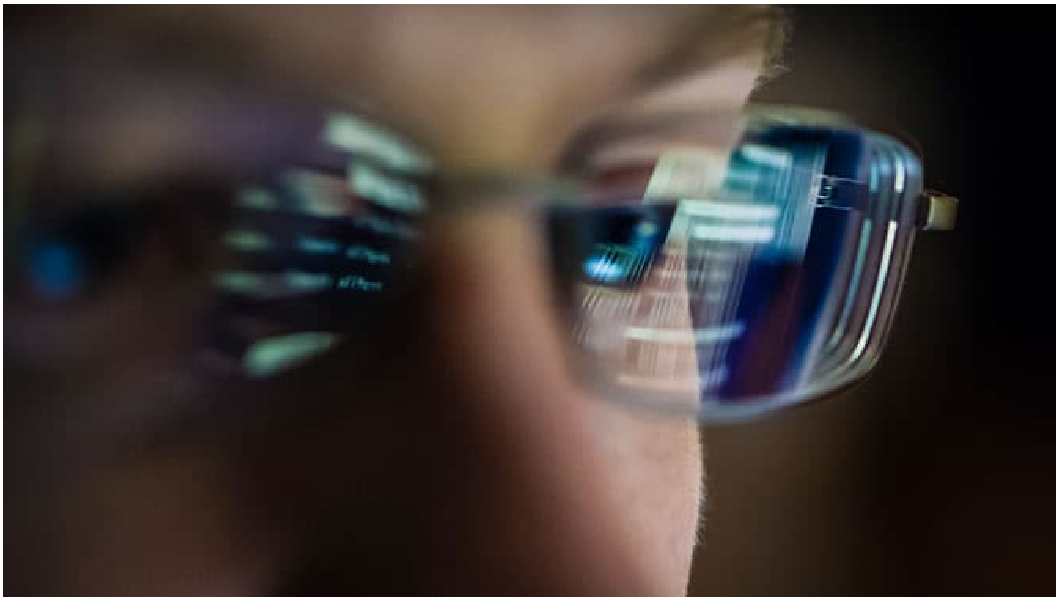
Chasing Shadows: A deep dive into the latest obfuscation methods being used by ShadowPad

pwc.co.uk/issues/cyber-security-services/research/chasing-shadows.html



08 December, 2021

By Adam Prescott, Cyber Threat Intelligence Analyst, PwC

While monitoring for the backdoor known as ShadowPad, our threat intel practice discovered a bespoke packing mechanism – which we named ScatterBee – being used to obfuscate malicious 32-bit and 64-bit payloads for ShadowPad binaries. The obfuscation mechanism has been briefly touched on in open source;[1] however in this blog we detail how the technique works, ways to analyse binaries obfuscated in this manner, and how to find further samples obfuscated with this bespoke method. This content has previously been made available privately to clients via PwC's intelligence subscription service.

During our analysis, further malicious samples were uncovered which indicate that one or more users of ShadowPad have access to ScatterBee, and have highly likely delivered some of these malicious payloads via watering hole attacks on sites that are used to deliver Adobe Flash update files.

Most of the malicious ScatterBee files can be directly linked back to a China-based threat actor that we are currently tracking as Red Dev 10.

Analysis

Throughout the rest of this blog we will detail a series of obfuscation techniques that, when combined, we assess is the result of a packing mechanism we call ScatterBee. The ScatterBee packing mechanism consists of control flow obfuscation, string encoding, dynamic API resolutions, several anti-analysis techniques and shellcode decoding/decrypting.

For anyone wanting to replicate the analysis detailed in this blog, we have provided an accompanying GitHub repository containing scripts and a walkthrough.

The malicious DLL loader

Discovery of the ScatterBee obfuscation began with a file tagged by ESET on an online multi-antivirus scanner as “a variant of Win32/Shadowpad.L”.

SHA-256	a8e5a1b15d42c4da97e23f5eb4a0adfd29674844ce906a86fa3554fc7e58d553
Filename	log.dll
File type	Win32 DLL
File size	209,408 bytes
Compilation timestamp	31/07/2020 08:08:43

This DLL exports one seemingly benign function called “log”, which just writes a given string to %TEMP%\log.txt, as well as exporting its entry point function.

The entry point function, which is automatically called when an executable loads this DLL, contains guardrails to make sure the executable loading log.dll has specific bytes at specific offsets, as seen in Figure 1.

Figure 1 - Checking bytes in calling executable

Searching for files with these bytes at these positions returns an MPRESS packed file that is likely a legitimate version of BDReinit.exe, a component of BitDefender. We have observed a similar guardrail technique in previous ShadowPad samples.[2]

Once the malicious DLL has verified it is being loaded by the target version of BDReinit.exe it will overwrite the parent executable’s entry point with a call into its own code.

Figure 2 - Overwriting the entry point of the parent executable

This is a common technique used by various malware families originating from China-based threat actors – notably in PlugX loaders – to gain execution of the malicious DLL’s code while running as the original and legitimate executable’s process.

Once the parent executable has finished loading its required DLLs, it will then execute code from its entry point, which now points to code in the malicious DLL. This is where the first unique obfuscation technique employed by ScatterBee is found.

Figure 3 - Calls to an obfuscated jump routine

Each of the calls to loc_100095f1 in Figure 3 are used to calculate where the next instruction to be executed is located. The code in this function makes use of pairs of inverted conditional branches to identical locations to further obfuscate how the destination is calculated, as seen in Figure 4.

Figure 4 - Opposing conditional branches

The result of the obfuscated code is to take the return address (the memory location immediately after the call) that is on the stack, get the next four bytes from memory, add them to the return address and then jump to the calculated address.

Figure 5 - Offsets used to calculate destination addresses

In the first highlighted example in Figure 5 the current return address is 0x100128c0; adding the 32-bit value 0xffff81ed to this address results in a target address of 0x1000aaad. From this point on every single instruction in the malicious DLL is followed by an obfuscated jump to the next address, preventing disassemblers from being able to follow the control flow of the sample. As a first attempt at deobfuscating the malicious code we replaced the calls to the obfuscated address calculation function by jmp instructions which jump to the correct location. The results of this can be seen in Figure 6.[3]

Figure 6 - Fixed control flow

The resulting code has similar instructions to a standard function epilogue (push ebp; mov ebp, esp) but then has a strange comparison instruction comparing the stack register – esp – to 0xe1cf. This is the second technique that ScatterBee employs to obfuscate control flow. Throughout the malicious code, the stack is compared to various low values and then a conditional jump is placed after the check. This fools disassemblers into thinking the code could take the jump if the current stack register is a small value. In practice, it is impossible for the stack register to be a small value, as on x86 and x64 systems the stack is placed in high memory ranges. Further, the targets of the conditional jumps are often into the middle of existing instructions, or to code halfway through functions which prevents disassemblers and decompilers from correctly analysing the flow of execution.

