MANAGED MALWARE ANALYSIS REPORT

Rerport no.	**1705001**
Performed	30.05.2017
File	Sample.exe
Size	275456
MD5	8b6d824619e993f74973eedfaf18be78
SHA1	0f04dad5194f97bb4f1808df19196b04b4aee1b8
SHA256	972e907a901a7716f3b8f9651eadd65a 0ce09bbc78a1ceacff6f52056af8e8f4
SHA512	cb61fc9c58d8ed1cf3a40fa676c1a1d685a09a920beca2b 577266ce17bdf9f7ee14927b5c33d4e23daf27d72f6ac32 ed4071570af88f83de39823acfb9785422
Contact	contact@korrino.com

Contents

1 Executive summary

The sample has been retrieved by a client from an infected node and submitted to Korrino for analysis. In order to infer about earlier elements of attack vector (before the infection), additional analysis needs to be performed (see: 1.2). Analysis has started 15.05.2017 and has been completed 30.05.2017.

1.1 Purpose of the sample

Upon completion of analysis of the sample the following conclusions have been made, based on detailed results described in section 3:

1. Sample is a part of a larger malicious architecture. It performs relaying activity on victims operating system. Other components of malicious architecture, e.g. keyloggers, info stealers, etc.. can use this application for reporting to remote nodes.
2. Sample's functionality includes screen grabbing. This makes information stealing scenario highly probable.
3. There are no indicators of a particular class of data being targeted. Nonetheless, based on conclusion that it's only a part of a broader architecture, it is recommended to identify and analyze other applications in architecture in order to make conclusions about targeted data (see: 1.2).
4. The sample is a sophisticated malware. It's designed as part of a modular solution. It uses efficient, thread-based architecture. The code has been rated as of high quality. That suggests that substantial investment has been made in order to build it. This in turn means that suspicions about the actor behind this attack are moving towards state sponsored actors rather than less capable actors.
5. Malicious architecture is highly configurable. Its capabilities might be easily expanded upon download and installation of other modules.

1.2 Recommended further actions

Based on conclusions of this report, following actions are recommended (beginning with immediately required actions):

1. In order to disrupt attacker communication with infected nodes, we recommend blocking network communication patterns by administrators (based on Indicators of Compromise enclosed in Appendix A – Indicators of C).

2. In order to locate infections, gather evidence and sanitize compromised nodes we recommend detecting infected machines by administrators and/or users (based on Indicators of Compromise enclosed in Appendix A – Indicators of C), starting with high profile systems.
3. In order to retrieve remaining applications of malicious architecture we recommend performing forensic analysis on infected machines. Forensic analysis should aim at retrieving more suspicious artifacts and perform analysis on them if necessary.
4. In order to gain more actionable information regarding origin and purpose of malicious application, we recommend performing analysis on identified loadable and executable artifacts.
5. In order to prepare for future attack vectors with highest risk levels, we recommend performing threat modeling and risk analysis for selected highly valuable systems and assets.
6. In order to raise security level for attack scenarios with highest calculated risk, we recommend to implement appropriate security controls based on results from recommendation 5.

1.3 Remaining issues

Sample contains one object that has very limited functionality. During C&C communication simulation the analyst was unable to simulate commands from malicious actor (botmaster) that would enable any functionality in this object. There is suspicion that most of this object's functions were removed from sample before its compilation and performing the attack.

2 About the technical summary

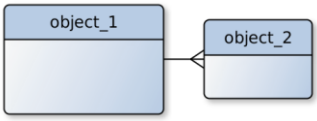
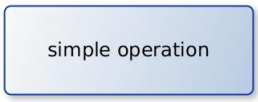
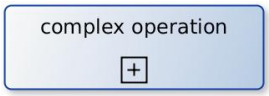
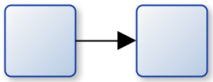
The technical summary of analysis is divided into two perspectives. The first is the architecture overview, the most general overview of the application. The purpose of this perspective is to answer the question: "How is this sample built?"

The second is operation overview. This part is focused on general functionality as well as individual operations being performed by the sample and referenced back to parts of its architecture. The purpose of this perspective is to answer the question: "How does this sample work?"

The process of reversing compiled sample's functionality is very complex. In order to simplify presenting its conclusions, graphical models are being used throughout this report. These models are based on ERD notation and BPMN notation. However, various elements can be used slightly differently than in everyday use of this notations. These differences are made for the sake of greater readability. For the same reason parts of objects attributes, functions and methods that are employed at assembly level are being omitted.

2.1.1 Graphic models

In order to visualize interpretations of selected constructions and operations, graphic representations are provided. Two of most frequently used classes of these representations are: object models (used mainly in 3.1) and operation models (used in 3.4). The building blocks of models are described in the following table.

Object symbol	Meaning
	Represents relation one to many between two objects. In this case one object *object_1* is in relation one-to-many with object *object_2* (e.g. *object_1* contains list of *object_2*)
	Represents complex operation pool. Subsequent suboperations are being executed within it. In this report it is referred to as “operation”.
	Represents simple operation. Simple operation is a representation of a sequence of assembly instructions that constitute a single logical entity.
	Represents complex operation. Complex operation is described in detail in another diagram in a separate operation pool with a corresponding caption.
	Represents system interaction. System interaction occurs when described process is interacting with its environment (e.g. filesystem) via library calls or system calls instead of performing instructions.
	Represents operation’s starting point. In terms of machine language it represents procedure prologue.
	Represents operation’s stopping point. In terms of assembly language it represents procedure epilogue.
	Represents program’s termination point. It means that program being described is terminating.
	Represents sleeping point. It means that thread execution has been suspended conditionally or unconditionally for a certain amount of time.
	Represents signaling point. It means that thread is performing synchronization activities with another thread or another program, e.g. sending a signal via Windows *SetEvent* syscall.
	Represents communication point. It means that thread is performing communication operations, such as *recv* function from *ws2_32.dll* library.
	Represents conditional exclusive gateway. It means that process control flow can select one of two directions based on a condition being met.
	Represents conditional complex gateway. It means that process control flow can select one of many directions based on a condition being met.
	Represents an important annotation for one of described process elements, e.g. reference for analysis artifact.
	Describes control being passed from one operation to another subsequent one.

2.1.2 Execution visualizations

In order to investigate detailed program executions used to formulate general operations overview, execution visualizations are provided.

