection_names;
          *result = (unsigned __int8 *)'mtH';
        }
      }
      else
      {
        *detection_info_struct->detection_names = (char **)heap_alloc_or_realloc(
                                                           *(LPVOID *)detection_info_struct->detection_names,
                                                           5u);
        result = (unsigned __int8 **)*detection_info_struct->detection_names;
        *result = (unsigned __int8 *)'ssoP';
        *((_BYTE *)result + 4) = 0;
      }
    }
    else
    {
      *detection_info_struct->detection_names = (char **)heap_alloc_or_realloc(
                                                         *(LPVOID *)detection_info_struct->detection_names,
                                                         4u);
      result = (unsigned __int8 **)detection_info_struct->detection_names;
      **detection_info_struct->detection_names = (char *)'ypS';
    }
```

Figure 14: Detection name prefix replacement code.

3. If there are more than 2 parts, the last part (the suffix), is replaced in the following way:

Trend Micro Suffix	SiliVaccine Suffix
'0' – '9'	'A' – 'J'
'O'	'Org'
Everything else	calculated hex string

4. If there are more than 3 parts, the part before the suffix is replaced with a calculated hex string.

5. For each part, the first letter is converted to uppercase and everything else to lowercase.

6. A new name is constructed by joining all the parts with a dot (.) separator:

```
          chr = detection_name_[i];
          if ( first_letter == 1 )
          {
            if ( chr < 'a' || chr > 'z' )
            {
              if ( chr >= 'A' && chr <= 'Z' )
                first_letter = 0;
              goto continue;
            }
            modified_chr = chr - 0x20;          // change to upper case
            first_letter = 0;
          }
          else
          {
            if ( chr < 'A' || chr > 'Z' )
              goto continue;
            modified_chr = chr + 0x20;          // change to lower case
          }
          detection_name_[i] = modified_chr;
continue:
          if ( ++i >= len )
            goto break;
        }
      }
    }
    goto break;
  }
construct_detection_names:                      // construct Sili's detection name
  SV_renamed_length = malware_info_struct->detection_segments_count + total_segments_len;
  SV_renamed_detection = (char *)ml_heap_alloc_not_persistent(SV_renamed_length);
  memset(SV_renamed_detection, 0, SV_renamed_length);
  for ( index = 0; index < malware_info_struct->detection_segments_count; ++index )
  {
    qmemcpy(
      &SV_renamed_detection[strlen(SV_renamed_detection)],
      malware_info_struct->detection_names[index],
      (char *)malware_info_struct->detection_names[index]
    + strlen((const char *)malware_info_struct->detection_names[index])
    + 1
    - (char *)malware_info_struct->detection_names[index]);
    if ( index < malware_info_struct->detection_segments_count - 1 )
      *(_WORD *)&SV_renamed_detection[strlen(SV_renamed_detection)] = '.';
  }
  return SV_renamed_detection;
```

Figure 15: Constructing the final renamed detection name.

Here are a few examples of Trend Micro names and their matching SiliVaccine names.

Trend Micro Signature	SiliVaccine Signature
PRIMUS.512-O	Primus.512.Org
TROJ_STEAL1	Trj.Steal1
MAL_NUCRP-5	Mal.Nucrp.F

Malware White-Listing

During our research we discovered that the authors of SiliVaccine have chosen to white-list a single very specific malware signature, and effectively ignore any detection of files matching that specific signature.

The white-listed signature is Trend Micro's 'MAL_NUCRP-5', described by Trend Micro as:

"...the Trend Micro detection for suspicious files that manifest behavior and characteristics similar to known NUWAR, TIBS, and ZHELAT variants."

This signature doesn't seem to be related to any one specific malware, but rather seems to detect specific packing related characteristics common in some malware. The 'MAL' prefix used in this signature indicates this is a generic signature, described by Trend Micro as a 'Second Level Generic Detection Name', essentially a heuristic signature created by observing a common pattern in existing malware. Unsurprisingly, Looking at a group of around 20 different files detected as MAL_NUCRP-5 reveals vastly different and un-related malware samples, ranging from fake AV installers to a dropper component seemingly related to a Chinese APT attack.

The white-listing functionality is hard-coded in the binary components of the program itself and is handled by implicit comparisons in the code that were added solely for that purpose. This effectively equals to removing the signature completely, which would have made more sense. However, the signature itself is still present in the pattern files that came with SiliVaccine. This indicates that the authors either have no direct way of modifying Trend Micro's pattern files, or no interest of doing so every time they update them.

Getting into the details, the white-listing is implemented by SiliVaccine's two major components; *SVMain* & *SVDealer* – namely, the components that invoke file scans using *SVKernel*. In multiple places throughout the code, immediately after the export function *SVFunc018* (VSVirusScanFileW) is called, the global string that stores the last scan's detection name is compared to '*Mal.Nucrp.F*'. That string is actually Trend Micro's detection name '*MAL_NUCRP-5*' after SiliVaccine's renaming. In case the detection name does equal '*Mal.Nucrp.F*', the code branches out to continue as if the scan didn't find anything (ignoring the detection completely).

```
MultiByteToWideChar(0, 0, &file_to_scan_mb, strlen(&file_to_scan_mb) + 1, &file_to_scan_w, 256);
if ( SVFunc018(0, &file_to_scan_w, 0) > 0 && ml_strcmp(&mk_g_detection_name, L"Mal.Nucrp.F") )
{
  mk_handle_malicious_file(&file_to_scan_w);// scan found file to be malicious and it's NOT Mal.Nucrp.F
  malicious_file_found = 1;
}
else
{
  malicious_file_found = 0;
}
```

```
ResetEvent(0);
is_malicious? = SVFunc018(SV_Struct, file_path, 0);// scan file
SetEvent(0);
if ( is_malicious? > 0 && !strwcmp(&SV_malware_name_wide, L"Mal.Nucrp.F") )// check if NOT Mal.Nucrp.F
  return -1;
mk_g_last_scan_result = is_malicious?;
if ( is_malicious? > 0 )
  ++_this->detection_counter;
return is_malicious?;
```

Figures 16 & 17: White-listing the *Mal.Nucrp.F* detection, following a file scan.

Another instance of this type of comparison can be found in the *SVMain* module, specifically in the renaming callback function that that was previously described. However, it's amusing to note that the authors seem to have made a typo this time and wrote '*Mal.Nurcrp.F*' (with an extra 'r' in the middle) by mistake.