Both of these techniques are likely applied as part of a custom compiler pass[4] as they significantly modify the control flow of the binary, which is easiest to do before the final assembly instructions have been generated. It is uncommon for China-based actors to employ such extensive custom obfuscation techniques and indicates either a greater level of capability or a greater need to avoid detailed analysis once discovered than other China-based threat actors. Similar techniques have been seen used by financially motivated threat actors (e.g. DoppelPaymer ransomware binaries) who go to extreme lengths to avoid their malware being analysed.

There are several approaches that could aid in statically analysing code obfuscated in this way, however we have taken the route of rebuilding the malicious binary with the jump and stack obfuscations removed. In doing this, the resulting binary will be very close to what would be produced from compiling the original source code with a standard compiler.

Figure 7 - Deobfuscated code from Figure 6

The results of this deobfuscation can be seen in Figure 7.[5] This demonstrates the benefit of this approach as in Figure 6 only the first three meaningful instructions were able to be displayed in an analysis tool, whereas in the deobfuscated binary a plain disassembly listing is evident, showing many more instructions while taking up less space.

We chose to leave the stack comparison instructions in the deobfuscated binary while removing the fake branches for two reasons; firstly they do not affect execution of the sample as the obfuscation technique ensures they are never placed between a valid comparison instruction and its resulting conditional jump; and secondly each numerical value used in the obfuscated comparison instruction occurs exactly once in the original obfuscated sample; this means that when analysing the deobfuscated sample an analyst can verify that the output of the deobfuscation tool is accurate by searching for the constant value used in the original binary and checking the expected instructions in both binaries match up.

With a rebuilt binary, decompilation tools were then able to successfully analyse the malicious binary. The differences in outputs are clearly demonstrated in Figure 8 and Figure 9 with the same code being attempted to be decompiled in both figures.

Figure 8 - Decompilation before deobfuscation

Figure 9 - Decompilation after deobfuscation

The next obfuscation technique employed by ScatterBee is to resolve API functions dynamically at runtime. This is achieved by decoding strings specifying the library and function names required, then searching the Process Environment Block (PEB) for the kernel32 functions LoadLibraryA and GetProcAddress and using them to retrieve a pointer to the needed function. The string encoding algorithm is used extensively by ScatterBee obfuscated binaries for API call obfuscation, data obfuscation and string obfuscation.[6]

The encoding algorithm is a stream cipher that takes a 32-bit value as a seed and for each byte in the encoded string:

- Multiplies the current seed by 17;
- Subtracts the 32-bit constant value 0x443246ba from the seed;
- Stores the result as the seed for the next iteration; and,
- Sums each byte of the resulting seed to give the final XOR byte to use with the current encoded byte.

This algorithm will generate a pseudo random sequence of bytes that will be different for each seed used. Different values have been observed being used as the subtraction value in the algorithm. Sometimes the algorithm terminates when it decodes a null character, while other implementations have it run over a fixed number of bytes.

Once these obfuscation methods have been dealt with, it is possible to analyse the functionality of this malicious DLL. It will look for a file in the same folder called log.dll.dat and read the contents. The first four bytes of the file are a little-endian integer to use as the seed value with the previously described encoding algorithm. In this instance, the value 0x107e666d is added to the seed during each iteration instead of having 0x443246ba subtracted.[7]

A buffer is created in memory for the decoded payload, using VirtualAlloc with a length 4,096 bytes greater than the length of the payload. The extra space is so that the malware can generate a random number less than 4,096 via a call to QueryPerformanceCounter, and then use the value as an offset into the buffer to write the payload. This will prevent some detection methods that rely on malicious payloads being written at the start of memory segments and also hinder analysts in determining the entry point of the payload when analysing the sample dynamically.

The malicious payload

SHA-256	8065da4300e12e95b45e64ff8493d9401db1ea61be85e74f74a73b366283f27e
Filename	log.dll.dat
File type	Binary
File size	861,074 bytes

The payload is position independent shellcode that uses the same ScatterBee obfuscation techniques as the loader. After deobfuscating the payload to rebuild analysable code there are numerous calls to addresses that are outside the payload’s loaded memory (Figure 10).

Figure 10 - Calls using invalid memory locations

This is caused by a further obfuscation technique that is employed by ScatterBee shellcode to patch specific parts of the shellcode at run time. The logic for how the shellcode finds and applies the patches to its own memory is described below.

The first function that the shellcode calls searches through its own memory for a configuration data section by checking that there are six specific integer values consecutively in memory. It XORs every four bytes in the shellcode with 0xAD48FB1D, checking whether the following integer matches the result. Once a match is found it then checks that the next following integer, XORed with 0xE642D205, matches its subsequent integer value and that the integer after that, XORed with 0x868910EE, also matches its subsequent integer value. The valid data in this sample that signifies the start of the configuration information is shown in Figure 11.

Figure 11 – XOR bytes at start of config

The three integers that immediately follow these XOR bytes represent the size of the code section (0xC9000), data section (0x3000) and patch metadata section (0x5AD0) of the shellcode. It further checks the integrity of the payload by checking that the first byte of the shellcode is 0xE9, which corresponds to the initial jmp instruction used by the malware. This is designed to thwart a common malware analysis technique of loading a payload into memory with a breakpoint on the first instruction which has the effect of replacing the first byte (0xE9) with 0xCC.

Once the shellcode has passed these checks it uses the patch metadata section to overwrite data in its own memory. The metadata section is an array of pairs of four-byte integer values, the second integer value in each pair is used as the value to overwrite the four bytes in the shellcode at the offset specified by the first integer value.[8]

The same code from Figure 10 after the patching has been applied can be seen in Figure 12.

