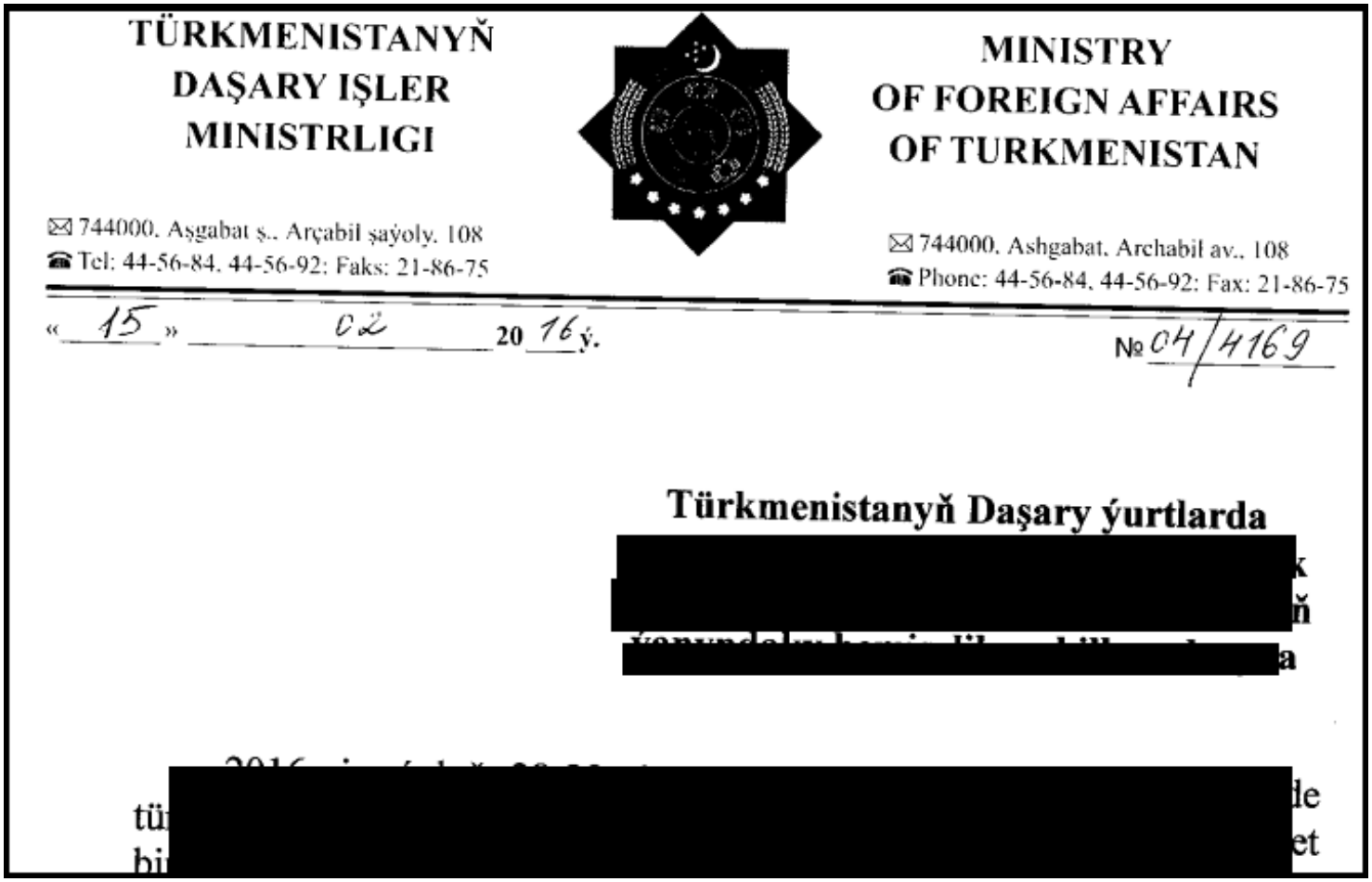
Introducing WhiteBear

securelist.com /introducing-whitebear/81638/

By GReAT

As a part of our Kaspersky APT Intelligence Reporting subscription, customers received an update in mid-February 2017 on some interesting APT activity that we called WhiteBear. Much of the contents of that report are reproduced here. WhiteBear is a parallel project or second stage of the Skipper Turla cluster of activity documented in another private intelligence report “Skipper Turla – the White Atlas framework” from mid-2016. Like previous Turla activity, WhiteBear leverages compromised websites and hijacked satellite connections for command and control (C2) infrastructure. As a matter of fact, WhiteBear infrastructure has overlap with other Turla campaigns, like those deploying Kopiluwak, as documented in “KopiLuwak – A New JavaScript Payload from Turla” in December 2016. WhiteBear infected systems maintained a dropper (which was typically signed) as well as a complex malicious platform which was always preceded by WhiteAtlas module deployment attempts. However, despite the similarities to previous Turla campaigns, we believe that WhiteBear is a distinct project with a separate focus. We note that this observation of delineated target focus, tooling, and project context is an interesting one that also can be repeated across broadly labeled Turla and Sofacy activity.

From February to September 2016, WhiteBear activity was narrowly focused on embassies and consular operations around the world. All of these early WhiteBear targets were related to embassies and diplomatic/foreign affair organizations. Continued WhiteBear activity later shifted to include defense-related organizations into June 2017. When compared to WhiteAtlas infections, WhiteBear deployments are relatively rare and represent a departure from the broader Skipper Turla target set. Additionally, a comparison of the WhiteAtlas framework to WhiteBear components indicates that the malware is the product of separate development efforts. WhiteBear infections appear to be preceded by a condensed spearphishing dropper, lack Firefox extension installer payloads, and contain several new components signed with a new code signing digital certificate, unlike WhiteAtlas incidents and modules.


TÜRKMENISTANYŇ
DAŞARY IŞLER
MINISTRLIGI

✉ 744000. Aşgabat ş., Arçabil şaýoly. 108
☎ Tel: 44-56-84, 44-56-92; Faks: 21-86-75