```
if ( detection_name )
{
  mk_copy_TM_detection_to_SV_struct(detection_name + 8, &sili_detection_info);// perform renaming
  mk_modify_TM_Detection_format(&sili_detection_info);
  malware_name_string? = mk_construct_full_detection_name_(&sili_detection_info);
  MultiByteToWideChar(0, 0, malware_name_string?, strlen(malware_name_string?) + 1, &SV_malware_name_wide, 256);
  ml_heap_free_wrapper(malware_name_string?);
  mk_heap_free_array((LPVOID *)&sili_detection_info.detection_names);
  if ( !strwcmp(&SV_malware_name_wide, L"Mal.Nurcrp.F") )// A typo for Mal.Nucrp.F
  {
    CStringData_vftable_ptr = CStringData - 16;
    v17 = -1;
    v10 = _InterlockedDecrement((volatile signed __int32 *)(CStringData - 16 + 12));
    v11 = v10 == 0;
    v12 = v10 < 0;
    goto finish;                          // exit
  }
  mk_init_C_String(&Reused_Cstring, &mk_g_window_text_or_default_extension);
  LOBYTE(v17) = 1;
  mk_set_CString(&Reused_Cstring, &SV_malware_name_wide);
  mk_store_detection__(mk_g_sili_detection_, (int)&Reused_Cstring);
```

Figure 18: Typo in the white-listing comparison to *Mal.Nucrp.F* detection.

This does not affect the white-listing functionality however, since an extra implicit comparison is made after each scan instance anyway.

It's interesting to note that all the other similarly named signatures (*MAL_NUCRP-**) are not whitelisted. A scan of a group of files detected by Trend Micro as '*MAL_NUCRP-2*' (1 unique file), '*MAL_NUCRP-5*' (8 unique files) and '*MAL_NUCRP-6*' (3 unique files) is shown below:

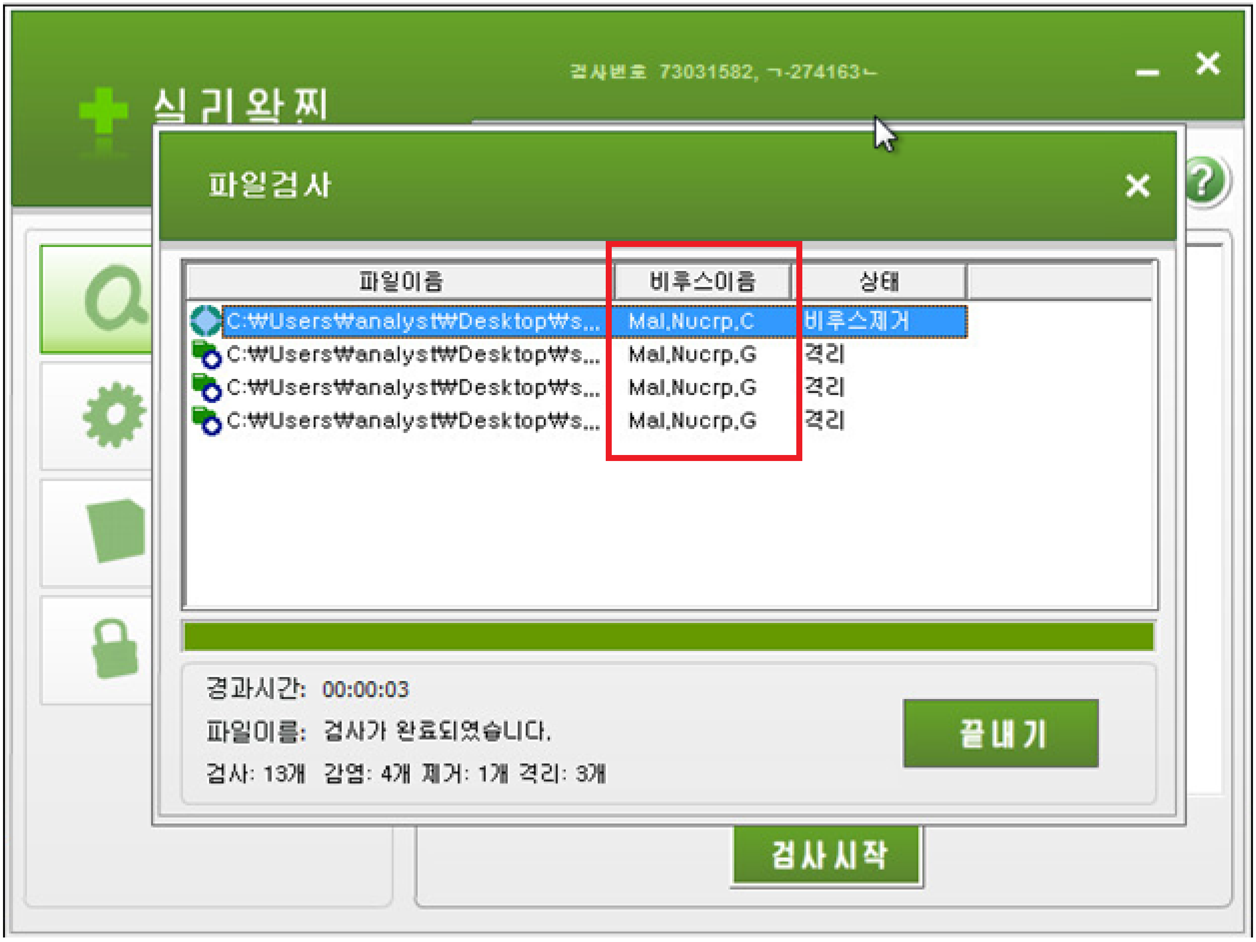


Figure 19: No detection *MAL_NUCRP-5*, as opposed to *MAL_NUCRP-2* and *MAL_NUCRP-6.*

Note how files that should have been detected as '*Mal.Nucrp.F*' are all notably missing, while the other files (*Mal.Nucrp.C* & *Mal.Nucrp.G*) are all correctly detected.

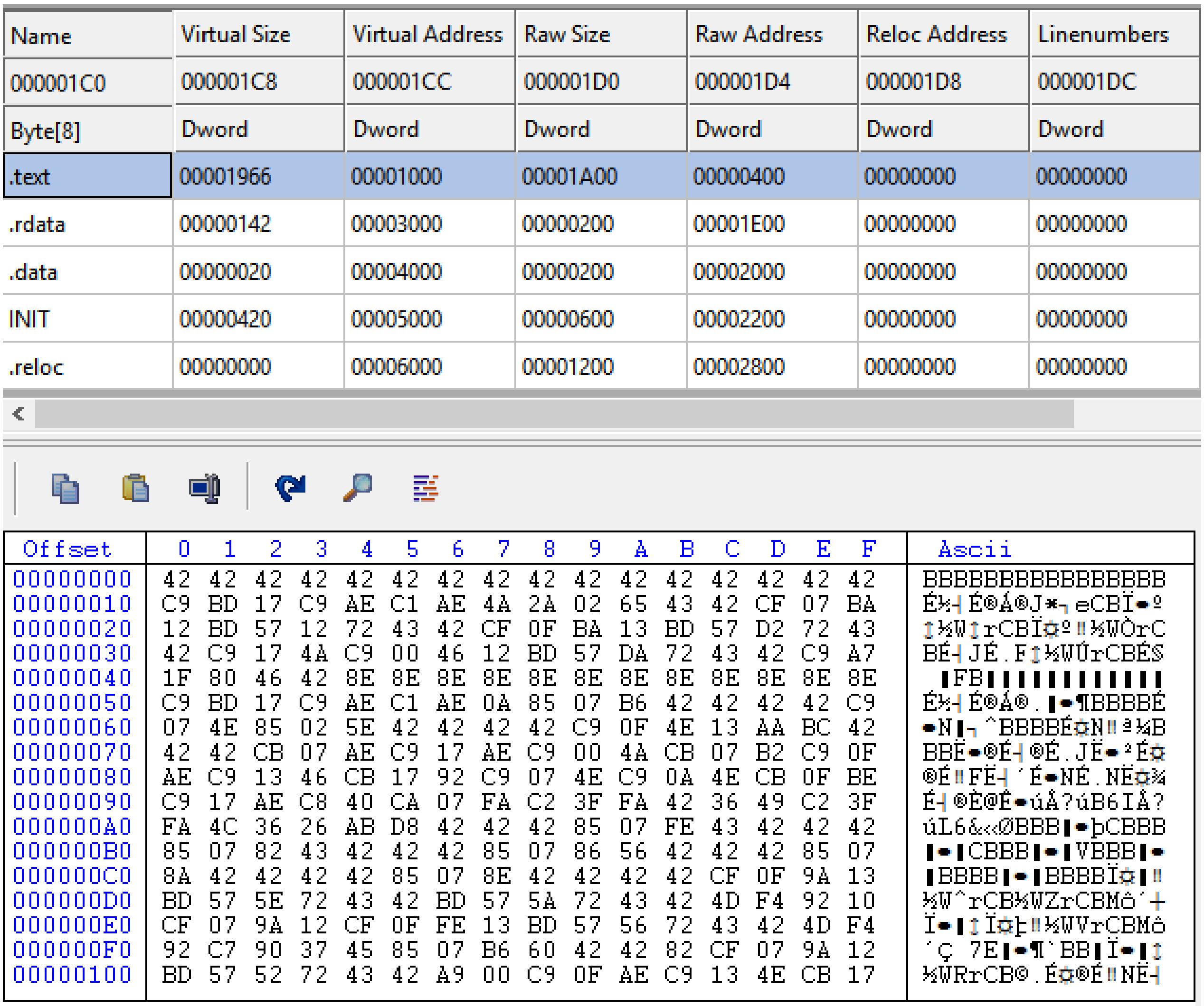
Kernel Drivers

SiliVaccine uses 3 driver components:

- *SVHook.sys* – Kernel-mode process information collection module.