Figure 12 - Patched function calls

After we have removed the ScatterBee obfuscation layers from the shellcode, the final payload can be analysed in detail. In this instance, the payload matches what is described as ShadowPad.4 in open source.[9]

An example of configuration information in a 32-bit sample is shown in the following structure:

Table 1 - Configuration data structure

Offset	Size	Description
0x0	6 DWORDs	Used to mark the start of the config
0x18	DWORD	Size of code section of shellcode
0x1c	DWORD	Size of data section of shellcode
0x20	DWORD	Size of patch metadata section
0x24	DWORD	Space for pointer to obfuscated data written at runtime
0x28	DWORD	Value of 0,1,2 or 3 used to determine the operating mode of the backdoor
0x2c	DWORD	If set; target PID queried during backdoor operation
0x34	19 WORDs	An array containing relative offsets to obfuscated strings
0x5a	Six DWORDs	Null padding
0x72	4 WORDs	An array containing relative offsets to obfuscated strings
0x7a	16 BYTEs	0x08 repeated – reason unknown
0x8a	DWORD	Value 0x1e – reason unknown
0x8e	DWORD	Null padding
0x92	DWORD	Value 0x350b – reason unknown
0x96	10 DWORDs	Null padding
0xbe	Variable	Start of obfuscated string data used with relative offset arrays

Each of the offsets in the arrays at 0x34 and 0x72 in the configuration structure point to an obfuscated string that is used by the ScatterBee encoded ShadowPad payloads to specify sample specific variables such as C2s and filenames to use. The obfuscated strings consist of one WORD to use as a decoding seed, a WORD specifying the length of the encoded string, and then the encoded data.

Examples of each of these decoded strings with a description of possible usage is shown in the table below. The first 19 entries correspond to the array starting at 0x34 and the final four entries correspond to the array starting at 0x72.

Table 2 - Configuration strings

Description	Example data (multiple shown where configs have differences across samples)
Timestamp	“2020/10/26 16:31:13”, “6/30/2020 1:25:52 PM”
Campaign code	“Chrome.exe”, “ccc”
Filepath	“%ALLUSERSPROFILE%\\DRM\\Chrome\\”, “%PROGRAMDATA%\\”
Spoofed name	“Chrome.exe”, “msdn.exe”
Loader filename	“log.dll”
Payload filename	“log.dll.dat”
Service name	“Chrome_update”, “WMNetworkSvc”
Alternative service name	“Chrome_update”, “WMNetworkSvc”
Alternative service name	“Chrome_update”, “WMNetworkSvc”

Registry key path	“SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run”
Possibly service description	“Chrome_update”, “WMSVC”
Program to inject into	“%ProgramFiles%\\Windows Media Player\\wmplayer.exe”
Alternative injection target	“%windir%\\system32\\svchost.exe”
Alternative injection target	“%windir%\\system32\\winlogon.exe”
Alternative injection target	“%windir%\\explorer.exe”
C2	“TCP://207.148.98[.]61:443”
Alternative C2	“UDP://cigy2jft92[.]kasprsky[.]info:53”
Alternative C2	Empty string
Alternative C2	Empty string
Proxy info string	“SOCKS4\n\n\n\n\n”
Proxy info string	“SOCKS4\n\n\n\n\n”
Proxy info string	“SOCKS5\n\n\n\n\n”
Proxy info string	“SOCKS5\n\n\n\n\n”

64-bit variations

As alluded to previously, we also found 64-bit versions of ShadowPad obfuscated with ScatterBee. The following table shows the details of one of the 64-bit loaders.

SHA-256	c72436969d708905901ac294d835abb1c4513f8f26cb16c060d2fd902e1d5760
Filename	secur32.dll
File type	Win64 DLL
File size	142,848 bytes
Compilation timestamp	2040-04-30 05:32:44

Whereas all of the 32-bit loaders found so far have had the filename log.dll, we have found that the 64-bit ScatterBee loaders are named either mscoree.dll or secur32.dll. The functionality of these loaders is identical to the 32-bit variants, in that they search the current directory for a file with the same name as themselves with “.dat” appended (secur32.dll.dat or mscoree.dll.dat), then deobfuscate and load it into memory.

In 32-bit versions of ScatterBee loader files, there are a limited number of strings in plaintext in the .data section of the malicious binary, along with plaintext stack strings for kernel32, LoadLibraryA and GetProcAddress, whereas in the 64-bit samples there are no strings relating to the ScatterBee encoded sections. Some 64-bit ScatterBee files also employ a different encoding algorithm to the stream cipher in various places, and may hint that several different users of ScatterBee have added their own take on obfuscation to the tool.

The encoding algorithm predominantly seen in 64-bit ScatterBee samples uses a combination of MD5 and AES to decode data. The process is as follows:

- Takes 16 hardcoded bytes from the shellcode, along with the final four bytes of the encoded data and MD5 hash them;
- Uses CryptDeriveKey with the resulting hash as input to generate an AES-128 key; and,
- Calls CryptDecrypt on the encoded data (minus the final four bytes used to create the key).

The below table shows the details of an encoded, 64-bit ScatterBee payload.

SHA-256	943778353ce3af1043ec161ef18c9ba3e1ad6a9915dfe1783dff7aac8b53df16
Filename	secur32.dll.dat
File type	Binary
File size	608,679 bytes

The configuration data in these samples starts with a similar section of six DWORDs that are used as XOR markers, along with the sizes of the code, data, and patch sections of the payload. However the subsequent data is a series of obfuscated chunks.

Each chunk begins with a four-byte marker that contains the chunk ID in the high byte, and the length of the chunk in the lowest three bytes. For example, the DWORD 0x80000774 has a chunk type of 0x80 and a length of 0x774 bytes. The chunks are decoded by using either of the previously described algorithms - the stream cipher, or the MD5 and AES algorithm.

In the payloads that we have access to, these chunks contain various different IDs. The chunks with 0x80 as their chunk type contain similar data to the 32-bit configuration data,[10] although the encoded strings can use either the AES encoding algorithm or the stream cipher algorithm. Chunks with an ID of 0x02 contain 0x20 bytes of unknown data followed by a valid PE file. These PE files are ShadowPad modules that further enhance the capabilities of the running backdoor. In files seen by PwC, some are obfuscated with ScatterBee techniques and some are not. We have not seen any other IDs in chunks from samples that we have analysed, however, from the code in the ShadowPad backdoor it supports further chunks with IDs of 0x83, 0x84, 0x90, 0x91, 0x92 and 0xa0.

Further malicious files

Pivoting on the names of the DLL loaders and the code sequences used to calculate the obfuscated jumps uncovered 25 malicious DLLs obfuscated with ScatterBee and 10 further malicious payload files that use ScatterBee obfuscation/packing.

Pivoting on the stack comparison code also uncovered trojanised flash installers, a malicious loader and a ZIP archive (detailed in Table 3). All of these malicious files are part of an execution chain that executes variants of ShadowPad, and so far PwC has not found any files obfuscated with ScatterBee that do not deliver ShadowPad, likely indicating that ScatterBee is a core part of the build process of one or more ShadowPad users.