Execution visualization can be viewed and edited with FreeMind software[1] (available for free). It contains hierarchical representation of a section of a sample's recorded execution. Blue entries describe called functions detected as library calls or identified internal calls of the sample. Black entries represent unidentified calls. Red entries represent interesting (suspicious) calls.

Diagram 1: Example of execution visualisation

[1] http://freemind.sourceforge.net/wiki/index.php/Main_Page

2.1.3 Typographical conventions

Italic font type is used when referencing object names, class names, methods, attributes, etc. application's programmatic constructs.

Objects methods and objects attributes are presented in C++-like *object::method* and *object::attribute* convention.

3 Technical summary

3.1 Architecture overview

This diagram describes relations between objects used in samples operations. The objects are created during initialization and afterwards they interact with each other to carry out parts of process they are being responsible for.

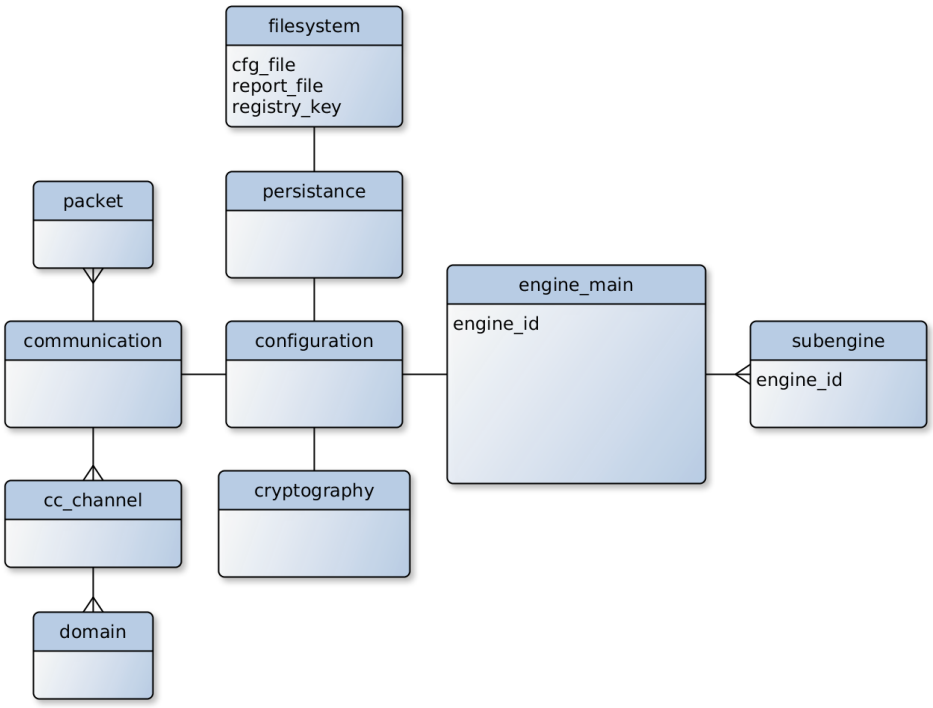


Diagram 2: relations between engine_main and surrounding objects

engine_main is most important part of the samples internal architecture. It interacts with *communication* object via *configuration* with *receive* and *report* methods. *communication* in order

perform C&C communication operations object uses data and methods provided by *cc_channel* object, which describes particular C&C channel. *engine_main* also interacts with victim's operating system registry and filesystem using *persistence* object.

3.2 Objects definitions

The analyzed application consists of several classes (or types) of objects that interact with each other. In order to understand their purpose and the way they interact with each other, their definitions are being provided in this chapter.

BYTE type

This type represents value stored on 8 bits, i.e. between 0x0 and 0xff.

WORD type

This type represents value stored on 2 BYTEs, i.e. word for x86 and amd64 processors.

DWORD type

This type represents value stored on 4 BYTEs, i.e. double word for x86 and amd64 processors.

VTable type

This type represents pointer to virtual table of class. Classes inheriting from other classes or possessing virtual methods contain this pointer. It can be used to identify and distinguish between some classes and draw conclusions about relations between them.

WString

This class is used for storing and processing of UNICODE strings.

Vector<T>, Queue<T>, List<T>

This classes are used to store sets of <T> objects in a structured manner.

filesystem

Filesystem class is responsible for interacting with the victim's operating system filesystem.

persistence

This class is responsible for providing persistence functionality, i.e. storing necessary data in registry and filesystem. *Persistence* object contains *filesystem* object.

cryptography

This class is responsible for providing cryptographic and pseudorandom functionality. One of its most heavily used functions are *packet_2_command*, which converts encrypted C&C packets into decrypted commands and *result_2_packet*, which converts command results into encrypted packets, ready to be reported via C&C.

packet

Packet represents structure containing unprocessed (before final decryption) data received by c*ommunication* class. In the process of decryption it is converted into c*ommand* class (see below) with c*ryptography* object methods.

Offset	Content	Description
0x00	BYTE* text	Packet contents as retrieved via *communication::receive*
0x04	DWORD length	Length of packet's content in BYTEs

command

command represents decrypted data that is organized in a structure ready to be consumed by the respective target engine. Its result can be transformed back into encrypted *packet* structure using *cryptography* object methods.

Offset	Content	Description
0x00	**DWORD** engine_Id	Id of an engine that this command is directed to
0x04	**DWORD** cmd_id	Id of a command
0x08	**BYTE*** content	Command content, interpretation varies among commands
0x0c	**DWORD** length	Command content length, in BYTEs

cc_channel

cc_channel object represents one of possible communication channels. It provides methods for C&C interaction (receiving data and reporting data) that are used by *communication* object.

Offset	Content	Description
0x00	vtable	Table with virtual functions – netapi64.dll +0x2a6a0
0x08	**Vector<WString>** domain	Vector containing WStrings with C&C domains
0x20	**WString** current_domain	Currently used C&C domain

Method name	Functionality
receive_data	Tries to receive data from remote node
report_data	Tries to report data to remote node
generate_uri	Generates particular URL string used for an operation
configure_headers	Configures headers for incoming HTTP request
encode	Encodes data for transport
decode	Decodes data from transport

communication

Communication object provides an interface for interacting with C&C infrastructure. Two main methods, *receive* and *report*, are used by receiving (*thread_receiving*) and reporting (*thread_reporting*) threads, described in subsequent parts of this report.