MINISTRY
OF FOREIGN AFFAIRS
OF TURKMENISTAN

✉ 744000, Ashgabat, Archabil av., 108
☎ Phone: 44-56-84, 44-56-92; Fax: 21-86-75

«15» 02 2016 ý. № 04/4169

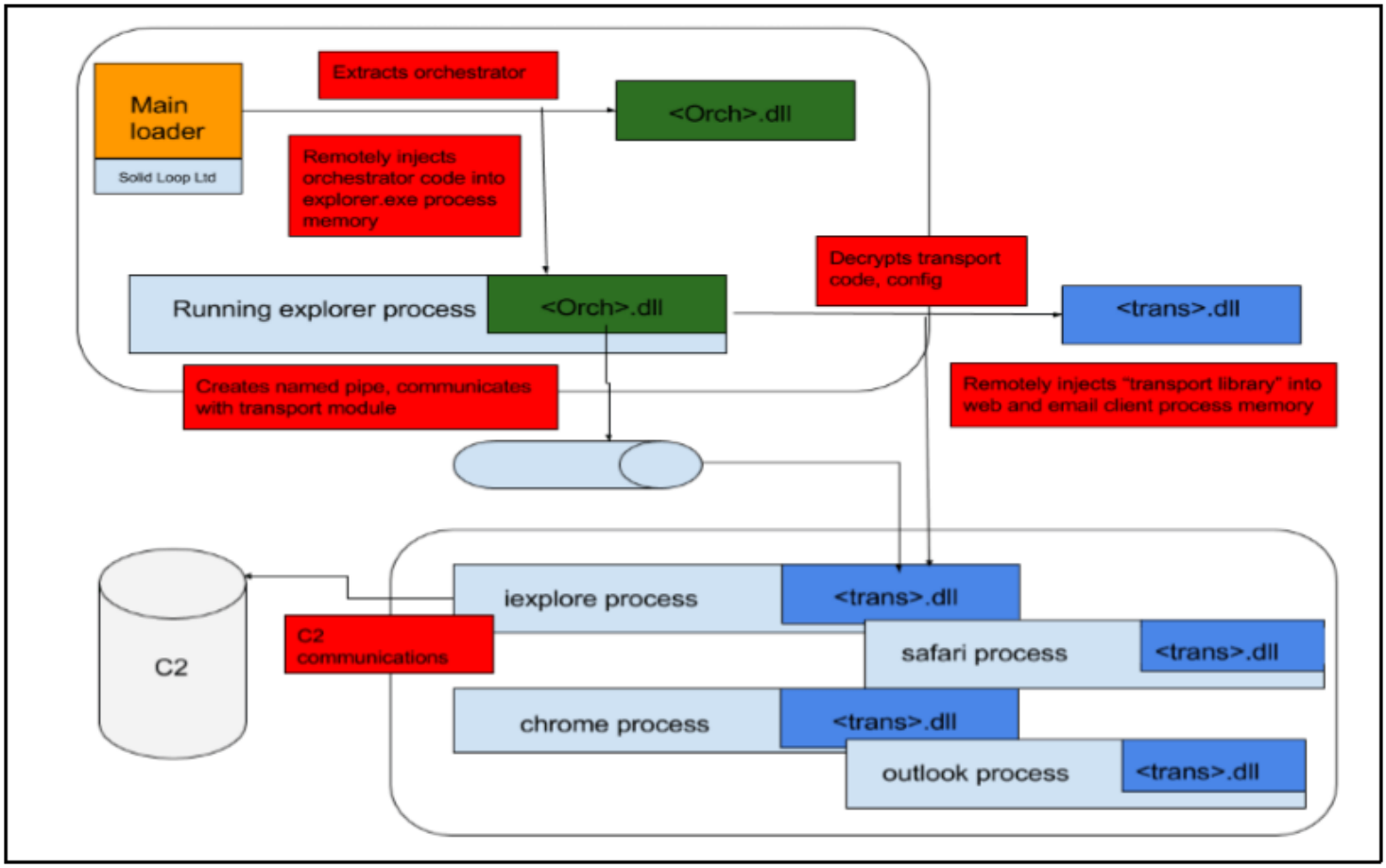
Türkmenistanyň Daşary ýurtlarda

[redacted]

The exact delivery vector for WhiteBear components is unknown to us, although we have very strong suspicion the group spearphished targets with malicious pdf files. The decoy pdf document above was likely stolen from a target or partner. And, although WhiteBear components have been consistently identified on a subset of systems previously targeted with the WhiteAtlas framework, and maintain components within the same filepaths and can maintain identical filenames, we were unable to firmly tie delivery to any specific WhiteAtlas component. WhiteBear focused on various embassies and diplomatic entities around the world in early 2016 – tellingly, attempts were made to drop and display decoy pdf's with full diplomatic headers and content alongside executable droppers on target systems.

Technical Details

The WhiteBear platform implements an elaborate set of messaging and injection components to support full presence on victim hosts. A diagram helps to visualize the reach of injected components on the system.



WhiteBear Binary loader

Sample MD5: b099b82acb860d9a9a571515024b35f0
Type PE EXE
Compilation timestamp 2002.02.05 17:36:10 (GMT)
Linker version 10.0 (MSVC 2010)
Signature "Solid Loop Ldt" UTCTime 15/10/2015 00:00:00 GMT – UTCTime 14/10/2016 23:59:59 GMT

The WhiteBear binary loader maintains several features including two injection methods for its (oddly named) "KernelInjector" subsystem, also named by its developer
– Standart
– WindowInject (includes an unusual technique for remotely placing code into memory for subsequent thread execution)

The loader also maintains two methods for privilege and DEP process protection handling:
– GETSID_METHOD_1
– GETSID_METHOD_2

The binary contains two resources:
– BINARY 201
– File size: 128 bytes
– Contains the string, "explorer.exe"
– BINARY 202
– File size: 403456 bytes
– File Type: PE file (this is the actual payload and is not encrypted)
– This PE file resource stores the "main orchestrator" .dll file

Loader runtime flow

The loader creates the mutex “{531511FA-190D-5D85-8A4A-279F2F592CC7}”, and waits up to two minutes if it is already present while logging the message “IsLoaderAlreadyWork +”. The loader creates the mutex “{531511FA-190D-5D85-8A4A-279F2F592CC7}”, and waits up to two minutes. If it is already present while logging the message “IsLoaderAlreadyWork +”, it extracts the resource BINARY 201. This resource contains a wide string name of processes to inject into (i.e. “explorer.exe”).