Table 3 - Early stage ScatterBee files

SHA-256	Description
f7ef194f2dcc341ba03f76872cb7c0dfbae8f79118f99cf73dfccfb146c4e966	Drops and executes a DLL search order hijacked Oleview.exe.
39f92aed5dfa2cd20ae7df11e16acce9bb2e80c7e6539bc81f352d42ab578eb6	Trojanised flash installer.
ebe4347e993c81d145b68a788522d5c554edfa74c35e9e61ededd6c510e80c75	Trojanised flash installer.
dbb02aaea56a1f0200b76f3f5b2d3596622503633285c7798b4248e0a558f01c	ZIP archive containing Oleview.exe along with a malicious DLL and payload.

The trojanised installers both contain the same logic for executing their embedded payloads. The initial file is a 64-bit Windows executable that writes two files from its resource section to disk in the folder returned by GetTempPathA. The names and descriptions of the files dropped are as follows:

- A hack tool, named “Microsoft.Win32.TaskScheduler.dll”, for carrying out operations relating to Windows Task Scheduler[11]; and,
- A malicious second stage loader named “td.Principal.UserId =.exe”.

SHA-256	2a3cf204dcc977df6347a039428ae863066700cecfac965dcaeb7b9bd61bc1b6
Filename	td.Principal.UserId =.exe
File type	64-bit .NET executable
File size	9,757,184 bytes
Compilation timestamp	2042-06-11 00:17:24

The Task Scheduler hack tool is not executed by this or any other stage seen by PwC, and is highly likely an artefact of the build process that supports a persistence mechanism not used by this sample. However, the second stage loader ("td.Principal.UserId =.exe") is executed by the trojanised installer in a call to CreateProcessA. This second stage loader is a .NET executable responsible for dropping and executing a legitimate Adobe Flash installer and a DLL search order hijacked copy of Oleview.exe, as well as creating a task that runs as a LogonTrigger.

First, the malicious loader will attempt to disable all network adapters returned by a query of "SELECT * From Win32_NetworkAdapter". Then, it reads five resources from its own resources section and writes them to disk as the following:

Table 4 - Dropped files

File path	SHA-256
%TMP%\OLEVIEW.exe[12]	2e642afdd36c129e6b50ae919ca608ac0006ce337f2a5a7a6fb1eef6a4ad99e7
%TMP%\IVIEWERS.dll	e328060057f454232aab79a2c521414ee110c13925ac53e1bfacd7f2155e38d2
%TMP%\IVIEWERS.dll.dat	9cbfa03a65e6cd4b62b2aa60a4cc4785b824378f735de2596a1195b75f71ecf3
%TMP%\helper.exe	f4effcf4d7321be824fd637b27f404250d0b1f03205bbc0682022d61aba5801e
%TMP%\flashplayerax_install_cn_fc.exe	c4edf7b8cdffb67fcd62ef81485c04648b11a14a8452f407133f131e2f74a57a

Next it will create a new TaskDefinition (registered as "FlashUpdate") with an action that is triggered by LogonTrigger with the following details:

Table 5 - Persistence task details

Field	Value
RegistrationInfo.Description	Adobe Tech.co
RegistrationInfo.Author	Adobe Tech.co
Principal.UserId	system
Actions	ExecAction with an argument of %TMP%\helper.exe

With the persistence task registered, the .NET executable reenables the network adapters and creates three processes to execute the dropped .exe files.

The first three files in Table 4 are a DLL search order hijacking triplicate of files with similar functionality to the ScatterBee files described earlier in this report. When the legitimate Oleview.exe is executed by the .NET executable it will load the malicious IVIEWERS.dll, which will in turn load and execute the malicious ScatterBee obfuscated ShadowPad payload contained in IVIEWERS.dll.dat.

flashplayerax_install_cn_fc.exe is also executed by the .NET executable and is a legitimate Adobe Flash installer.

helper.exe is a binary written in Go, which acts as a HTTP server and serves up the response “Hello!” when any client connects to it. It is highly likely that this is another artefact left in the loader by accident, or that is still under development, to allow the threat actor to gain persistence on the victim machine via a secondary backdoor.

The file f7ef194f2dcc341ba03f76872cb7c0dfbae8f79118f99cf73dfccfb146c4e966, from Table 3, is a similar dropper to the first stage of the trojanised installers; however, in this case it simply drops the three OLEVIEW related files straight to disk and executes them.

These first stage droppers also have strings and logic embedded in them to support dropping and executing two further files that were not present in these samples - %TMP%\flsh.exe and %TMP%\schost.exe.

Among the additional DLLs discovered, there was a cluster of eight files that stand out from the rest.

Table 6 – Xiamen submitted DLLs

SHA-256	Filename
8396e35b19f906f9c6e342e6cd90ab8bbbecc90f9090b0afe68f4fa53530bc33	ALTTEST.dll
15371908d89caef3f4487298a452e58732d9f671f2c6a1f07036d123ce3c840d	ALTTEST.dll
a41348407e01886e76baf7cb8bb0efcf790b213cab87924b8a4f6bf8a9502350	ALTTEST.dll
02a18df00e241f82cecb7477f661ebe3f26012cdfc5b8172d634c07af4468130	ALTTEST.dll
7c8b6dfcdbcb6e0d87513eec841302a202e7371cdff16101d1594ea34a8dd1af	ATLTEST.dll[13]
c951a1d1294c46c995189dce4a70da0460dd19c0b7136a4905f41212cdead0c7	ALTTEST.dll
f768bd36e88ffa496e7b6c538f2259cbdab0317e88432a99050f550b4c9f2f12	ALTTEST.dll
c738af04c5b531abdb303a68cfb8994bb8db6e088bf99b45f85bdb863d3fb3e5	MyDRes.dll

All of the files in Table 6 were submitted to an online multi-antivirus scanner from locations in Xiamen, China. All of the files have an exported DLL name of Dll.dll, and all apart from the last one were also submitted by the same submitter ID within the space of about 20 minutes. Each of these files are slightly different DLLs: some are MFC binaries, some are meant to be run as Service DLLs; however, all of them contain almost identical copies of ScatterBee packed shellcode to load a .dat file into memory. Only two of these samples have code that would enable the ScatterBee shellcode to run if loaded with an appropriate executable file:

- f768bd36e88ffa496e7b6c538f2259cbdab0317e88432a99050f550b4c9f2f12; and,
- c738af04c5b531abdb303a68cfb8994bb8db6e088bf99b45f85bdb863d3fb3e5.

