Malware Instrumentation Application to Regin Analysis

Matthieu Kaczmarek
Email: tecamac@gmail.com

This paper was presented at Botconf 2015, Paris, 2-4 December 2015, www.botconf.eu
It is published in the Journal on Cybercrime & Digital Investigations by CECyF, https://journal.cecyf.fr/ojs

DOI: 10.18464/cybin.v1i1.2

***Abstract*—The complexity of the Regin malware underlines the importance of reverse engineering in modern incident response. The present study shows that such complexity can be overcome: substantial information about adversary tactics, techniques and procedures is obtained from reverse engineering.**

An introduction to the Regin development framework is provided along with instrumentation guidelines. Such instrumentation enables experimentation with malware modules. So analysis can derectly leverage malware's own code without the need to program an analysis toolkit.

As an application of the presented instrumentation, the underlying botnet architecture is analyzed. Finally conclusions from different perspectives are provided: defense, attack and counter intelligence.

I. INTRODUCTION

This study presents malware analysis techniques which leverage instrumentation to overcome static analysis limitations. Those techniques are applied to the Regin malware which is a good example of complex malware that does not exhibit its full potential via simple sandbox executions.

Regin is built on a Service Oriented Architecture (SOA) where modules are plugged according to the operation purpose. Such modules are generally not self-activated, they require a specific context and commands to exhibit their behavior. Facing such malware, analyst usually fallback on static analysis and toolkit development to decode and decrypt adversary data saved with the malware.

The need for manual analysis of the malware modules is illustrated by [1], [2] where each known Regin module is thoroughly analyzed to enumerate the malicious capabilities. The present study propose to go beyond such analysis calling modules routines via reuse of the malware orchestrator. Such technique enable experiment and rapid triage of modules as underlined in [3]. Similarly [1], [2] describe the Virtual File System (VFS) format to enable access to malware data. [2] even provides code excerpts of a toolkit to re-implement such access. We rather propose to directly use Regin VFS module to access such data. The latter approach decreases development time while providing full compatibility.

The main drawback of the present approach is the need of a deep understanding of Regin internals. Indeed [1] and [2] only scratch the surface of Regin development framework focusing on modules. The instrumentation the orchestrator require a deep understanding of the Remote Procedure Call protocol which lies in the internals of the malware. On the other hand, this additional reverse engineer enable to go beyond usual studies providing an understanding of the Regin botnet network structure.

The study is divided into 4 sections. Section II pictures the design of the Regin malware. The understanding of the sound development standards used to create this malware is the entry point to reus the internal malware logic. Section III presents technical details about code instrumentation describing the key malware structures and routines that can be reused for the subsequent analysis. Section IV applies the techniques of the preceeding sections to the malware networking. This section shows how much high level information can be extracted from the technical analysis. Finally, Section V summarizes findings from three perspectives: defense, attack and counter-intelligence.

II. DESIGN

The first challenge in reverse engineering Regin malware is its Service Oriented Architecture (SOA). Such an architecture is composed of modules which talk with one another via Remote Procedure Calls (RPC). Modules communicates either locally inside a single instance or remotely over the global botnet. This architecture enable work distribution over instances making it easier to operate a large network of probes collecting information.

A. Overview

A strong design is implemented to ensure stealthiness, confidentiality, availability, scalability and reliability. The resulting botnet can be safely and securely operated over a large network with only average skills and low human interaction.

Stealthiness: It is difficult to quickly identify core Regin code as malware. Indeed, it rather looks like good quality software developed with strong design and strict coding guidelines. Furthermore implementation details underline stealthiness efforts, for example a configurable delay between cryptography rounds is implemented to avoid CPU spikes. Such a delay also dramatically decrease performance, this fact shows that stealthiness is an important requirement.

Confidentiality: Cryptography is a cornerstone of the implementation. All data are stored encrypted and network communications leverage asymmetric cryptography for authentication and confidentiality. Indeed the RPC protocol enable end-to-end encryption with routing so that traffic

interception on relays does not disclose the content of the traffic. Furthermore, digital signing restricts communication to authenticated Regin node.