The loader makes a pipe named: \\.\pipe\Winsock2\CatalogChangeListener-%03x%01x-%01x

Where the “%x” parameter is replaced with the values 0xFFFFFFFF 0xEEEEEEEE 0xDDDDDDDD, or if it has successfully obtained the user’s SID:
\\.\pipe\Winsock2\CatalogChangeListener-%02x%02x-%01x
With “%x” parameters replaced with numbers calculated from the current date and a munged user SID.

The pipe is used to communicate with the target process and the transport module; the running code also reads its own image body and writes it to the pipe. The loader then obtains the payload body from resource BINARY 202. It finds the running process that matches the target name, copies the buffer containing the payload into the process, then starts its copy in the target process.

There are some interesting, juvenile, and non-native English-speaker debug messages compiled into the code:
– i cunt waiting anymore #%d
– lights aint turnt off with #%d
– Not find process
– CMessageProcessingSystem::Receive_NO_CONNECT_TO_GAYZER
– CMessageProcessingSystem::Receive_TAKE_LAST_CONNECTION
– CMessageProcessingSystem::Send_TAKE_FIN

WhiteBear Main module/orchestrator

Sample MD5: 06bd89448a10aa5c2f4ca46b4709a879
Type, size: PE DLL, 394 kb
Compilation timestamp: 2002.02.05 17:31:28 (GMT)
Linker version: 10.0 (MSVC 2010)
Unsigned Code

The main module has no exports, only a DllMain entry which spawns one thread and returns. The main module maintains multiple BINARY resources that include executable, configurations, and encryption data:

101 – RSA private (!) key
102 – RSA public key
103 – empty
104 – 16 encrypted bytes
105 – location (“%HOMEPATH%\ntuser.dat.LOG3”)
106 – process names (e.g. “iexplore.exe, firefox.exe, chrome.exe, outlook.exe, safari.exe, opera.exe”) to inject into
107 – Transport module for interaction with C&C
108 – C2 configuration
109 – Registry location (“\HKCU\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Explorer\Screen Saver”)
110 – no information
111 – 8 zero bytes

Values 104 – 111 are encrypted with the RSA private key (resource 101) and compressed with bzip2.4. The RSA

key is stored with header stripped in a format similar to Microsoft's PVK; the RSA PRIVATE KEY header is appended by the loader before reading the keys into the encryption code. Resource 109 points to a registry location called "external storage", built-in resources are called "PE Storage".

In addition to storing code, crypto resources, and configuration data in PE resources, WhiteBear copies much of this data to the victim host's registry. Registry storage is located in the following keys. Subkeys and stored values listed below:
[HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\ScreenSaver]
[HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Explorer\ScreenSaver]

Registry subkeys:
{629336E3-58D6-633B-5182-576588CF702A} Contains the RSA private key used to encrypt/decrypt other resources / resource 101
{3CDC155D-398A-646E-1021-23047D9B4366} Resource 105 – current file location
{81A03BF8-60AA-4A56-253C-449121D61CAF} Resource 106 – process names
{31AC34A1-2DE2-36AC-1F6E-86F43772841F} Contains the internet C&C transport module / resource 107
{8E9810C5-3014-4678-27EE-3B7A7AC346AF} Resource 108 – C&C config
{28E74BDA-4327-31B0-17B9-56A66A818C1D} Resource 110 "plugins"
{4A3130BD-2608-730F-31A7-86D16CE66100} Resource 111
{119D263D-68FC-1942-3CA3-46B23FA652A0} Unique Guid ("ObjectID")
{1DC12691-2B24-2265-435D-735D3B118A70} "Task Queue"
{6CEE6FE1-10A2-4C33-7E7F-855A51733C77} "Result Queue"
{56594FEA-5774-746D-4496-6361266C40D0} unknown
{831511FA-190D-5D85-8A4A-279F2F592CC7} unknown

Finally, if the main WhiteBear module fails to use registry storage, it uses "FS Storage" in file %TEMP%\KB943729.log. The module reads all of its data and binary components from one of the storages and then verifies the integrity of data (RSA+bzip2 compression+signature).

The module maintains functionality which is divided into a set of subsystems that are loosely named by the developers:
• result queue
• task queue
• message processing system
• autorun manager
• execution subsystem
• inject manager
• PEStorage
• local transport manager/internal transport channel

It creates the following temporary files:
%TEMP%\CVRG72B5.tmp.cvr
%TEMP%\CVRG1A6B.tmp.cvr
%TEMP%\CVRG38D9.tmp.cvr

%TEMP%\~DF1E05.tmp contains the updated body of the loader during an update.