All the other files either, will not run the packed shellcode, or would require another loader beyond just an executable importing their DLL.

The clustering of file submissions from the same location, the similarity of the files exported names, the presence of almost identical copies of ScatterBee packed shellcode, the mixture of functioning and none-functioning samples, and the submission name of ALTTEST.dll in many of these samples all add weight to the possibility that a developer or user of ScatterBee is based in Xiamen, and was testing and/or developing the ScatterBee packer during January 2021. Alternatively, there is a possibility that these submissions are from a researcher related to Positive Technologies, as their public blog on this malware family was published the day after these submissions to the online multi-antivirus scanner.

Targeting

Based on submissions to an online multi-antivirus scanner of the obfuscated payloads, it is highly likely that the threat actor using the ScatterBee obfuscated ShadowPad binaries has targeted:

- A military organisation in Afghanistan;

- An aviation organisation in Hong Kong; and,
- A company with a security operations centre based in India.

There are also numerous submissions from users based in China, some of which may represent testing whether the current version of the malicious file is detected by antivirus vendors, and others that are likely organisations based in China that are being targeted by a ShadowPad user. This targeting is consistent with our historical tracking of ShadowPad victims, based on communications with known command and control servers.

Infrastructure

When extracting the ShadowPad payloads from the ScatterBee encoded payloads we found the following C2s in use in the configuration sections of the backdoors:

Table 7 - ScatterBee encoded ShadowPad C2s

SHA-256	Configured C2s
5f1a21940be9f78a5782879ad54600bd67bfcd4d32085db7a3e8a88292db26cc	cigy2jft92[.]kasprsky[.]info
0371fc2a7cc73665971335fc23f38df2c82558961ad9fc2e984648c9415d8c4e	ti0wddsnv[.]wikimedia[.]vip
c602456fae02510ff182b45d4ffb69ee6aae1166746000124168580 7db2e29c3	6czumi0fbg[.]symantecupd[.]com
04089c1f71d62d50cbd8009dfd557aa1e6db1492a9fa2b35902182c07a0ed1c1	yjij4bpade[.]nslookup[.]club
8065da4300e12e95b45e64ff8493d9401db1ea61be85e74f74a73b366283f27e	207.148.98[.]61
fb17b3886685887aeb8f7c3496c6f7ef06702ec1232567278286c2f8ec4351bb	172.18.165[.]105 (private IP)
943778353ce3af1043ec161ef18c9ba3e1ad6a9915dfe1783dff7aac8b53df16	kazehaya0110[.]chickenkiller[.]com
7579e864d47898f1322bb189bdd21b537b40e549149318ce8409f1d57233fa48	fljhcqwe[.]com
9cbfa03a65e6cd4b62b2aa60a4cc4785b824378f735de2596a1195b75f71ecf3	a[.]fljhcqwe[.]com

The first four domains in Table 7 were already tracked by us as Red Dev 10, and have resolved to IP addresses that have previously shown up in our scans for ShadowPad C2s. Pivoting on these domains and IPs uncovers a highly connected set of infrastructure that includes the following domains, most of which also have numerous subdomains that have been observed used as C2 addresses in other variants of ShadowPad.

Table 8 - Red Dev 10 domains

Domains
dnslookup[.]services
livehost[.]live
windowshostnamehost[.]club
kasprsky[.]info
symantecupd[.]com
wikimedia[.]vip
nslookup[.]club

Red Dev 10 has made a habit of using NameCheap and Namesilo when registering its domains, and this activity follows that pattern. In addition, the subdomains under several of these domains also follow a pattern of having between 8 and 12 random alphanumeric characters, which, combined with domains registered by NameCheap and Namesilo that resolve to IP addresses assigned to The Constant Company, as well as being parked resolving to 127.0.0[.]1 when not in use, allows analysts to pivot and find more potentially malicious domains.

While investigating this cluster of infrastructure, several of the domains shared self-signed SSL certificates that were themed around Microsoft. This, together with the domain names chosen in Table 8, shows a general pattern of trying to spoof the legitimacy of infrastructure employed by these campaigns.

The remaining C2s from Table 7 are not easily linked together beyond being found in ScatterBee encoded ShadowPad samples, which leaves open the possibility that there may be multiple groups using the packer, or that for some operations that greater care is taken to compartmentalise the activity.

Putting together the use of ShadowPad (predominantly a tool used by China-based threat actors), C2 infrastructure that we have previously tracked as Red Dev 10, and the likely targeting of targets aligning to previous ShadowPad usage, we assess that most of this activity is highly likely Red Dev 10, with the possibility that a small subset of this activity could be an as yet unknown China-based threat actor.

Conclusion

PwC has been tracking ShadowPad since 2017 and has observed numerous evolutions of the technical capability. During this time, there has also been widespread reporting about its use in supply chain attacks. Despite this, multiple threat actors continue to use ShadowPad for long term compromise of sensitive organisations, including in the military and telecommunications sectors. This activity aligns extremely closely to the threat actor we track as Red Dev 10, which is a known ShadowPad user.

The ScatterBee obfuscation technique documented in this report is likely the latest attempt to minimise detection in victim networks. Whether this technique is exclusively used by one threat actor, or a general development of ShadowPad capability, remains to be seen.

More detailed information on each of the techniques used in this blog, along with mitigations, can be found on the following MITRE pages:

[1] 'Higaisa or Winnti? APT41 backdoors, old and new', Positive Technologies, https://www.ptsecurity.com/ww-en/analytics/pt-esc-threat-intelligence/higaisa-or-winnti-apt-41-backdoors-old-and-new/ [illegible] 2021)
[2] [illegible] memory dumps last forever
[3] See https://github.com/PwCUK-CTO/ScatterBee_Analysis/blob/main/Scripts/ScatterJump.py for the IDA plugin that fixes these jumps
[4] Writing an LLVM Pass [illegible] what-is-a-pass
[5] See https://github.com/PwCUK-CTO/ScatterBee_Analysis/blob/main/Scripts/ScatterRebuildPayload.py for an IDA python script which can rebuild ScatterBee code
[6] See https://github.com/PwCUK-CTO/ScatterBee_Analysis/blob/main/Scripts/ScatterDecodeAPICalls.py for an IDA python script to rename the functions that call API functions
[7] See https://github.com/PwCUK-CTO/ScatterBee_Analysis/blob/main/Scripts/ScatterDecodePayload.py for a script that can take an encoded payload file and decode it to its ScatterBee encoded shellcode
[8] See GitHub repo ScatterBeePatch.py for a python script that applies the patches to a payload file.
[9] [illegible] BackDoor.ShadowPad.4 [illegible]
[10] [illegible] however they contain all the same information as previously detailed in the 32-bit analysis. Of note [illegible] removed from the configuration.
[11] GitHub [illegible] TaskScheduler
[12] The legitimate OleView.exe file is always seen named in capitals when dropped by ScatterBee related files.
[13] The spelling variation here was present in the name of the file submitted