Offset	Content	Description
0x00	vtable	Table with virtual functions – netapi64.dll +0x2a4c8
0x04	**configuration*** config	Pointer to configuration object
0x08	**Vector<cc_channel_*>** channels	Vector containing pointers to C&C channel information
0x18	**cc_channel*** cur_channel	Pointer to current C&C channel
0x1c	**Queue<packet*>** packets	Queue containing packets received via C&C channel

Method name	Functionality
receive	Receives and enqueues packet from C&C channel
report	Reads report data from report file and sends it over C&C channel

configuration

configuration is a part of *engine_main*, core application's object that supervises operations performed by remaining engines (also called: subengines). It interacts with *communication* object, *cryptography* engine, *persistence* object.

Offset	Content	Description
0x00	**Vtable**	Table with virtual functions – netapi64.dll +0x2a54c
0x04	**communication*** comm	Pointer to communication object
0x08	**cryptography*** crypto	Pointer to cryptographic engine object
0x0c	**persistence*** persistence	Pointer to registry key object
0x20	**Vector<engine_*>** engines	Vector containing pointers to all registered engines
0x30	**Vector<DWORD>** codes	Verb codes
0x40	**BYTE** engines_count	Registered subengine count

Method name	Functionality
read_report_file	Reads content of report file in order to send it via C&C channel

engine_main

The most important object class is *engine_main*. Its responsible for supervising other objects and all engines that perform operations.

It is one of four engines that are being constructed during the sample's execution. It inherits from abstract class *engine_*. Its identification DWORD is 0x3033. Its vtable starts at netapi64.dll+0x2a568.

Offset	Content	Description
0x00	vtable	Table with virtual functions – netapi64.dll +0x2a568
0x04	**DWORD** engine_id	Main engine identification DWORD is 0x3033
0x08	**configuration** my_config	Engine_main contains Configuration object

Method name	Functionality
run_routine	Each engine has this method. It is responsible for running a separate thread which performs operations specific to this engine. The functionality of *engine_main::run_routine* is described in operations overview section.
process_command	Each engine has this method. It is responsible for processing commands directed to it. The list of *engine_main* commands is described in operations overview section.
process_packet	Converts *packet* into *command* and relays it into target engine
fetch_packet	Fetches *packet* from received packets queue
engine_count	Contains current count of registered engines
process_result	Each engine has this method which is responsible for generating output of last processed command. The process of producing *engine_main*'s output is described in operations overview section.
start_engines	This method is responsible for starting (creating threads) for all engines registered in Configuration my_config.

engine_reporter

engine_reporter is an object responsible for collecting data from other components of malicious application, reformatting them and preparing for submission via C&C channel.

It inherits from abstract class *engine_*. Its identification is 0x2103. It's vtable starts at netapi64.dll+0x2a64c.

Offset	Content	Description
0x00	vtable	Table with virtual functions – netapi64.dll +0x2a64c
0x04	DWORD engine_id	Main engine identification DWORD is 0x2103

Method name	Functionality
run_routine	Each engine has this method. It is responsible for running a separate thread which performs operations specific to this engine. The functionality of *engine_reporter::run_routine* is described in operations overview section.
process_command	Each engine has this method. It is responsible for processing commands directed to it. The list of *engine_reporter* commands is described in operations overview section.
produce_result	Each engine has this method which is responsible for generating output of last processed command. The process of producing *engine_reporter*'s output is described in operations overview section.
append_data	Reformats data submitted via mailslot and enlists it for further processing for *engine_main*
report	Reformats data submitted from other engines and enlists it for further processing for *engine_main*

engine_backdoor

engine_backdoor provides backdoor functionality for application. It can create new processes of cmd.exe and pipes that are used to communicate with it. Using *engine_backdoor*, botmaster is able to execute any command that is understood by the system interpreter (cmd.exe).

It is one of four engines that will be constructed. It inherits from abstract class *engine_*. Its identification is 0x2106. It's vtable starts at netapi64.dll+0x2a680.

Offset	Content	Description
0x00	vtable	Table with virtual functions – netapi64.dll +0x2a680
0x04	DWORD engine_id	Main engine identification DWORD is 0x2106

Method name	Functionality
run_routine	Each engine has this method. It is responsible for running a separate thread which performs operations specific to this engine. The functionality of *engine_backdoor::run_routine* is described in operations overview section.
process_command	Each engine has this method. It is responsible for processing commands directed to it. The list of *engine_backdoor* commands is described in operations overview section.
produce_result	Each engine has this method which is responsible for generating output of last processed command. The process of producing *engine_reporter*'s output is described in operations overview section.

3.3 Using architecture in operations

This section describes the most probable role of sample in its malicious environment. It shows interactions between source sample (*sample.exe*), *netapi64.dll* library, C&C channel, report files, external executables (e.g. *cmd.exe*), reporting mailslot and other presumed components that are working in cooperation with main sample (e.g. *sample_2.exe*, *sample_3.exe*, *sample_4.exe*).