- *SVFilter.sys* – File system filter driver used for real-time and AV files protection.
- *ststdi2.sys* – Network Transport Driver Interface (TDI) Driver.

Protection Mechanisms

Both *SVHook* and *SVFilter* drivers are 'packed' with a strange packer detected as 'BobCrypt2 protector' by Detect-It-Easy. This is an obscure detection which seems like a false positive, as the packer employed here would hardly qualify as a protector. Instead, the packing used is a simple XOR of the *.text* section, with the byte 0x42, something that easy to spot when looking at it:



Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers
000001C0	000001C8	000001CC	000001D0	000001D4	000001D8	000001DC
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword
.text	00001966	00001000	00001A00	00000400	00000000	00000000
.rdata	00000142	00003000	00000200	00001E00	00000000	00000000
.data	00000020	00004000	00000200	00002000	00000000	00000000
INIT	00000420	00005000	00000600	00002200	00000000	00000000
.reloc	00000000	00006000	00001200	00002800	00000000	00000000

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42
00000010  C9 BD 17 C9 AE C1 AE 4A 2A 02 65 43 42 CF 07 BA
00000020  12 BD 57 12 72 43 42 CF 0F BA 13 BD 57 D2 72 43
00000030  42 C9 17 4A C9 00 46 12 BD 57 DA 72 43 42 C9 A7
00000040  1F 80 46 42 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E 8E
00000050  C9 BD 17 C9 AE C1 AE 0A 85 07 B6 42 42 42 42 C9
00000060  07 4E 85 02 5E 42 42 42 42 C9 0F 4E 13 AA BC 42
00000070  42 42 CB 07 AE C9 17 AE C9 00 4A CB 07 B2 C9 0F
00000080  AE C9 13 46 CB 17 92 C9 07 4E C9 0A 4E CB 0F BE
00000090  C9 17 AE C8 40 CA 07 FA C2 3F FA 42 36 49 C2 3F
000000A0  FA 4C 36 26 AB D8 42 42 42 85 07 FE 43 42 42 42
000000B0  85 07 82 43 42 42 42 85 07 86 56 42 42 42 85 07
000000C0  8A 42 42 42 42 85 07 8E 42 42 42 42 CF 0F 9A 13
000000D0  BD 57 5E 72 43 42 BD 57 5A 72 43 42 4D F4 92 10
000000E0  CF 07 9A 12 CF 0F FE 13 BD 57 56 72 43 42 4D F4
000000F0  92 C7 90 37 45 85 07 B6 60 42 42 82 CF 07 9A 12
00000100  BD 57 52 72 43 42 A9 00 C9 0F AE C9 13 4E CB 17
```