Availability: Key network components are always redundantly implemented. Examples include master nodes, 127.0.0.2 and 127.0.0.3, and central reporting nodes, 127.0.0.4 and 127.0.0.5. Network transport is also redundantly implemented supposedly to palliate incompatibility.

Scalability: The network protocol and the associated network structure are designed to support a large number of instances. For example master nodes can be locally mapped so that sub-networks obey to different masters balancing load. Furthermore, the internal networking works over a Virtual Private Network (VPN) overlay providing a 32bit virtual address to each nodes with routing and network address translation capabilities. This is a typical functionality in large network management that are rarely observed in malware but for large botnet infrastructures.

Reliability: Strong design efforts are put in making Regin immune to operator errors. On one hand, extensive automation is implemented with customization according to the hosting environment. On the other hand double checks are required for key components. For example, it is very unlikely that an operator would generate unwanted network traffic to a command and control disclosing botnet assets. To enable communications between two nodes at least three elements need to be configured: a virtual IP, a public key and transport channel. Operators likely use predefined standard configuration which certainly go through quality/security assessment. Operations can be entrusted to differently skilled personnel in a tiered service supporting a scalable model.

B. Architecture

A Regin module is a self-contained unit identified by a 16bit integer, a WORD. Each module implements specific functionalities, such as cryptography; Module 000f[1], or compression; Module 000d. Modules can combine several routines to provide more complex services. For example, Module 0007 implements an encrypted and compressed virtual file systems combining Modules 000f and 000d[2].

The malware embeds a minimal set of required core modules listed in Figure 1. Additional modules can be plug into live a instance according to the infection purpose. Such additional modules are usually stored inside the virtual file system supported by Modules 0007 and 003d.

An infection can also feature kernel components. The kernel side implements a second SOA independent from the user-land side. As a standalone system, it also implements a set of core modules listed in Figure 1. Note that modules with similar functionality have identifiers increased by 63 with respect to their user land counterpart.

Kernel and user lands communicate via Module c3bf which implements shared memory and a notification mechanism

ID	Functionality
0001	RPC Dispatcher
0007	Virtual File System
0009	Networking
000b	Logging
000d	Compression
000f	Cryptography
0011	RPC Dump
0013	Neighborhood
0019	UDP Transport
0033	Inactivity Triggers
003d	Virtual File System
c373	TCP Transport

ID	Functionality
0065	Orchestrator
006b	Virtual File System
0071	Compression
0073	Cryptography
00a1	Virtual File System
c3bf	Bridge Kernel and User
c427	Host Parameters
c42f	Process Watch
c431	Hook Engine

Fig. 1. Core Modules

hooking on ZwDuplicateObject. In a nutshell handlers[3] 01 and 03 respectively writes and reads the shared memory transferring RPC between user and kernel lands. This mechanism might be subject to change in the different flavors of Regin but the bridge module ID shall remain the same: c3bf. This is a benefits of the SOA architecture; as long as the interface between modules is preserved, the underlying implementation can be changed without dramatic compatibility issue.

There might exist other flavors of the SOA. According to the author knowledge, standard nodes always features modules with odd number identifiers. However, modules with even numbers are referenced in the code such as Module 000a which seems to be a central reporting module.

```
mov rcx, [rsp+38h+rpc]
mov r8d, 0Ah                ;; Module ID
mov rax, [rcx+RPC.module]
mov r9b, 5                  ;; Handler ID
mov rdx, [rax+MODULE.regin]
mov rax, [rdx+REGIN.helper]
mov edx, 7F000002h
  ;; Master node 127.0.0.2
call [rax+HELPER.queueASync]
  ;; (void rpc, DWORD node, WORD ModID, BYTE HdlID)
```

Similarly some code stubs feature module identifier translation such as the next one where the identifier 001a is translated into the identifier 003d adding the constant 23. Such compilation patterns are common where macro are defined in the source code to adapt constant according to compilation flags. Typically, this pattern suggest that this code fragment might be compiled either with Module 003d or Module 001a according to compilation flags.

```
mov  rcx, [rsp+78h+rpc] ;; rpc
lea  edi, [rbx+23h]     ;; rbx = 1Ah
mov  r9b, 3             ;; Handler ID
mov  r8d, edi           ;; Module ID
mov  edx, 7F000001h     ;; Local node
call queueRPC
```