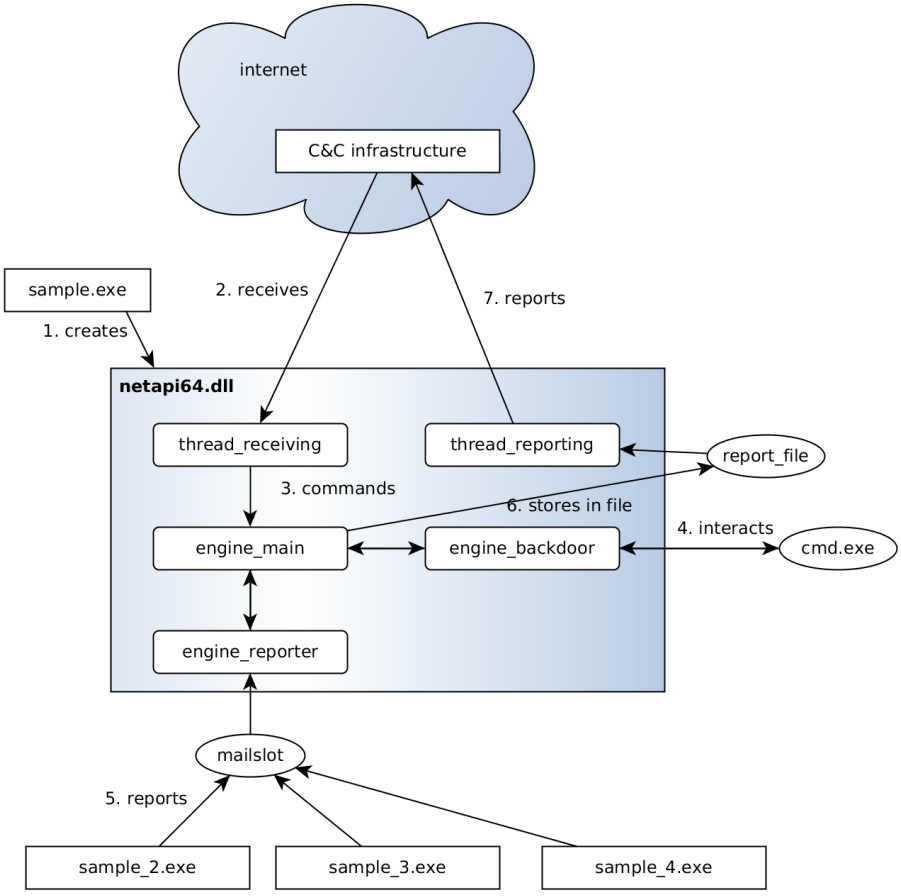


Diagram 3: Malicious architecture and its operations

netapi64.dll component communicates with C&C infrastructure via HTTPS protocol. It receives packets of data (2) that are decrypted and converted into commands and distributed internally (3).

These can be used to configure and adjust internal components as well as interact with external programs with *engine_backdoor* (4).

Another use case for the malicious architecture is other components reporting data to mailslot created by *engine_reporter* (5).

These presumed components may be e.g. info stealers injected into web browsers or specialized software used to interact with particular interfaces (e.g. SCADA systems). Reported data is stored in the report file (6). And is included into the main report and sent to remote C&C infrastructure (7).

3.4 Operations overview

This report does not include operations performed prior to executing sample on victim's machine. In order to investigate such operations, additional evidence needs to be collected for analysis.

Upon it's execution, sample performs direct attempt to drop another loadable module (*netapi64.dll*) by extracting it from its *.data* section and execute it. No additional protectors are employed in this process.

After that, *rundll32.exe* is being used for executing dropped modules code. *Netapi64.dll* begins operation by verifying if the system is already infected and in case it's not - spawning several operating threads.

In spawned threads, particular objects carry out their operations, which include communicating with C&C, processing commands and their results, servicing backdoor, reconfiguration, registering and executing additional modules and relaying submitted data to C&C infrastructure.

3.4.1 Dropping and executing library

After execution start, sample executable is writing part of its *.data* section into file pointed by: *C:\%ALLUSERPROFILE%\netapi64.dll*, which in our analysis environment corresponds to: *C:\ProgramData\netapi64.dll*.

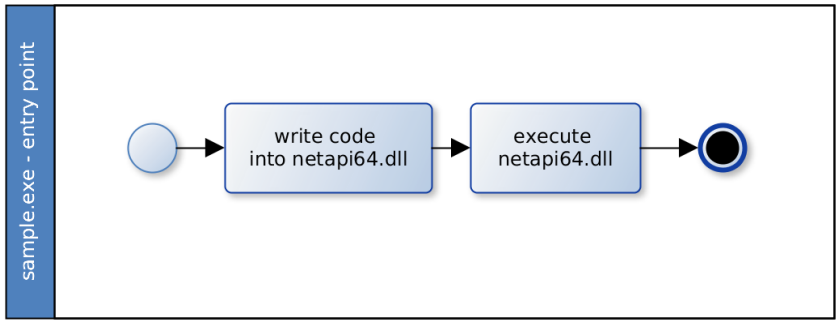


Diagram 4: entry point od analyzed sample

After successfully dropping the library into the filesystem, the main sample attempts to execute it using *ShellExecuteW* library call in conjunction with *rundll32.exe* application. The library entry being called is named: *open*.

3.4.2 Main thread in library

The first thing that the malicious code executed in *netapi64.dll* does is to verify if the installation has been completed on host operating system. It attempts to open mutex with defined name (Appendix A – Indicators of C). If mutex is successfully opened, main thread interprets it as sign of application being already started and exits.

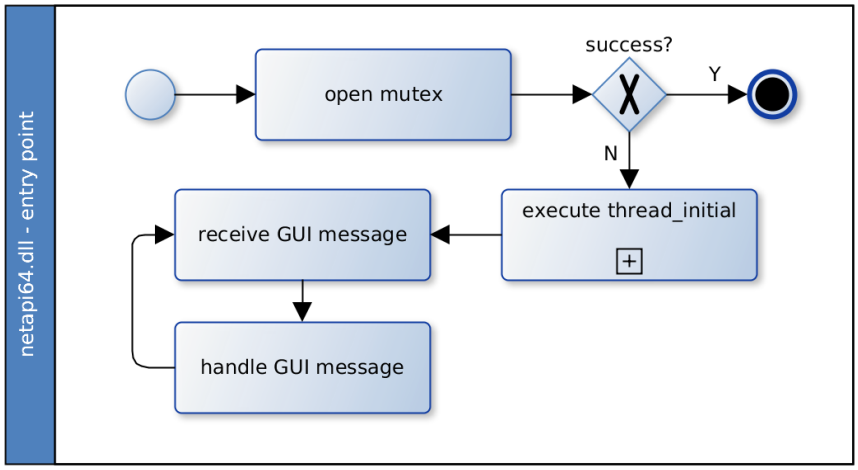


Diagram 5: netapi64.dll entry point

If the mutex could not be opened, main thread creates thread *thread_initial*, which is responsible for bootstrapping and running malicious application.

Afterwards the main thread proceeds to execute the GUI message dispatching loop. This loop is a construction typical to legitimate GUI applications. It is responsible for handling interactions from user, such as moving the window around or clicking over GUI controls. In most cases such applications register their own custom functions for handling GUI messages.

Since no actual window has been presented to user in this case and no custom handling function has been registered, it is assumed that the purpose of this construction is to legitimize malicious code. Some automatic analysis tools might lower malicious score based on such construction being present in application.

3.4.3 Thread thread_initial

Thread *thread_initial* is responsible for constructing main application objects and creating threads for subengines that are servicing application processes.

It starts by creating the major supervisor of all application processes, that is *engine_main* object. It also constructs its necessary attributes, among them: *communication*, an object responsible for handling

C&C communication processes, *cc_channel*, describing current C&C channel configuration and *configuration*, which is responsible for maintaining application configuration.

Afterwards it proceeds to construct subengines that will be employed to service various tasks required by the operations. These are: *engine_reporter*, responsible for collecting partial reports submitted to application's mailslot and *engine_backdoor*, which provides backdoor functionality.

All of the engines are registered within *engine_main* on a special list. It uses it to properly distribute commands received from C&C channel among subengines and collect results of executing these commands.