Appendix A – Indicators of compromise

Indicator	Type
9cbfa03a65e6cd4b62b2aa60a4cc4785b824378f735de2596a1195b75f71ecf3	SHA-256
dbb02aaea56a1f0200b76f3f5b2d3596622503633285c7798b4248e0a558f01c	SHA-256
d29113e3417dcba9d0e2d540fc53f702869dc7dc018a6b053bc3f70b4e55e436	SHA-256
5f1a21940be9f78a5782879ad54600bd67bfcd4d32085db7a3e8a88292db26cc	SHA-256
0371fc2a7cc73665971335fc23f38df2c82558961ad9fc2e984648c9415d8c4e	SHA-256
fb17b3886685887aeb8f7c3496c6f7ef06702ec1232567278286c2f8ec4351bb	SHA-256
26de542f77da51071389463fad1a50c687b70d902bbd0800db6c959e40dff755	SHA-256
8065da4300e12e95b45e64ff8493d9401db1ea61be85e74f74a73b366283f27e	SHA-256
c0fbb71af4863db0cd82942974957088908f815ef7f02b197834e22d02d4a460	SHA-256
c0aae2d5e77acb8b35037f3cd3b76e92eebdb1c53cf3775921bd6f64d94e9a99	SHA-256

991511785a05f4dfbf1212e3fb69ff3b666659ecba5f3e5e9c8fbe9804afd23c	SHA-256
943778353ce3af1043ec161ef18c9ba3e1ad6a9915dfe1783dff7aac8b53df16	SHA-256
7579e864d47898f1322bb189bdd21b537b40e549149318ce8409f1d57233fa48	SHA-256
c951a1d1294c46c995189dce4a70da0460dd19c0b7136a4905f41212cdead0c7	SHA-256
7c8b6dfcdbcb6e0d87513eec841302a202e7371cdff16101d1594ea34a8dd1af	SHA-256
c602456fae02510ff182b45d4ffb69ee6aae11667460001241685807db2e29c3	SHA-256
5e7e336bc7b489c3d4c59af861580ed73a5731d26560488bce03befdef9faadf	SHA-256
c72436969d708905901ac294d835abb1c4513f8f26cb16c060d2fd902e1d5760	SHA-256
dbb32cb933b6bb25e499185d6db71386a4b5709500d2da92d377171b7ff43294	SHA-256
37417f300e1382b5b1b93e0be675ba8ab2d418747ea3fa015329f7ca405ae603	SHA-256
c738af04c5b531abdb303a68cfb8994bb8db6e088bf99b45f85bdb863d3fb3e5	SHA-256
ffc5bc143ab2320ae6989ccdf8c37a3d7c3c51c09eabf5a94ada86ab7c3abebd	SHA-256
a8e5a1b15d42c4da97e23f5eb4a0adfd29674844ce906a86fa3554fc7e58d553	SHA-256
1e06fd5b9aa0e5260369e52ec2d9f87060941de835234afd198b1d4c0b161678	SHA-256
7cbd4339c33af40c70d27256cf3ec473bea588ac33ddfa64a8771344c82d9e6c	SHA-256
cb5f8759831829614b82ed4a3bf1ac3f27f1640faf2a1f15ba728751e2fa44fa	SHA-256
04089c1f71d62d50cbd8009dfd557aa1e6db1492a9fa2b35902182c07a0ed1c1	SHA-256
531e54c055838f281d19fed674dbc339c13e21c71b6641c23d8333f6277f28c0	SHA-256
042541cc39bafdcb0565ee468359ef575256f5adfda0e53c915ecdbbedd91316	SHA-256
5a151aa75fbfc144cb48595a86e7b0ae0ad18d2630192773ff688ae1f42989b7	SHA-256
f768bd36e88ffa496e7b6c538f2259cbdab0317e88432a99050f550b4c9f2f12	SHA-256
8d1a5381492fe175c3c8263b6b81fd99aace9e2506881903d502336a55352fef	SHA-256
a41348407e01886e76baf7cb8bb0efcf790b213cab87924b8a4f6bf8a9502350	SHA-256
f8c5e93d6114f5a69d1544504d9d7f6a1d7397e3e5e0cce8e24e6d7b884c109e	SHA-256
2a3cf204dcc977df6347a039428ae863066700cecfac965dcaeb7b9bd61bc1b6	SHA-256
15371908d89caef3f4487298a452e58732d9f671f2c6a1f07036d123ce3c840d	SHA-256
96dc16bbc0f3e6e80fba447e3a3e1085fddf8e97edf286ee8b3fd82954f565bb	SHA-256
39f92aed5dfa2cd20ae7df11e16acce9bb2e80c7e6539bc81f352d42ab578eb6	SHA-256
8396e35b19f906f9c6e342e6cd90ab8bbbecc90f9090b0afe68f4fa53530bc33	SHA-256
ebe4347e993c81d145b68a788522d5c554edfa74c35e9e61ededd6c510e80c75	SHA-256
02a18df00e241f82cecb7477f661ebe3f26012cdfc5b8172d634c07af4468130	SHA-256
f7ef194f2dcc341ba03f76872cb7c0dfbae8f79118f99cf73dfccfb146c4e966	SHA-256
f4effcf4d7321be824fd637b27f404250d0b1f03205bbc0682022d61aba5801e	SHA-256
06539163f71f8bd496db75ccb41db820	MD5