Figure 20: *.text* section encrypted using a simple XOR with 0x42.

The EntryPoint of the binary is in the “*.reloc*” section, which contains the packer’s code.

SVFilter

The *SVFilter.sys* component is a file system filter driver utilized by SiliVaccine for 2 main purposes:

1. Real-time protection functionality – together with SVDealer, the user-mode component.
2. Protecting SiliVaccine’s binary components from deletion and modification.

Real Time Protection

The Real-time protection functionality is implemented by *SVDealer*, which uses the *SVFilter* driver to hook file system activity and scan files that are being accessed in real-time. *SVFilter* is loaded by *SVDealer* which communicates and controls it with it via IOCTLs. *SVDealer* instructs the driver to attach itself to all existing file system driver stacks, and then waits for a scan signal from the driver. Once the signal arrives, it reads the file path that needs to be scanned via a separate IOCTL, scans it, and reports back to *SVFilter* whether the file was found to be malicious (and not white-listed). The following screenshot shows this functionality from *SVDealer*’s side:

```
if ( DeviceIoControl(hSVFilter, 0xA6266207, &drives_list, 4u, &drives_list, 4u, &br, 0) )// attach fs drives
{
  g_logical_drives_str = drives_list;
  CreateThread(0, 0, mk_monitor_drives_related_thread_, 0, 0, &ThreadId);
  WaitForSingleObject(SVFilter_start_scan_event, 0xFFFFFFFF);// wait for scan signal from SVDealer
  ResetEvent(SVFilter_start_scan_event);
  memset(&file_to_scan_mb, 0, 256u);
  while ( DeviceIoControl(hSVFilter, 0xA626222C, 0, 0, &file_to_scan_mb, 256u, &br, 0) )// get file to scan from SVFilter
  {
    MultiByteToWideChar(0, 0, &file_to_scan_mb, strlen(&file_to_scan_mb) + 1, &file_to_scan_w, 256);
    if ( SVFunc018(0, &file_to_scan_w, 0) > 0 && ml_strcmp(&mk_g_detection_name, L"Mal.Nucrp.F") )
    {
      mk_handle_malicious_file(&file_to_scan_w);// scan found file to be malicious and it's NOT Mal.Nucrp.F
      malicious_file_found = 1;
    }
    else
    {
      malicious_file_found = 0;
    }
    if ( !DeviceIoControl(hSVFilter, 0xA6262230, &malicious_file_found, 1u, 0, 0, &br, 0) )// report scan result to SVFilter
      break;
    SetEvent(SVDealer_finished_scan_event);
    WaitForSingleObject(SVFilter_start_scan_event, 0xFFFFFFFF);
    ResetEvent(SVFilter_start_scan_event);
    memset(&file_to_scan_mb, 0, 256u);
  }
}
```

Figure 21: Conducting a scan by *SVDealer*, following a signal from *SVFilter* (real-time protection from *SVDealer*'s side)

The matching functionality on *SVFilter*'s side is buried deep inside a long and confusing function. After analyzing that function, it becomes evident that it is rather needlessly long and complex, and underneath it all, it simply performs the functionalities described above in a disorganized and confusing way. After multiple overlapping and weirdly specific checks, the function finally arrives to the real-time scanning functionality. The real-time file scan is only invoked upon execution of files. The code checks if the file is opened for execution, and if so stores the file path and signals *SVDealer* that a file needs to be scanned. Afterwards it checks the result reported by *SVDealer*. If the file is reported malicious, its name is stored in a list (whose purpose is not completely clear as it's not used in any sensible way), and access to file is blocked. The following snippet of code shows this functionality:

```
  if ( (options_ & 0xFF000000) == 0x1000000 && !file_attributes_ )// Opening an existing file
  {
    DesiredAccess_ = (unsigned __int16)DesiredAccess;
    if ( (unsigned __int16)DesiredAccess == 0xA1 )// FILE_READ_DATA | FILE_EXECUTE | FILE_READ_ATTRIBUTES, executing a file
      goto perform_scan;
    if ( (unsigned __int16)DesiredAccess != FILE_EXECUTE )// not executing, skip
      goto save_file_and_call_next_driver;
    if ( mk_check_remove_file_from_infected_list(file_name_) )// Is file in malicious list?
perform_scan:
      do_scan_file = 1;
    if ( DesiredAccess_ != 0x20 || do_scan_file )// if file is in malicious list or executing a file then scan with SVDealer
    {
      mk_scan_file_by_SVDealer(file_name_, do_scan_file);
      if ( do_scan_file )
      {
        if ( mk_g_malware_detected_by_SVDealer )
        {
          if ( DesiredAccess_ == 0xA1 )       // if this file was opened for execution, add to malicious list
          {
            mk_check_remove_file_from_infected_list(file_name_);
            mk_add_file_to_infected_list(file_name_);
          }
          goto ACCESS_DENIED;
        }
      }
    }
  }
```

Figure 22: Signaling *SVDealer* regarding file execution, and handling retrieved scan results (real-time protection from *SVFilter*'s side).