All of them have their servicing routine called *run_routine* that is used in creation of servicing threads.

All of them have the method *process_command* used for processing commands after particular engine receives them from *engine_main*.

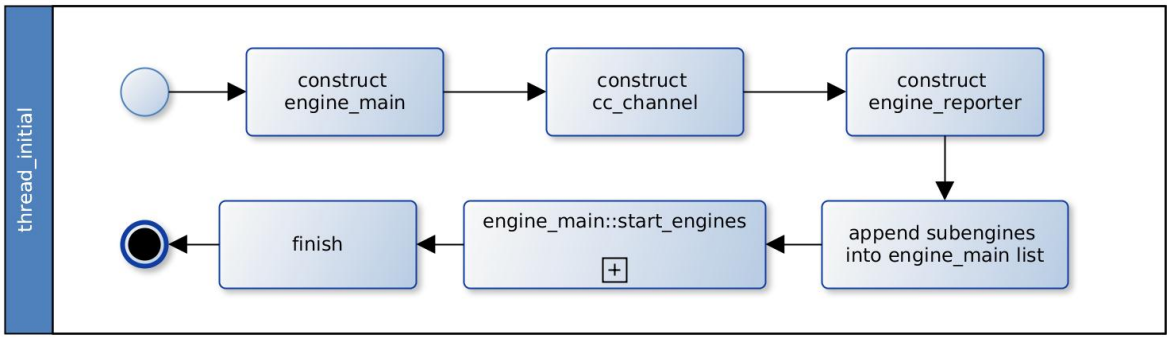


Diagram 6: Initial thread is started

After registering necessary subengines, *thread_initial* calls method *engine_main::start_engines*. This method does not return as long as application is operating. When it finishes, initial thread performs cleaning activities (such as freeing memory) and exits.

3.4.4 Method engine_main::start_engines

The first thing that method *engine_main::start_engines* does is creating a mutex that will signal to other components of malicious architecture that the application is up and running.

This method also starts two threads servicing C&C communication, namely *thread_receiving* and *thread_reporting*, the first responsible for receiving and decoding incoming packets, the latter responsible for sending reports to remote C&C nodes.

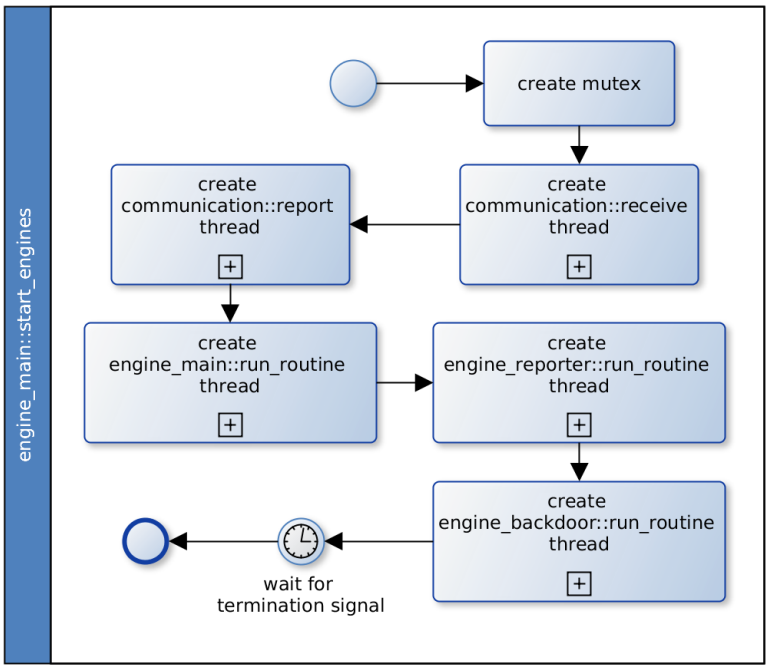


Diagram 7: engine_main is starting engines

Afterwards, it proceeds to create threads for each of the registered engines *run_routine* methods.

After completing thread creation, *engine_main::start_engines* waits for signal that operations are finished. If such signal is received, it returns.

3.4.5 Method communication::receive

The *communication::receive* method runs in a dedicated thread. It is responsible for retrieving data from C&C infrastructure, decrypting it and creating a *packet* structure. *packet* contains two attributes: decoded data and its length. It will be converted into *command* class at later time by *engine_main*.

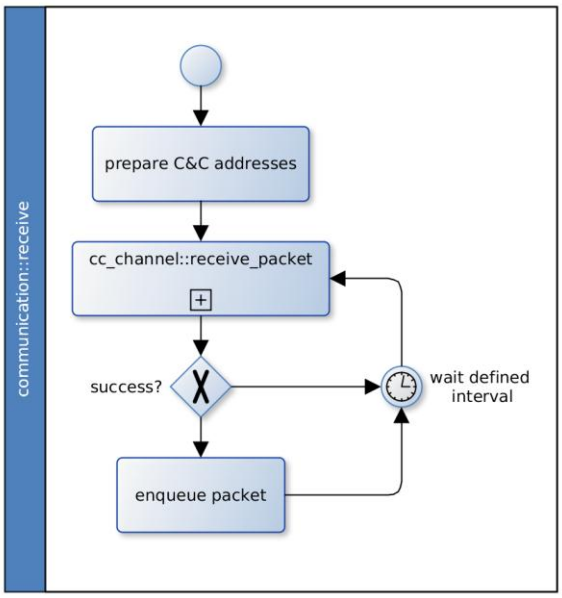


Diagram 8: Receiving loop

3.4.6 Method cc_channel::receive_packet

Receiving routine is relying on WinInet library functions for setting up secure connection with remote node and receiving data. Upon successful retrieval, the data is being verified for consistency and decrypted.