Every day (as specified by local time) the main module restarts the transport subsystem which includes:
• message processing
• named pipe transport ("NPTransport")

If the registry/file storage is empty, the module performs a 'migration' of hardcoded modules and settings to the

storage location. This data is encrypted with a new RSA key (which is also stored in the registry).

The data in the registry is prepended with a 0xC byte header. The maximum size of each registry item is 921,600 bytes; if the maximum size is exceeded, it is split into several items. The format of the header is shown below: [4:service DWORD][4:chunk index][4:chunk size including header]

Every time the orchestrator module is loaded it validates that the storage area contains the appropriate data and that all of the components can be decrypted and validated. If these checks fail the module reinstalls a configuration from the resource “REINSTALL”.

Pipe Transport

The module generates the pipe name (with the same prefix as the loader); waits for incoming connections; receives data and pushes it to the ‘message processing system’. The module generates the pipe name (with the same prefix as the loader); waits for incoming connections; receives data and pushes it to the ‘message processing system’. Every packet is expected to be at least 6 bytes and contain the following header: [4:ID][2:command]

List of commands:
1 : new task
2 : update the loader + orchestrator file
4 : send task result
5 : send settings
6 : write results to registry/file storage
7 : enable / disable c2 transport / update status
8 : uninstall
9 : nop
10 : “CMessageProcessingSystem::Receive_NO_CONNECT_TO_GAYZER”; write results to registry
11: write the last connection data ‘{56594FEA-5774-746D-4496-6361266C40D0}’ aka “last connection” storage value
12: “give cache” – write cached commands from the C&C
13: “take cache” – append C&C commands to the cache

Depending on the command, the module returns the results from previously run tasks, the configuration of the module, or a confirmation message.

An example of these tasks is shown below:
• write a file and execute it with CreateProcess() capturing all of the standard output
• update C&C configuration, plugin storage, etc
• update autoruns
• write arbitrary files to the filesystem (“File Upload”)
• read arbitrary files from the filesystem (“File Download”)
• update itself
• uninstall
• push task results to C2 servers

The “LocalTransport manager” handles named pipe communication and identifies if the packet received is designated to the current instance or to someone else (down the route). In the latter scenario the LocalTansport manager re-encrypts the packet, serializes it (again), and pushes the packet via a named pipe on the local network to another hop, (NullSessionPipes). This effectively makes each infected node a packet router.

The Autorun manager subsystem is responsible for tracking the way that the malicious module starts in the system and it maintains several different methods for starting automatically (shown below):

LinkAutorun The subsystem searches for a LNK file in the target directory, changes the path to “cmd.exe” and the description to ‘ /q /c start “” “%s” && start “” “%s” ‘
TaskScheduler20Autorun The subsystem creates the ITaskService (works only on Windows Vista+) and uses the ITaskService interface to create a new task with a logon trigger
StartupAutorun The subsystem creates a LNK file in %STARTUP%
ScreenSaverAutorun The subsystem installs as a current screensaver with a hidden window
HiddenTaskAutorun The subsystem creates the task ITaskScheduler (works only on pre-Vista NT). The task trigger start date is set to the creation date of the Windows directory
ShellAutorun Winlogon registry [HKCU\Software\Microsoft\Windows NT\CurrentVersion\Winlogon] Shell=”explorer.exe, …”

File Uninstallation is done in a discreet manner. The file is filled with zeroes, then renamed to a temporary filename before being deleted

WhiteBear Transport library (aka “Internet Relations”, “Pipe Relations”)

Sample MD5: 19ce5c912768958aa3ee7bc19b2b032c
Type: PE DLL
Linker timestamp: 2002.02.05 17:58:22 (GMT)
Linker version: 10.0
Signature “Solid Loop Ldt” UTCTime 15/10/2015 00:00:00 GMT – UTCTime 14/10/2016 23:59:59 GMT

This transport library does not appear on disk in its PE format. It is maintained as encrypted resource 107 in the orchestrator module, then decrypted and loaded by the orchestrator directly into the memory of the target process. This C2 interaction module is independent, once started, it interacts with the orchestrator using its local named pipe.