Protection of Internal Components

The same filter function in *SVFilter* also protects SiliVaccine’s binary files on disk by blocking any write access to them. The following screenshot displays the code segment that performs the relevant checks:

```
  if ( mk_strcmp_ascii(file_name_, SiliVaccine_install_dir, strlen(SiliVaccine_install_dir))// exe or dll in SiliVaccine's dir?
    || mk_strcmp_wrapper(&file_name_[strlen(file_name_) - 4], ".exe")
    && mk_strcmp_wrapper(&file_name_[strlen(file_name_) - 4], ".dll")
    || strchr(&file_name_[strlen(SiliVaccine_install_dir) + 1], '\\')// are we entering a directory?
    || (curr_pid = PsGetCurrentProcessId(), curr_pid == (HANDLE)IO_0xA626223C)// is this a permitted process?
    || curr_pid == (HANDLE)IO_0xA6262238
    || curr_pid == (HANDLE)IO_0xA6262240
    || !IO_0xA6262244 )
  {
keep_checking:
    if ( !mk_g_is_realtime_protection_on_ )     // skip checks if real time protection is OFF
    {
      if ( !file_name_ )
        goto Call_next_driver;
      goto add_file_and_call_next_driver;
    }
```

Figure 23: Protecting SiliVaccine files on disk by *SVFilter* from any write access.

SVHook

The *SVHook* driver is an odd and somewhat confusing component. The name implies it’s used for kernel-mode hooking, but it doesn’t contain any such functionality. Instead it is loaded and utilized by *SVMain* in order to query windows object metadata from the kernel when SiliVaccine performs a memory scan of the system. It essentially serves as a kernel mode agent to query information accessible only from the kernel.

The most interesting thing about this component is the debugging strings left in the code by the authors. One the functions in *SVHook* contains multiple *DbgPrint* calls, describing the state of the function, while referencing to the function itself as ‘*sub_800754*’:

```
PID = (PVOID)input->process_id;
DbgPrint("sub_8000754 Start\r\n");
if ( (unsigned int)PID < PID_LIMIT )
{
  DbgPrint("sub_8000754 1st Conditon FALSE\r\n");
  status = PsLookupProcessByProcessId(PID, &p_eprocess);
  if ( status < 0 )
    return status;
  if ( status >= 0 )
  {
    KeAttachProcess(p_eprocess);
    junk = 0;
    status = ObReferenceObjectByHandle(OUTPUT_dup->object_name.Name.Buffer, 0x80000000, 0, 0, &Object, 0);
    if ( status >= 0 )
    {
      PID = Object;
      if ( Object != *(PVOID *)&OUTPUT_dup->object_name.Name.Length )
      {
        ObfDereferenceObject(Object);
        status = STATUS_INVALID_PARAMETER;
      }
    }
    junk = -1;
    KeDetachProcess();
  }
  ObfDereferenceObject((PVOID)p_eprocess);
  if ( status < 0 )
    return status;
  PID = Object;
}
else
{
  DbgPrint("sub_8000754 1st Conditon True\r\n");
```

Figure 24: Peculiar debug strings appear in *SVHook*'s binary.

This is most probably an auto-generated name by the IDA reverse-engineering tool, indicating this function may have been copied from another driver the authors have reverse-engineered. The function itself seems to receive a handle from user-mode and returns the matching object name.

It's also interesting to note that the driver supports 13 IOCTL commands, while only 3 are ever called by the user-mode components (*SVMain*). Additionally, the driver seems to contain several bugs and mistakes that indicate this component was slapped together quickly and without fully understanding the purpose. For example, one of the used IOCTLs will always fail since the authors seem to have made a mistake in their condition statement: The input buffer size sent in the IOCTL by *SVMain* is 12 bytes (as it should probably be):

```
IOCTL_input.process_id = proc_info->ProcessID;
IOCTL_input.Object = (PVOID)proc_info->Object;
IOCTL_input.Handle = proc_info->process_handle;
handle_index = (int)&IOCTL_input;
ml_SVHook_device_ioctl(0x83350004, &IOCTL_input, 12u, 0, 0);// buffer size is 12
```

However, the authors seem to have mixed up the condition when verifying the size of the input in *SVHook*, and they return the status code STATUS_INVALID_PARAMETER when that's the case:

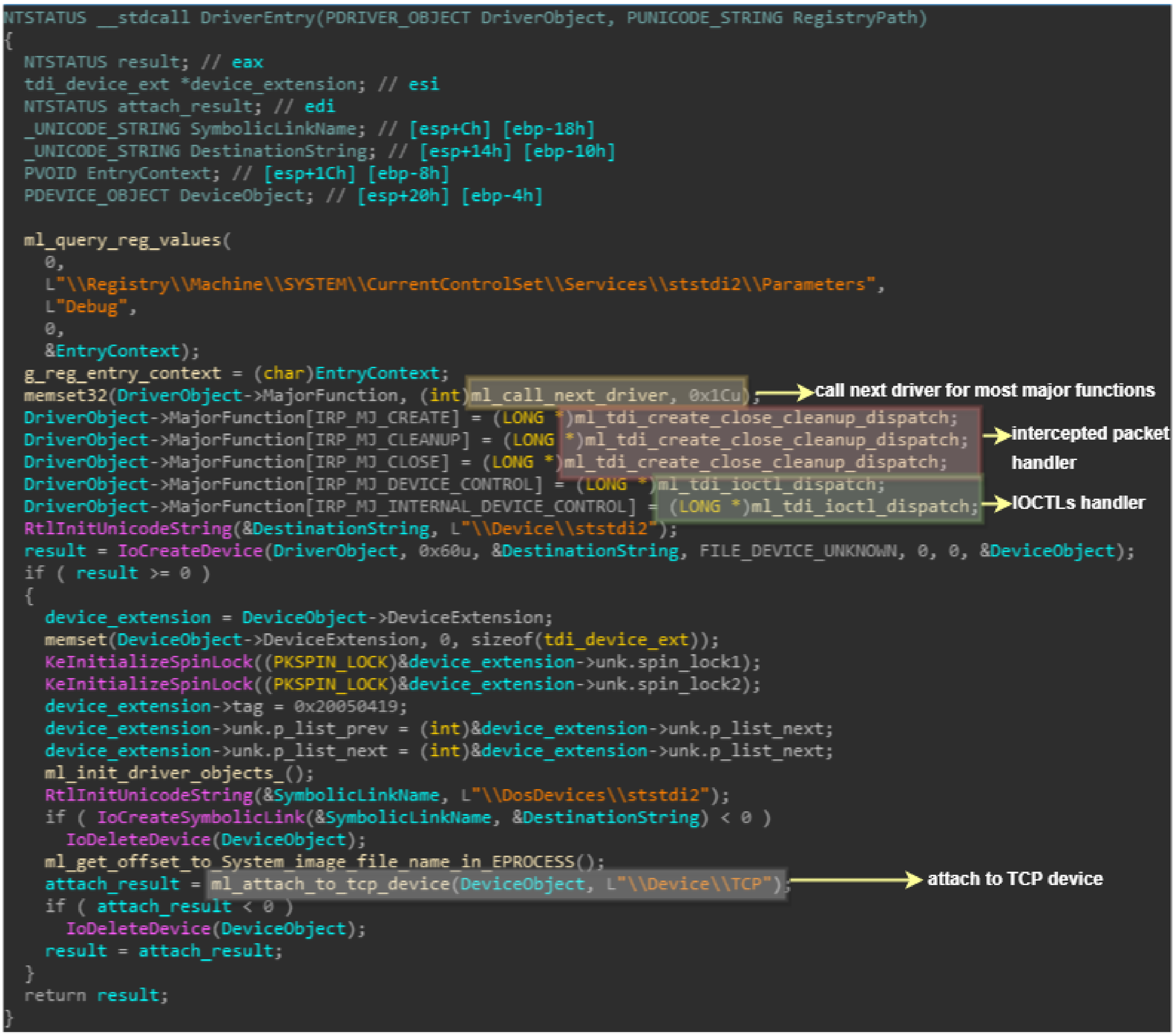
```
    break;
  case 4:                                    // 0x83350004 - called with input_buffer_length of 12
    if ( input_buffer_length == 12 )
      io_status->Status = STATUS_INVALID_PARAMETER;
    else
      io_status->Status = mk_close_handle_if_inheritable(input_buffer);
    break;
```

Figures 25 & 26: Programming mistake in buffer length check, when handling the 0x83350004 IOCTL by *SVHook.*

STSTDI

The final driver that can be found amongst SiliVaccine's components is *ststdi2.sys*. This is a TDI (Transport Driver Interface) filter driver, which is used to intercept TCP connections and log them in an internal data structure. The way this is done is by having the device corresponding to this driver attach on top of the system's TCP device, thus intercepting IRPs from higher level kernel TCP clients (for instance *HTTP.sys*, which uses the TDI API to communicate with TDI transport drivers).

If we observe the *DriverEntry*, we can see 3 dispatch functions. The first is a default function that just passes the IRP onwards to the TCP driver, making the TDI driver "transparent" for most cases. The second is the dispatch function that handles *IRP_MJ_CREATE*, *IRP_MJ_CLEANUP* and *IRP_MJ_CLOSE*, which will intercept any events of connection creation or termination. Finally, for *IRP_MJ_DEVICE_CONTROL* or *IRP_MJ_INTERNAL_DEVICE_CONTROL* the driver will deal with any particular passed IOCTLs.



```
NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax
  tdi_device_ext *device_extension; // esi
  NTSTATUS attach_result; // edi
  _UNICODE_STRING SymbolicLinkName; // [esp+Ch] [ebp-18h]
  _UNICODE_STRING DestinationString; // [esp+14h] [ebp-10h]
  PVOID EntryContext; // [esp+1Ch] [ebp-8h]
  PDEVICE_OBJECT DeviceObject; // [esp+20h] [ebp-4h]

  ml_query_reg_values(
    0,
    L"\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Services\\ststdi2\\Parameters",
    L"Debug",
    0,
    &EntryContext);
  g_reg_entry_context = (char)EntryContext;
  memset32(DriverObject->MajorFunction, (int)ml_call_next_driver, 0x1Cu);
  DriverObject->MajorFunction[IRP_MJ_CREATE] = (LONG *)ml_tdi_create_close_cleanup_dispatch;
  DriverObject->MajorFunction[IRP_MJ_CLEANUP] = (LONG *)ml_tdi_create_close_cleanup_dispatch;
  DriverObject->MajorFunction[IRP_MJ_CLOSE] = (LONG *)ml_tdi_create_close_cleanup_dispatch;
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = (LONG *)ml_tdi_ioctl_dispatch;
  DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = (LONG *)ml_tdi_ioctl_dispatch;
  RtlInitUnicodeString(&DestinationString, L"\\Device\\ststdi2");
  result = IoCreateDevice(DriverObject, 0x60u, &DestinationString, FILE_DEVICE_UNKNOWN, 0, 0, &DeviceObject);
  if ( result >= 0 )
  {
    device_extension = DeviceObject->DeviceExtension;
    memset(DeviceObject->DeviceExtension, 0, sizeof(tdi_device_ext));
    KeInitializeSpinLock((PKSPIN_LOCK)&device_extension->unk.spin_lock1);
    KeInitializeSpinLock((PKSPIN_LOCK)&device_extension->unk.spin_lock2);
    device_extension->tag = 0x20050419;
    device_extension->unk.p_list_prev = (int)&device_extension->unk.p_list_next;
    device_extension->unk.p_list_next = (int)&device_extension->unk.p_list_next;
    ml_init_driver_objects_();
    RtlInitUnicodeString(&SymbolicLinkName, L"\\DosDevices\\ststdi2");
    if ( IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString) < 0 )
      IoDeleteDevice(DeviceObject);
    ml_get_offset_to_System_image_file_name_in_EPROCESS();
    attach_result = ml_attach_to_tcp_device(DeviceObject, L"\\Device\\TCP");
    if ( attach_result < 0 )
      IoDeleteDevice(DeviceObject);
    result = attach_result;
  }
  return result;
}
```


Figure 27: *STSTDI*'s *DriverEntry* function,
showing the main dispatch routines and attachment to the TCP device.