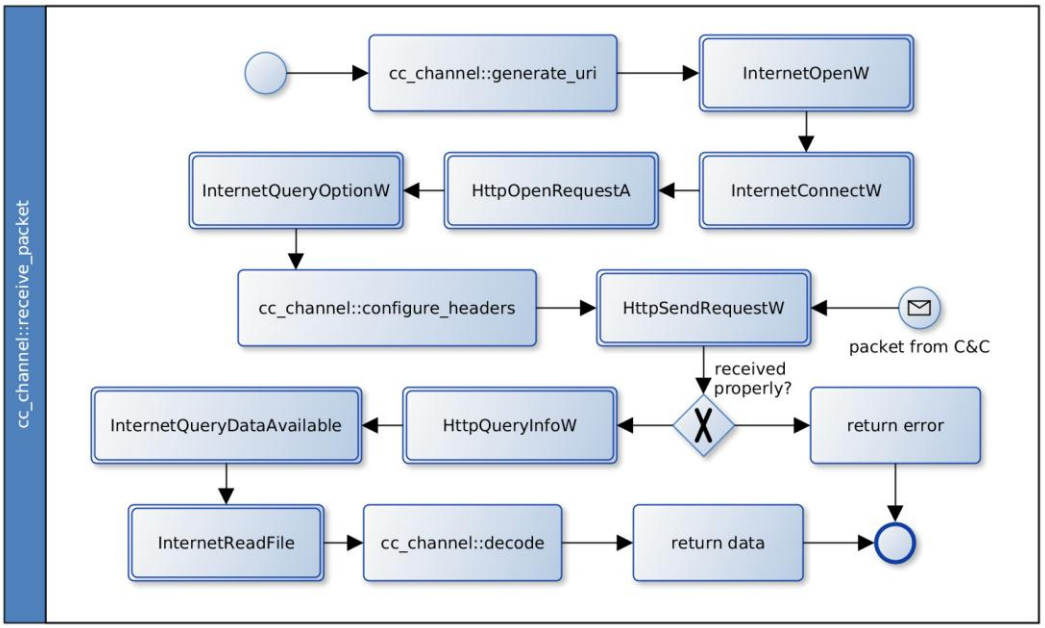


Diagram 9: Receiving procedure

After receiving data and creating a *packet* structure, it's inserted into *engine_main*'s packet queue for further processing.

3.4.7 Method engine_main::run_routine

engine_main's run_routine method is responsible for two important things. The first one being fetching *packets* received via C&C channel from queue (*engine_main::fetch_packet*) and processing them (*engine_main::process_packet*). The second one being collecting results from *Commands* processed by other subengines registered within *engine_main* and storing them in results file. These operations are the core of the whole module based processing of the sample.

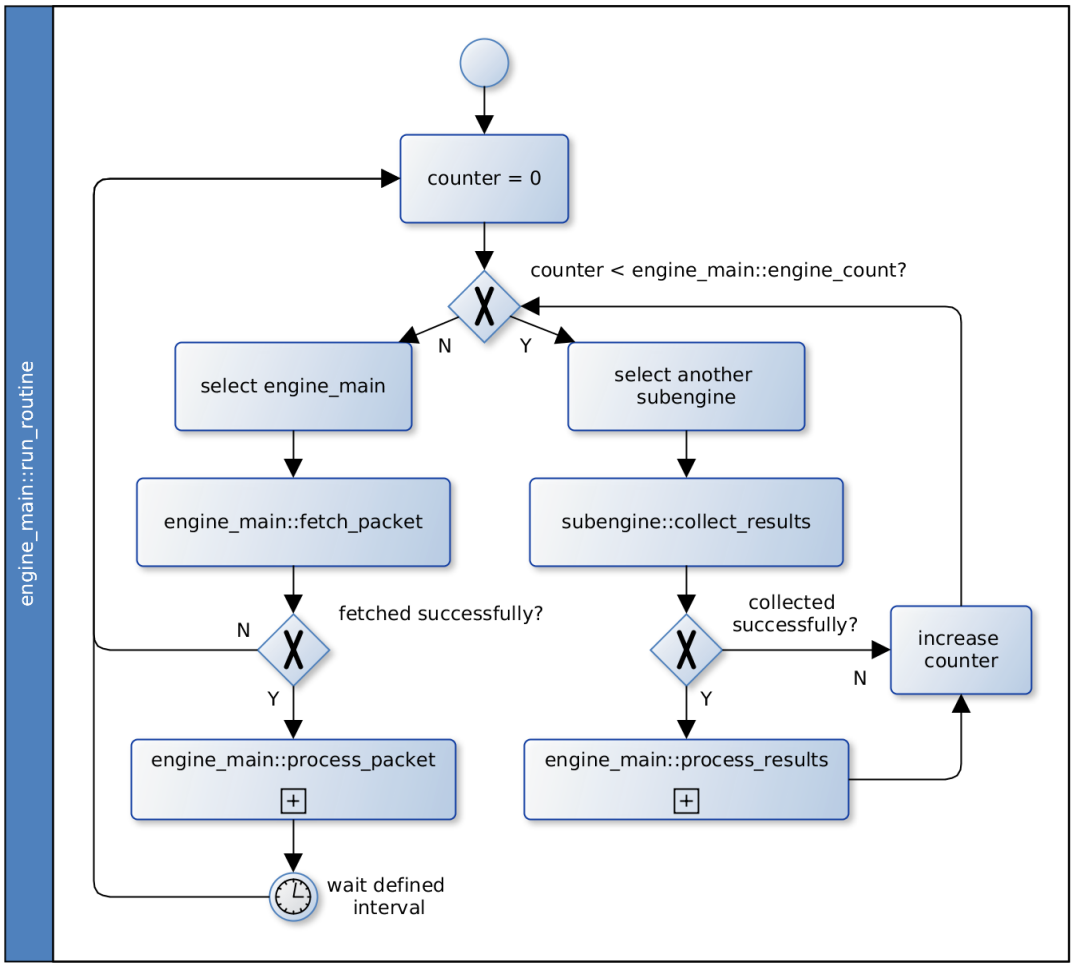


Diagram 10: Main execution loop of *engine_main*

3.4.8 Method engine_main::process_packet

Processing of packet is divided into two stages. In the first stage, *engine_main* uses *cryptography* methods to decrypt *packet* and convert it into *command* object. In second stage, *command*'s target *engine_id* value is being inspected and compared to all engines registered within *engine_main.* After identifying target subengine, *command* is inserted into this subengines *command* list for further processing.