To communicate with its C2 server, the transport library uses the system user agent or default “Mozilla/4.0 (compatible; MSIE 6.0)”.

Before attempting a connection with its configured C2 server, the module checks if the victim system is connected to Internet by sending HTTP 1.1 GET / requests to the following servers (this process stops after the first successful connection):
• update.microsoft.com
• microsoft.com
• windowsupdate.microsoft.com
• yahoo.com
• google.com

If there is no Internet connection available, the module changes state to, “CANNOT_WORK” and notifies the peer by sending command “7” over the local pipe.

The C2 configuration is obtained from the main module with the command “5”. This checks whether the module complies with the schedule specified in the C2 settings (which includes inactivity time and the interval between connections). The C2 interaction stages have interesting function names and an odd misspelling, indicating that the developer may not be a native English speaker (or may have learned the English language in a British setting):
“InternetRelations::GetInetConnectToGazer”
“InternetRelations::ReceiveMessageFromCentre”
“InternetRelations::SendMessageToCentre”
“PipeRelations::CommunicationTpansportPipe”

The module writes the encrypted log to %TEMP%\CVRG38D9.tmp.cvr The module sends a HTTP 1.0 GET request through a randomly generated path to the C2 server. The server’s reply is expected to have its MD5 checksum

appended to the packet. If C2 interaction fails, the module sends the command “10” (“NO_CONNECT_TO_GAYZER”) to the orchestrator.