The dispatch function that handles connections attempts to log them in a hash table, such that it will contain all the active connections at every point in time. In the figure below it can be seen that in the case of *IRP_MJ_CREATE*, i.e. connection creation, a system buffer associated to the IRP will be inspected and will determine if the connection address or context get appended to the hash table. In the same manner, upon interception of *IRP_MJ_CLOSE* (connection termination), the corresponding connection data will be located in the hash table and removed.

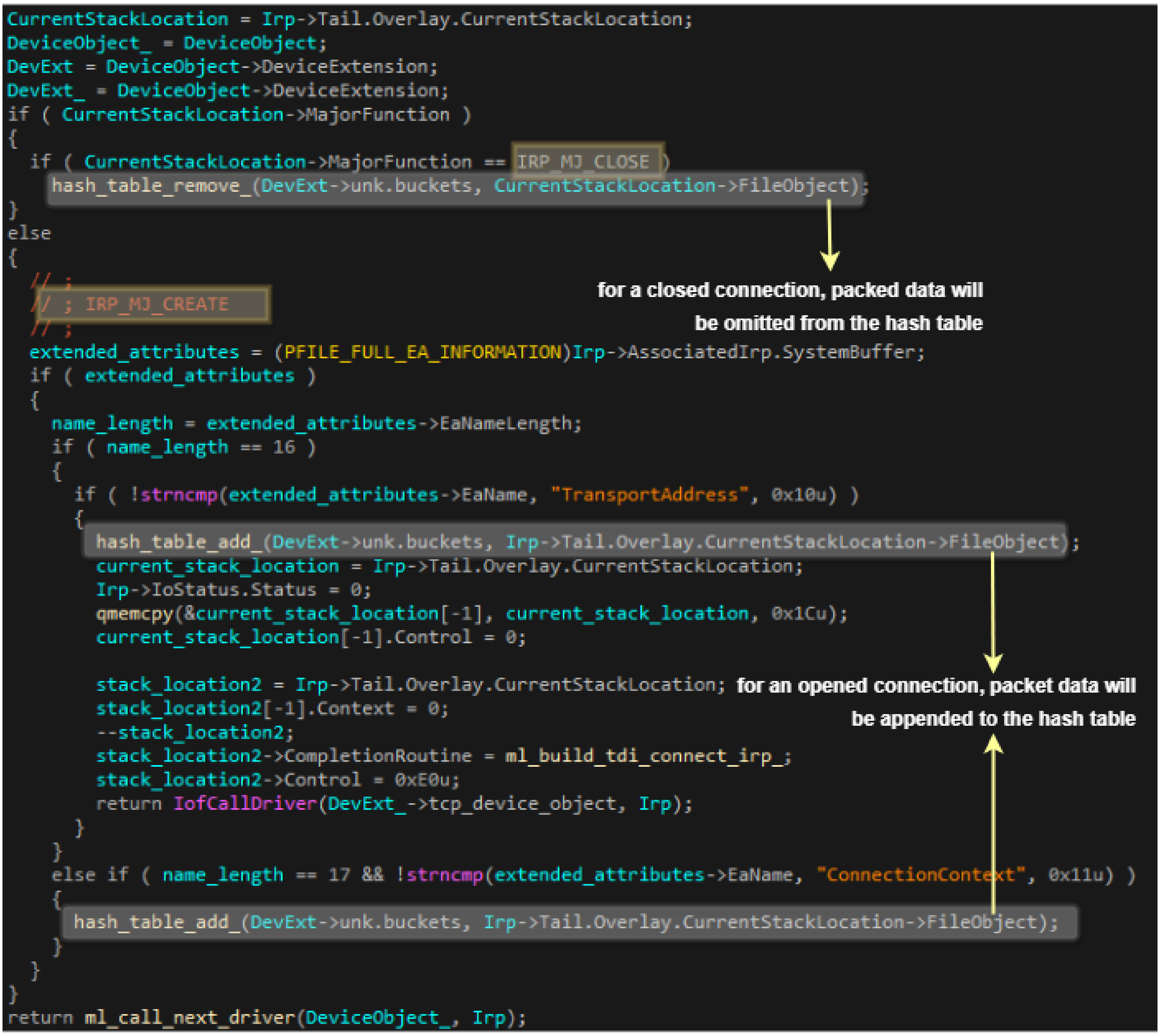


Figure 28: Handling of intercepted packets by *STSTDI.*

To use this logged data, the driver contains a handler for a set of IOCTLs, which allow an external entity (another driver or a user space component) to query and modify the underlying hash table. Strangely enough, there is no other component in the AV that actually issues these IOCTLs, making this driver seem redundant. It could be so that this is a legacy component, i.e. one that was previously incorporated but remained in the software even though it wasn't used.

Authorship Background

So far, we have seen the technical implementation specifics of various SiliVaccine components. There are however some intriguing details that can be found by looking at file meta-data. Namely, if we observe the version info resident within the resource section of the AV's PE components (which is only accessible in the unpacked versions of the files), we can spot two company names – PGI (or Pyonyang Gwangmyong Information Technology) and STS Tech-Service.

```
1 VERSIONINFO
FILEVERSION 4,0,0,5
PRODUCTVERSION 4,0,0,5
FILEOS 0x40004
FILETYPE 0x2
{
BLOCK "StringFileInfo"
{
        BLOCK "041204b0"
        {
                VALUE "CompanyName", "Pyongyang Gwangmyong Information Technology"
                VALUE "FileDescription", "SVKernel"
                VALUE "FileVersion", "4, 0, 0, 5"
                VALUE "InternalName", "SVKernel"
                VALUE "LegalCopyright", "Copyright © 1999 - 2012 Pyongyang Gwangmyong Information Technology. All rights reserved."
                VALUE "LegalTrademarks", "Copyright © PGI Tech."
                VALUE "OriginalFilename", "SVKernel.dll"
                VALUE "ProductName", "SVKernel"
                VALUE "ProductVersion", "4, 0, 0, 5"
        }
```

```
1 VERSIONINFO
FILEVERSION 4,0,2,2
PRODUCTVERSION 4,0,2,2
FILEOS 0x4
FILETYPE 0x1
{
BLOCK "StringFileInfo"
{
        BLOCK "000004b0"
        {
                VALUE "CompanyName", "STS Tech-Service"
                VALUE "FileDescription", "SiliVaccine Tray Manager"
                VALUE "FileVersion", "4, 0, 2, 2"
                VALUE "InternalName", "SVTray.exe"
                VALUE "LegalCopyright", "Copyright (C) 2013 STS Tech-Service"
                VALUE "OriginalFilename", "SVTray.exe"
                VALUE "ProductName", "SVTray"
                VALUE "ProductVersion", "4, 0, 2, 2"
        }
}
```

Figure 29: Company names hiding in the version info of SiliVaccine's binaries.