493698b1d7acfbf57848b964b4b0ae97	MD5
69be59f365f74b406e505a8c0e128047	MD5
bf98b795957d40ed8e0c52403af659d2	MD5
8b9436c358a1d7f0ca61eca81b5025f7	MD5
4ad23aae3409c31d3d72e1d10e9d957d	MD5
ffbadead054d1eac270f1a24d02e8a1f	MD5
3520e591065d3174999cc254e6f3dbf5	MD5
a22fce6e7c1b2d129602ff938a2ac039	MD5
ad82d23accb10b4c0fc7f8c9782ae6ad	MD5
2a4976a82a07016bd1b5de1a372d8e15	MD5
3e372906248b215ea0ee853cb4e29dd8	MD5
ab8b13f3a93baaa36b730cb42434620a	MD5
67329d4239551b51c481062b5d38a687	MD5
18b391d91883979fc2df9e13c8aee075	MD5
529e9edc37b668e13be6b077a399f195	MD5
42988a0bd2bbdf4454d5d15a2733aa31	MD5
ea6be331b5fa349a2fa464b062043b0e	MD5
d50b9ca68a3a650016e64ab4c3ff8e4c	MD5
409b27c8eab8b043cfe8854ca22799b3	MD5
70477683ea5a7e193bb80c6cf01da8dd	MD5
373eacf3ffd1b5722f9d3c1595092b4c	MD5
d7e153c2957a519a1ee6734820e5efbd	MD5
9563df80a0f9709baa909c25bdd64214	MD5
64cc83ba22f67c6c8c82c162f64a7c92	MD5
25f3713b9ff40b7fb1293213916c1dbc	MD5
c486da41dda4f55f5bafa4f22d877495	MD5
af10f874ee9a24d4a8d5e515af9c24a2	MD5
9d3aaaf04c684bf6c90ada2030ceaea3	MD5
21779cdfbe7ce838d3adc11f42b64191	MD5
5f3093473ae4167fd51d4282fce73741	MD5
42794ad1300ed3edb1ed2d1a473b77ad	MD5
52c28bdb6b1fc4d77b1ea58dc8c1c810	MD5
73790e781a0b3c7f1e1e8f9fa8f9d239	MD5
5fe99a8f8cbfe46832478aa9c9634ed6	MD5

263b7fb02bb4c05c789d2c1de92e0007	MD5
24f73d5f67bc6cf0bccaade97e04fbca	MD5
d2b97a3391c91d1577fb46963b8ef18a	MD5
af78467a6cdbb4efa3894a30edef608b	MD5
9d3a9edec791cb3eb7225be225337c1e	MD5
7c8c3700757ddb5c6d423d88dd944065	MD5
4d6705979b4ba29e44d3178ac979e1c6	MD5
5fcdb89a3b2eb7ff31c5122e8f145277	MD5
ff46982c58cf9cd0371e187a6c0dd6f7712c084c	SHA-1
880fa69a6efd8de68771d3df2f9683107fb484c0	SHA-1
0cfba69898627c620575cadfff92130429dcd019	SHA-1
ea43dbef69af12404549bc45fda756bfefcb3d88	SHA-1
cad05dec778a6dbdeb170a63bbbd18271b56d719	SHA-1
addf67b8bcb8074927431bdfe3e3c867b07f5333	SHA-1
7db78548aae9e4872b06ee9e79c29553947db3d6	SHA-1
c73329dfbe99de4abb93b4fda6310a0c5eedd8f9	SHA-1
47cdaf6c5c3fffeeff1f2c9e6c7649f99ab54932	SHA-1
3342ad3a686be7a873409ae01cfab2eb0b621840	SHA-1
215404d27c6a63a47561d6ab5258af26843b1769	SHA-1
34ce0df62814e3a2430784836914c629d49f22b1	SHA-1
c62b977c93979effb48a1614956c2a788abb22fe	SHA-1
fa397effbb1d2d9b276d9d109e79ef89790729bc	SHA-1
6512750a9da8c81c6b7c5b5301a60d4962c0c41b	SHA-1
b885b9c4a9cd7872cd995198834471e52219ae41	SHA-1
f8e4b7bd1cc973be7540f7310289530734307 59a	SHA-1
6966687463365f08cfb25fd2c47c6e9a27af22b0	SHA-1
9605ad1bf0432ffb148d422099e23eaa26bed4c8	SHA-1
30c63b1e252ea0dc72b97785c1874ab7b6ddef43	SHA-1
48daf01f86cfc9f22c446d602f0cdbd4b763dfc8	SHA-1
b73134449329fd640a6de94a36cbcbebb4d5f541	SHA-1
363e32fafd2732b3cfb53dfd39bef56da1affd7f	SHA-1
e96759fcb766744a7aae9692947b4ed4ba77ce37	SHA-1
55811e2fade5fa4412bd5ff7f17eca79887d6aff	SHA-1
a36e63f41ee3fdfaf2a826c0b6e7728af546981e	SHA-1

44fc5b13ac3947a3be3fff7808d5d664d7258cb9	SHA-1
03a47494b76aa6feed68053e44c0a2fde6172ea5	SHA-1
494d8239650f3acb0b946f0d00f6dbc9c2c05be0	SHA-1
1c997ddb204bc597f937a07665511ae7d9d98661	SHA-1
c227d3cdcb39b56eddb7ab62d0da62f006207764	SHA-1
d4086a747566d5a7b0e80f0c977e1e6db3410d26	SHA-1
e2898e362dd19a0fb6f317d559cbdb78eac6488c	SHA-1
9853fe35e1b6e06b53ad2234d4fa2156fa5ccf97	SHA-1
f6f6f352fa58d587c644953e4fd1552278827e14	SHA-1
b224ae9ffd8119d773dedb1863d46725c29143f8	SHA-1
7cd459821ef2daea764df2f52c896e6ab00ed263	SHA-1
3f2ec5d5ae8be0394baff82bd5c08fcf8df0e754	SHA-1
fd492b013d52e061f101b6086c5c4902abb4b0e0	SHA-1
ba985d268bca9ff3bf0b09ab63085b57f52d3574	SHA-1
1bbc81db4d2d98a1cf29d4f84d065c6556f7caed	SHA-1
12118603b97e6b3d3a8cb6e48ec7351e160da445	SHA-1
93fec58769f40285b5a76106377644924d0c1dd0	SHA-1
5zsi53pi6uu[.]livehost[.]live	Domain
coivo2xo[.]livehost[.]live	Domain
ui79zm8o9b[.]livehost[.]live	Domain
qrvc7pdnbf[.]symantecupd[.]com	Domain
pow2u24h7[.]wikimedia[.]vip	Domain
vt[.]livehost[.]live	Domain
c5t7dvucq[.]symantecupd[.]com	Domain
1dfpi2d8kx[.]wikimedia[.]vip	Domain
dns[.]dnslookup[.]services	Domain
bsyu[.]dnslookup[.]services	Domain
2og8qfrkrk[.]symantecupd[.]com	Domain
test[.]wikimedia[.]vip	Domain
dust[.]dnslookup[.]services	Domain
dntc[.]livehost[.]live	Domain
fljhcqwe[.]com	Domain
5q4qp9trwi[.]dnslookup[.]services	Domain
www[.]livehost[.]live	Domain