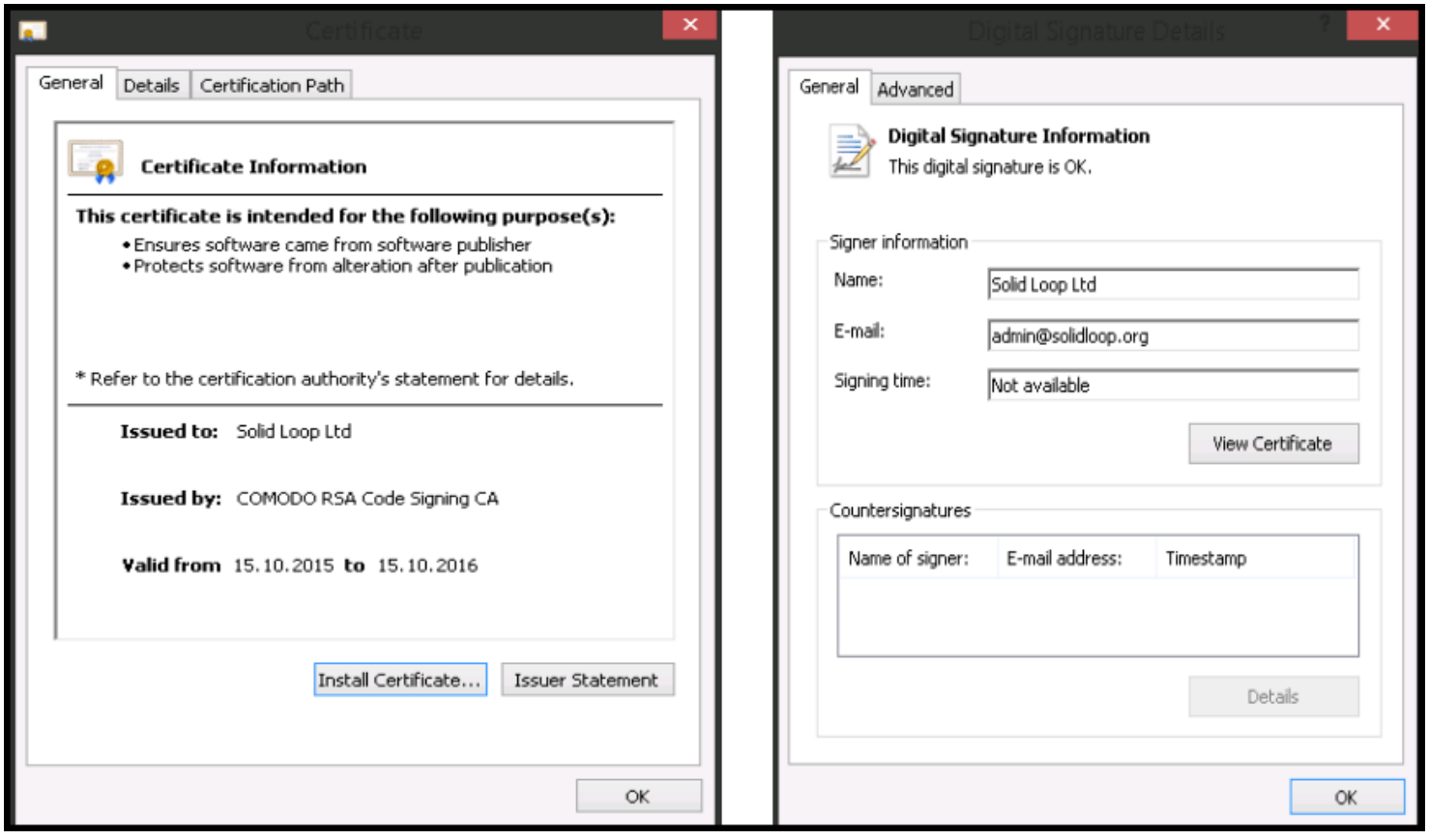
Unusual WhiteBear Encryption

The encryption implemented in the WhiteBear orchestrator is particularly interesting. We note that the resource section is encrypted/decrypted and packed/decompressed with RSA+3DES+BZIP2. This implementation is unique and includes the format of the private key as stored in the resource section. 3DES is present in Sofacy and Duqu2 components, however they are missing in this Microsoft-centric RSA encryption technique. The private key format used in this schema and RSA crypto combination with 3DES is (currently) unique to this threat actor.

The private key itself is stored as a raw binary blob, in a format similar to the one Microsoft code uses in PVK format. This format is not officially documented, but its structures and handling are coded into OpenSSL. This private key value is stored in the orchestrator resources without valid headers. The orchestrator code prepends valid headers and passes the results to OpenSSL functions that parse the blob.

Digital Code-Signing Certificate – Fictional Corporation or Assumed Identity?

Most WhiteBear samples are signed with a valid code signing certificate issued for “Solid Loop Ltd”, a once-registered British organization. Solid Loop is likely a phony front organization or a defunct organization and actors assumed its identity to abuse the name and trust, in order to attain deceptive code-signing digital certificates.



WhiteBear Command and Control

The WhiteBear C2 servers are consistent with long standing Turla infrastructure management practices, so the backdoors callback to a mix of compromised servers and hijacked destination satellite IP hosts. For example, direct, hardcoded Turla satellite IP C2 addresses are shown below:

C2 IP Address	Geolocation	IP Space Owner
169.255.137[.]203	South Sudan	IPTEC, VSAT
217.171.86[.]137	Congo	Global Broadband Solution, Kinshasa VSAT
66.178.107[.]140	Unknown – Likely Africa	SES/New Skies Satellites

Targeting and Victims

WhiteBear targets over the course of a couple years are related to government foreign affairs, international organizations, and later, defense organizations. The geolocation of the incidents are below:

- Europe
- South Asia
- Central Asia
- East Asia
- South America

Conclusions

WhiteBear activity reliant on this toolset seems to have diminished in June 2017. But Turla efforts continue to be run as multiple subgroups and campaigns. This one started targeting diplomatic entities and later included defense related organizations. Infrastructure overlap with other Turla campaigns, code artifacts, and targeting are consistent with past Turla efforts. With this subset of 2016-2017 WhiteBear activity, Turla continues to be one of the most prolific, longstanding, and advanced APT we have researched, and continues to be the subject of much of our research. Links to publicly reported research are below.

Reference Set

Full IOC and powerful YARA rules delivered with private report subscription

Md5
b099b82acb860d9a9a571515024b35f0
19ce5c912768958aa3ee7bc19b2b032c
06bd89448a10aa5c2f4ca46b4709a879

IP
169.255.137[.]203
217.171.86[.]137
66.178.107[.]140

Domain(s)
soligro[.]com – interesting because the domain is used in another Turla operation (KopiLuwak), and is the C2 server for the WhiteBear transport library
mydreamhoroscope[.]com