While we can guess from the name Gwangmyong, which is most likely a misspelling of Kwangmyong (DPRK's national intranet) that PGI is likely a North Korean establishment, STS Tech-Service remains somewhat of a mystery. This supposed company has no internet website or clear record of activity on the internet, leaving some question marks regarding its origins and line of business.

Nonetheless, some clues are publically available in the web, but rather than giving clear answers to the aforementioned questions, they leave room for more pondering. One example for this is the fact that STS Tech-Service is listed as a co-author of multiple Japanese applications, written alongside two companies – Silver Star Japan and Magnolia. Programs like Mahjong (popular Asian tile-based game) or Iron Security (file encryption utility), seem like legitimate apps developed by these companies for the Japanese market. Does this mean that STS Tech-Service is in fact based in Japan?

Figure 30: Examples of applications developed by STS Tech-Service in collaboration with Japanese companies.

製品名	「癜堂(R)シリーズ」
ジャンル	ゲームソフト
製品内容	「癜堂(R)シリーズ」は、新思考エンジンを搭載し人間の感性を再現。 初心者から上級者まで、人と対戦するような臨場感を味わうことができます。
発売日	2004年1月1日(木)
価格	1,980円（税別）
動作環境	Windows XP/Me/98/98SE/2000
販売元	ソースネクスト株式会社
コピーライト	（C）2004 MAGNOLIA C 2004 STS Tech-Service （C）2004 Korea Computer Center （C）2004 SilverStar Japan
製品詳細	http://www.sourcenext.com/products/dendo/go.html http://www.sourcenext.com/products/dendo/mahjong.html http://www.sourcenext.com/products/dendo/shogi.html

A more in-depth inquiry of this company gives evidence that it's not actually the case. Looking at an invitation brochure to an event titled "Pyongyang International Technology and Infrastructure Exhibition" that took place in North Korea back in 2006 we can see STS Tech-Service listed as one of the participating companies. The event's original purpose was to encourage cooperation between DPRK based businesses with other worldwide foreign companies. While a lot of the companies that can be seen in this brochure are based in various parts of the world, STS Tech-Service seems to be actually situated in the DPRK, according to information from the exhibition organizers.

Figure 31: Invitation to ITIE, an exhibition that took place in Pyonyang in 2006. STS Tech-Service was one of the participating companies.

The Mysterious Patch File

As previously mentioned, the copy of the installation file of SiliVaccine was sent to us by Martyn Williams, a freelance journalist, which in turn received it from a mailbox of a mysterious sender from a Japanese origin.

The Installer file is named 'SiliVaccine4.0_2014_07_08.exe', implying it is SiliVaccine version 4.0 dated 08.07.2014. Examining the file reveals it is actually a WinRAR SFX archive, containing an older dated installer and a supposed 'patch' file.

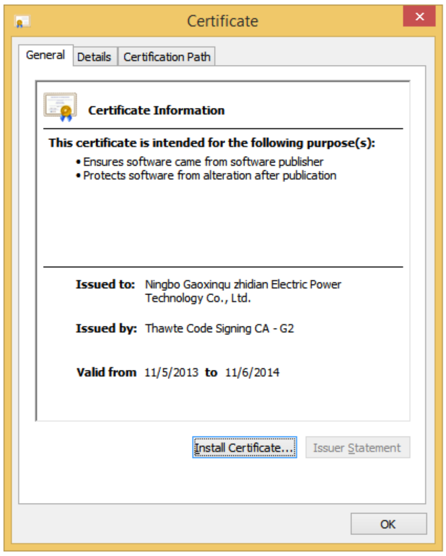
Name	Date modified	Type	Size
SiliVaccine4.0_2014_06_25.exe	11/22/2013 3:37 PM	Application	88,629 KB
SVPatch4.0_2014_07_08.exe	5/15/2014 3:42 PM	Application	117 KB

Figure 32: The contents the received installer file.

Upon execution of the SFX archive, both of the files inside are executed in turn, automatically. The older dated installer file inside is indeed the legitimate installer of SiliVaccine. A closer look at the supposed patch file however, reveals it to be a cleverly disguised JAKU malware instead, specifically a Stage 1 dropper.

It is interesting to note that the file is signed by a certificate that is extremely similar to one of the certificates reportedly used in the DarkHotel APT campaign. This makes sense – the JAKU campaign report states there are "clear connections" between JAKU and DarkHotel, implying the actor behind may be the same, while the DarkHotel Report by Kaspersky suggests that the actor behind the attacks has performed certificate theft. It's interesting to note that both of these campaigns contain possible links to North Korea.

Figure 33: The certificate used to sign the JAKU sample.



CA Root	Subordinate CA/Issuer	Owner	Status	Valid From	Valid To
thawte	thawte Primary Root CA	Xuchang Hongguang Technology Co.,Ltd. sha1/RSA (2048bits)	Revoked	7/18/2013	7/16/2014
thawte	thawte Primary Root CA	Ningbo Gaoxinqu zhidian Electric Power Technology Co., Ltd. sha1/RSA (2048bits)	Revoked	11/5/2013	11/5/2014

Figure 34: Stolen certificates used in the DarkHotel campaign, taken from Kaspersky's DarkHotel report.

Email Address: *

Submit