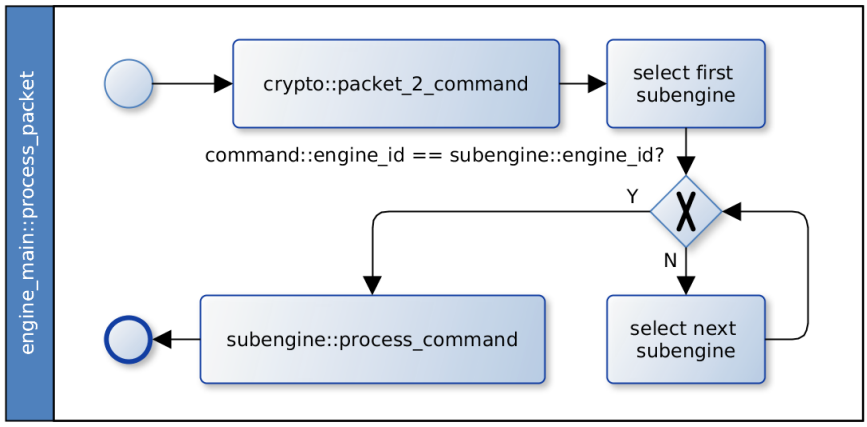


Diagram 11: Packet processing by engine_main

3.4.9 Method engine_main::process_command

The following table describes functions performed by *engine_main* upon receiving specific command.

Command id	Function
CMD_EM_GET_CONFIG	Send current configuration to C&C, including current C&C domains and registered engines ids.
CMD_EM_GET_ENG_IDS	Send registered engines ids to C&C
CMD_EM_SET_CC_INT	Set the interval between attempts to receive packet from C&C
CMD_EM_SAVE_CONFIG	Save configuration to configuration file
CMD_EM_CHANGE_CHANNEL	Change active cc_channel
CMD_EM_REGISTER_ENGINE	Load and register new engine within engine_main
CMD_EM_UNREGISTER_ENGINE	Unregister engine from engine_main
CMD_EM_REGISTER_CC	Register new cc_channel
CMD_EM_UNREGISTER_CC	Unregister selected cc_channel
CMD_EM_RUN_UNINSTALL	Uninstall application

3.4.10 Method engine_main::process_results

engine_main::run_routine periodically inspects all registered subengines for results of processed *commands*. If it encounters output being ready to report in one of subengines, it converts it into *packet* structure (i.e. encrypts and serializes it) and writes it into report file using *persistence* object.

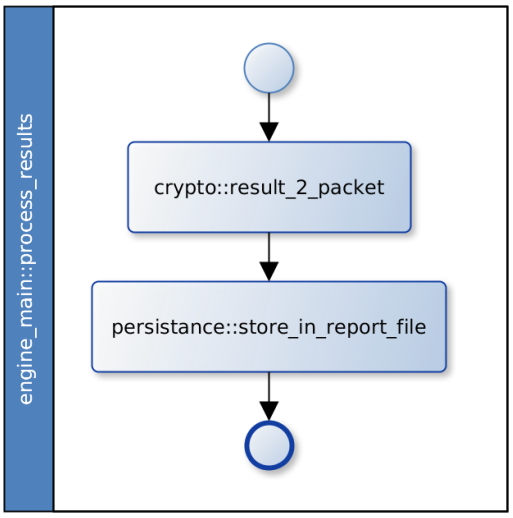


Diagram 12: Processing results by *engine_main*

3.4.11 Method engine_reporter::run_routine

The *engine_reporter::run_routine* is responsible for handling data submitted to applications mailslot by other elements of malicious architecture in the system. It creates a mailslot with defined name (can be found in Appendix A – Indicators of C) and afterwards it performs periodical checks for new data submitted to it. If it encounters new data it compares it to string “SCREEN”. In case it matches, a screenshot is created using *GDIPlus.dll* functions. In other cases data is handled as simple text.

In both cases submitted data is properly reformatted and inserted into queue for collecting by *engine_main::process_results*. After sufficient amount of data is collected, a report file will be submitted to malicious actors via C&C channel via *communication::report* (see:).

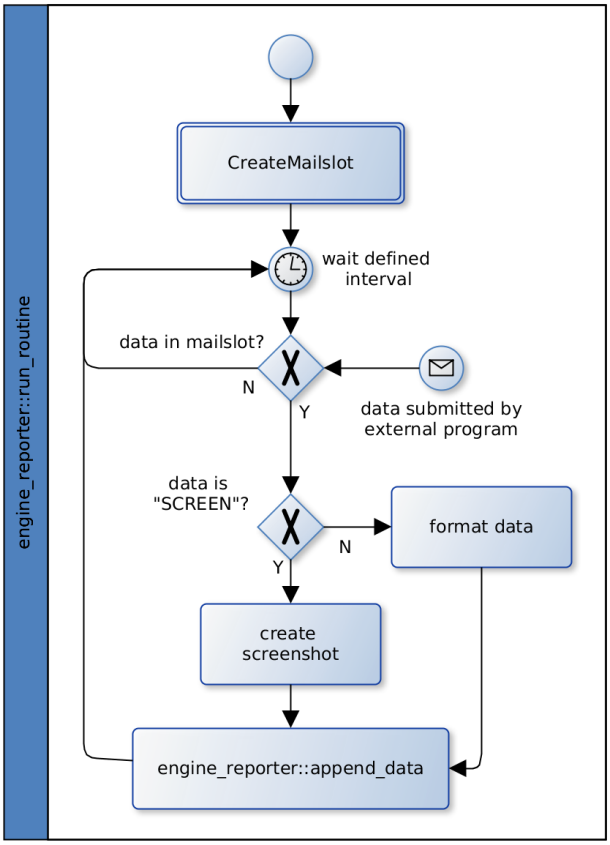


Diagram 13: Main execution loop of *engine_reporter*

Data can also be reported by other subengines to *engine_reporter* internally via *engine_reporter::report* method.

3.4.12 Method engine_backdoor::run_routine

Upon execution, engine_backdoor::run_routine calls *Sleep* function and does not perform any other operations. All functionality of this subengine is performed by *process_command* method.

3.4.13 Method engine_backdoor::process_command

The following table describes functions performed by *engine_backdoor* upon receiving specific command.

Command id	Function
CMD_EB_START(0x0)	Starting external program – system interpreter (cmd.exe) with connected input and output pipes
CMD_EB_STOP(0x1)	Stopping external program – interpreter
CMD_EB_EXECUTE(0x2)	Subsequent interpreter instructions submitted by CMD_EB_EXECUTE will be relayed to interpreter via input pipe and results will be retrieved via output pipe. All results will be reported to *engine_reporter*
CMD_EB_GET_STATE(0x3)	Reports state of *engine_backdoor* – STARTED or STOPED (sic)

3.4.14 Method communication::report

After sufficient amount of data has been collected in report file, *communication::report* thread will attempt to report its content to remote C&C node using *cc_channel::report_data* method.