Example log upon successful injection

|01:58:10:216|.[0208|WinMain]..
|01:58:14:982|.[0209|WinMain].**********
|01:58:15:826|.[0212|WinMain].DATE: 01.01.2017
|01:58:21:716|.[0215|WinMain].PID=2344.TID=1433.Heaps=3

|01:58:22:701|.[0238|WinMain].CreateMutex = {521555FA-170C-4AA7-8B2D-159C2F491AA4}
|01:58:25:513|.[0286|GetCurrentUserSID]._GETSID_METHOD_1_
|01:58:26:388|.[0425|GetUserSidByName].22 15 1284404594 111
|01:58:27:404|.[0463|GetUserSidByName].S-1-5-31-4261848827-3118844265-2233733001-1000
|01:58:28:263|.[0471|GetUserSidByName].
|01:58:29:060|.[0165|GeneratePipeName].\\.\pipe\Winsock2\CatalogChangeListener-5623-b
|01:58:29:763|.[0275|WinMain].PipeName = \\.\pipe\Winsock2\CatalogChangeListener-5623-b
|01:58:30:701|.[0277|WinMain].Checking for existence…
|01:58:31:419|.[0308|WinMain].— Pipe is not installed yet
|01:58:32:044|.[0286|GetCurrentUserSID]._GETSID_METHOD_1_
|01:58:32:841|.[0425|GetUserSidByName].22 15 1284404594 111
|01:58:33:701|.[0463|GetUserSidByName].S-1-5-31-4261848827-3118844265-2233733001-1000
|01:58:34:419|.[0471|GetUserSidByName].
|01:58:35:201|.[0318|WinMain].Loading…
|01:58:35:763|.[0026|KernelInjector::KernelInjector].Address of marker: 0x0025F96C and cProcName: 0x0025F860
|01:58:36:513|.[0031|KernelInjector::KernelInjector].Value of marker = 0xFFFFFEF4
|01:58:37:279|.[0088|KernelInjector::SetMethod].m_bAntiDEPMethod = 1
|01:58:38:419|.[0564|QueryProcessesInformation].OK
|01:58:41:169|.[0286|GetCurrentUserSID]._GETSID_METHOD_1_
|01:58:42:076|.[0425|GetUserSidByName].22 15 1284404594 111
|01:58:42:748|.[0463|GetUserSidByName].S-1-5-31-4261848827-3118844265-2233733001-1000
|01:58:43:169|.[0471|GetUserSidByName].
|01:58:43:701|.[0309|FindProcesses].dwPID[0] = 1260
|01:58:44:560|.[0345|WinMain].try to load dll to process (pid=1260))
|01:58:45:013|.[0088|KernelInjector::SetMethod].m_bAntiDEPMethod = 1
|01:58:45:873|.[0094|KernelInjector::LoadDllToProcess].MethodToUse = 1
|01:58:46:544|.[0171|KernelInjector::GetProcHandle].pid = 1260
|01:58:47:279|.[0314|KernelInjector::CopyDllFromBuffer].Trying to allocate space at address 0x20020000
|01:58:48:404|.[0332|KernelInjector::CopyDllFromBuffer].IMAGEBASE = 0x20020000.ENTRYPOINT = 0x2002168B
|01:58:48:763|.[0342|KernelInjector::CopyDllFromBuffer].ANTIDEP INJECT
|01:58:49:419|.[0345|KernelInjector::CopyDllFromBuffer].Writing memory to target process….
|01:58:49:935|.[0353|KernelInjector::CopyDllFromBuffer].Calling to entry point….
|01:58:51:185|.[0598|KernelInjector::CallEntryPoint].CODE = 0x01FA0000, ENTRY = 0x2002168B, CURR = 0x77A465A5, TID = 1132
|01:58:55:544|.[0786|KernelInjector::CallEntryPoint]._FINISH_ = 1
|01:58:56:654|.[0372|KernelInjector::CopyDllFromBuffer].CTRLPROC = 0
|01:58:57:607|.[0375|KernelInjector::CopyDllFromBuffer].+ INJECTED +
|01:58:58:419|.[0351|WinMain].+++ Load in 1260

References – past Turla research

The Epic Turla Operation
Satellite Turla: APT Command and Control in the Sky
Agent.btz: a Source of Inspiration?
The ‘Penquin’ Turla
Penquin’s Moonlit Maze
KopiLuwak: A New JavaScript Payload from Turla

Uroburos: the snake rootkit [pdf]