bj0wyck5v5[.]livehost[.]live	Domain
7ec8txihoa[.]dnslookup[.]services	Domain
wikimedia[.]vip	Domain
4yti11wlo5[.]livehost[.]live	Domain
cigy2jft92[.]kasprsky[.]info	Domain
6q4qp9trwi[.]dnslookup[.]services	Domain
sci[.]livehost[.]live	Domain
524ce3dm8h[.]symantecupd[.]com	Domain
lmogv[.]dnslookup[.]services	Domain
dlbo92v2ef[.]livehost[.]live	Domain
bctu[.]dnslookup[.]services	Domain
wcuhk[.]livehost[.]live	Domain
hccadkml89[.]dnslookup[.]services	Domain
r1d3wg7xofs[.]livehost[.]live	Domain
jn3thp2wl6[.]symantecupd[.]com	Domain
d89o0gm34t[.]livehost[.]live	Domain
coivotek[.]livehost[.]live	Domain
a[.]fljhcqwe[.]com	Domain
evbyo7jj0v[.]livehost[.]live	Domain
www[.]wikimedia[.]vip	Domain
bm2l41risv[.]livehost[.]live	Domain
wntc[.]livehost[.]live	Domain
69gy9k6wc2[.]symantecupd[.]com	Domain
wvt[.]livehost[.]live	Domain
m2[.]livehost[.]live	Domain
dns[.]livehost[.]live	Domain
8hh3aktk2[.]kasprsky[.]info	Domain
1160idswz5[.]kasprsky[.]info	Domain
files[.]windowshostnamehost[.]club	Domain
8hh3aktk[.]kasprsky[.]info	Domain
wiki[.]windowshostnamehost[.]club	Domain
windowshostnamehost[.]club	Domain
6lh9bgi4n[.]symantecupd[.]com	Domain
v2ray[.]windowshostnamehost[.]club	Domain

5s2zm07ao[.]wikimedia[.]vip	Domain
b3d3fn9n[.]kasprsky[.]info	Domain
6czumi0fbg[.]symantecupd[.]com	Domain
ns2[.]windowshostnamehost[.]club	Domain
dbtwcse10sd[.]kasprsky[.]info	Domain
mx[.]windowshostnamehost[.]club	Domain
wfftm5kcj[.]kasprsky[.]info	Domain
wlamazcsrv1[.]windowshostnamehost[.]club	Domain
cde858l2yf[.]kasprsky[.]info	Domain
bnmyphvq[.]wikimedia[.]vip	Domain
local[.]windowshostnamehost[.]club	Domain
juv0cumdo3[.]kasprsky[.]info	Domain
felzeaxrs8hd[.]kasprsky[.]info	Domain
c2[.]windowshostnamehost[.]club	Domain
687eb876e047[.]kasprsky[.]info	Domain
a6olaxgd[.]kasprsky[.]info	Domain
ur1lwzh2qp[.]kasprsky[.]info	Domain
hostmaster[.]wikimedia[.]vip	Domain
bc[.]windowshostnamehost[.]club	Domain
db311secsd[.]kasprsky[.]info	Domain
arress[.]windowshostnamehost[.]club	Domain
www[.]kasprsky[.]info	Domain
7hln9yr3y6[.]symantecupd[.]com	Domain
vwlamazcsrv1[.]windowshostnamehost[.]club	Domain
v3hagesrj[.]symantecupd[.]com	Domain
z16sxt822s[.]symantecupd[.]com	Domain
dnslookup[.]services	Domain
ybk47i6z8q[.]wikimedia[.]vip	Domain
d89o0gm35t[.]livehost[.]live	Domain
zk4c9u55[.]wikimedia[.]vip	Domain
dsyu[.]livehost[.]live	Domain
wsyu[.]livehost[.]live	Domain
sc[.]livehost[.]live	Domain
w0eew6nkmb[.]livehost[.]live	Domain

r315imowtg[.]symantecupd[.]com	Domain
o56n1tosy[.]livehost[.]live	Domain
ti0wddsnv[.]wikimedia[.]vip	Domain
symantecupd[.]com	Domain
wctu[.]livehost[.]live	Domain
4iiiessb[.]wikimedia[.]vip	Domain
tei1sw0d98[.]symantecupd[.]com	Domain
livehost[.]live	Domain
nslookup[.]club	Domain
kasprsky[.]info	Domain
60.250.18[.]188	IPv4
141.164.35[.]117	IPv4
139.180.135[.]175	IPv4
66.42.44[.]130	IPv4
182.162.136[.]235	IPv4
128.199.232[.]13	IPv4
182.16.112[.]226	IPv4
149.28.145[.]214	IPv4
207.148.78[.]244	IPv4
207.148.99[.]56	IPv4
149.28.152[.]196	IPv4
139.180.135[.]200	IPv4
158.247.219[.]236	IPv4
207.148.98[.]61	IPv4
45.76.100[.]224	IPv4
139.180.187[.]35	IPv4
158.247.217[.]102	IPv4
45.76.148[.]41	IPv4
141.164.61[.]70	IPv4
141.164.63[.]174	IPv4
202.182.96[.]238	IPv4
139.180.141[.]227	IPv4

158.247.206[.]194	IPv4
139.180.156[.]26	IPv4
112.121.168[.]2	IPv4
141.164.62[.]81	IPv4
108.160.134[.]80	IPv4
5bcd1346428b6d7f1f19c0f175d96800c5a0951d	SSL SHA-1 fingerprint
743f1ef860a1cad5c046cb0099c479acf6815b97	SSL SHA-1 fingerprint
61c39c6c60f7a45ff18806ed855985ef48d954ef	SSL SHA-1 fingerprint
f1f5fe0dd96e165e049b8a7d508ccd951c7cca0b	SSL SHA-1 fingerprint
9575b444beeed7a16d639223b08e18e29b5eb5a4	SSL SHA-1 fingerprint
c9b276bd2166c95726fbe33f126fa0a014f84a36	SSL SHA-1 fingerprint
5aa19bfcbc980d65df184e644053bf4732929d8e	SSL SHA-1 fingerprint
log.dll.dat	Filename
secur32.dll.dat	Filename
mscoree.dll.dat	Filename

Related content

Cyber security: Research

Explore our latest cyber security research, developed to share new knowledge on critical cyber security issues.

Contact us

Form

Hide