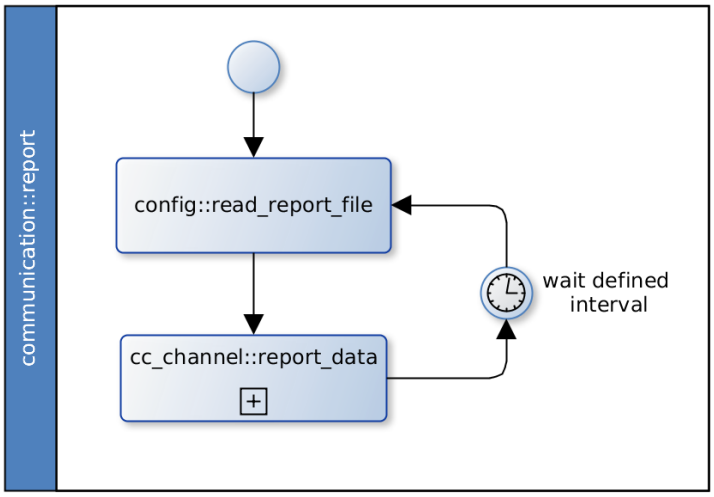


Diagram 14: Reporting loop

3.4.15 Method cc_channel::report_data

Reporting routine is relying on WinInet library functions for setting up secure connection with remote node and sending data. This routine is very similar to receiving routine.

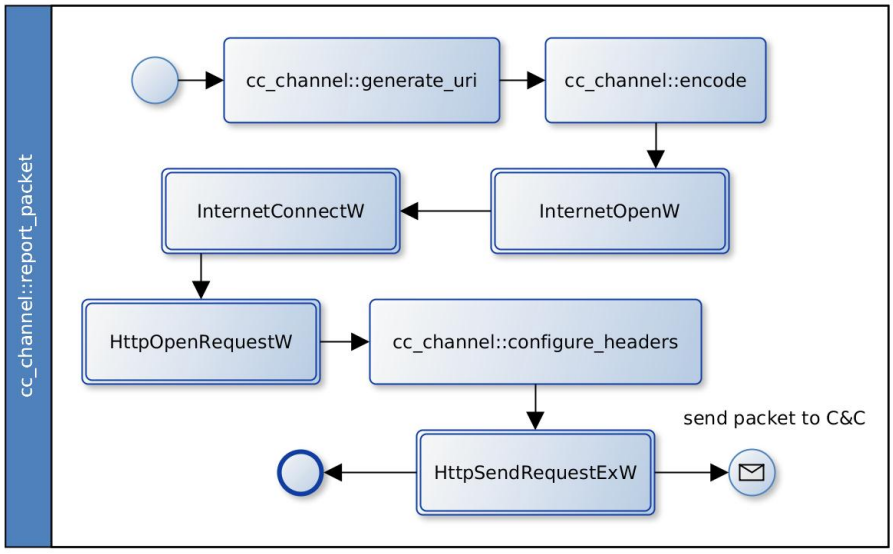


Diagram 15: Reporting procedure

4 Appendix A – Indicators of Compromise

Indicators of Compromise presented below can be used to determine if particular system was compromised or if particular network segment contains compromised hosts. Label **<%random%>** means that part of an object property has been randomly generated.

4.1.1 IoC in filesystem

File	Sample.exe (submitted for analysis)
Name	Unknown
Location	Unknown
Size	275456
MD5	8b6d824619e993f74973eedfaf18be78
SHA1	0f04dad5194f97bb4f1808df19196b04b4aee1b8
SHA256	972e907a901a7716f3b8f9651eadd65a 0ce09bbc78a1ceacff6f52056af8e8f4
SHA512	cb61fc9c58d8ed1cf3a40fa676c1a1d685a09a920beca2b577266ce17bdf9 f7ee14927b5c33d4e23daf27d72f6ac32ed4071570af88f83de39823acfb9 785422

Artifact	Dropped library
Name	netapi64.dll
Location	%ALLUSERPROFILE% (e.g. C:\ProgramData\)
Size	233984
MD5	59c01073d4dd18baa5f0cc00b62e6524
SHA1	a06b8bb4ebeffce36c062d1948ee4ed2042e8852
SHA256	7102b96bce8db14ebb18d6c47261f080 1315730ba24c86237efb4fbfc555ad4c
SHA512	c3188d39ed293a59dbcb7d529060385128f97c88883622c6741527c4d434f 51bb531000dacb4019e16b9c80bf6bf15a2ce79dca60df836ca5a465e84d9 783567

Artifact	Reporting file
Name	hi**<%random%>**.tmp
Location	<%DEFAULTUSERPROFILE%>\AppData\Roaming\
Size	Unknown
MD5	Unknown
SHA1	Unknown
SHA256	Unknown
SHA512	Unknown

Artifact	Configuration file
Name	co**<%random%>**.tmp
Location	<%DEFAULTUSERPROFILE%>\AppData\Roaming\
Size	Unknown
MD5	Unknown
SHA1	Unknown
SHA256	Unknown
SHA512	Unknown

Artifact	Registry key for configuration
Name	{899903DD-E913-412C-ADC8-63A405AF0E77}
Location	<HKEY_USERS>**<%random%>**\Software\Microsoft\MediaPlayer\

4.1.2 IoC in network communication patterns

Artifact	C&C domain
Status	**INACTIVE**
Pattern	**microsoftsupp.com**
Owner on date of analysis	Unknown
URI	https://microsoftsupp.com/*

Fast-flux	No
IP resolved on date of analysis	Irrelevant (blocked)
IP CC	Irrelevant (blocked)

Artifact	**C&C domain**
Status	**INACTIVE**
Pattern	**inteldrv64.com**
Owner on date of analysis	Unknown
URI	https://inteldrv64.com/*
Fast-flux	No
IP resolved on date of analysis	Irrelevant (blocked)
IP CC	Irrelevant (blocked)

4.1.3 Other IoC

Artifact	**Activation mutex**
Name	9bfc46eb-e42b-47ab-9c9f-00de3e533fef

Artifact	**Reporting mailslot for *engine_reporter***
Name	\\.\mailslot\95ca3a2a-6503-49da-9fe4-21bce3e6dc8

5 Appendix B – list of enclosed analysis artifacts

Enclosed in this report are execution visualizations.

1. ev_main_thread.mm – execution visualization of netapi64.dll's main thread
2. ev_thread_initial.mm – execution visualization of netapi64.dll's thread *thread_initial*
3. ev_thread_receiving.mm – execution visualization of netapi64.dll's main *thread_receiving*
4. ev_er_run_routine.mm – execution visualization of netapi64.dll's main *engine_reporter::run_routine*