A last observation supporting the hypothesis of the existence of several Regin flavor is the access control list provided by Module 0009 Handler 1f. Figure 2 provides an example, it grants access to unsigned foreign RPC[4] according to the source module identifier, the destination handler identifier and the destination module identifier. Typically Module 0009 is

[1]By convention mudule and handler identfier are implicitely written under hexadecimal form.

[2]This is illustrated in Section III-B Figure 5

[3]A handler is a routine of a module. This is further explained in Section III-A

[4]Further details about digital signing and access control are provided in Sections IV-D

allowed to query Module 0009 Handlers 11-15, 24 to initiate encrypted communications via a session key exchange. But modules with even number identifiers seem to be granted greater access.

The previous observations suggest that master nodes are compiled with different module identifiers. So that the master nodes are granted privileged control over regular nodes.

C. Remote Procedure Call

Modules features routines indexed by 8bit integers. Service are delivered querying those routine through a specific sequence of event.

Marshaling: The client initialize a data structure and write the RPC input. The code below is an example of marshaling observed in Regin code. An RPC structure is initialized and a BYTE is marshaled into the input buffer.

```
;; Create a rpc
lea   rdx, [rsp+48h+rpc]
mov   r9, [rax+REGIN.helper]
call  [r9+HELPER.rpcNew]
test  al, al
jz    loc_180018624 ;; jmp if error

;; Marshalling
mov   rcx, [rsp+48h+rpc]
mov   rax, [rcx+RPC.module]
mov   rdx, [rax+MODULE.regin]
mov   rax, [rdx+REGIN.helper]
mov   dl, bpl              ;; BYTE to write
call  [rax+HELPER.in.writeByte]
test  al, al
jz    short loc_180018606 ;; jmp if error
```

Queuing: The client send the RPC structure to the local dispatcher, that is Module 0001, with parameters specifying the destination address module and routine. This messaging can either be synchronous or asynchronous. Synchroneous calls wait for the RPC to be fully processed before continuing the execution. Asynchroneous calls leave in separate threads and the caller continues its execution without waiting for the RPC completion.

This corresponds to the following code pattern which where the previously initialized RPC is queued to Module 0007 Handler 03. This is the virtual file system management module, Handler 03 reads a file system record which ID is provided as argument: the byte that has been marshaled into the RPC structure.

```
;; Queueing
mov   rcx, [rsp+48h+rpc] ;; rpc
lea   r8d, [rbx-1]       ;; module ID rbx = 8
mov   r9b, 3             ;; handler ID
mov   rax, [rcx+RPC.module]
mov   rdx, [rax+MODULE.regin]
mov   rax, [rdx+REGIN.helper]
mov   edx, r12d          ;; Virtual IP
call  [rax+HELPER.rpcQueue]
```

Orchestration: If the destination is local, 127.0.0.1, then the dispatcher simply applies the specified routine to the RPC data structure. If the destination is remote, the RPC structure is transferred to the networking manager, Module 0009, for routing to the destination node. Networking is further addressed in Section IV.

Unmarchaling: The destination module routine reads the input from the RPC structure. For example, the following unmarchaling code reads a byte from the input buffer.

```
mov   rax, [rcx+RPC.module]
lea   rdx, [rsp+78h+id]
mov   rcx, [rax+MODULE.regin]
mov   rax, [rcx+REGIN.helper]
mov   rcx, rdi
call  [rax+HELPER.in.readByte]
```

Processing: The output are processed and output are marshaled back into a dedicated buffer in the RPC structure. Finally the RPC returns back to the client following the same steps in the reverse direction. The code below achieves the indented processing: read and decrypt a virtual file system record. Then the output is marshaled in the output buffer of the RPC structure.

```
;; Processing: read the VFS
mov   rcx, cs:vfs          ;; vfs_structure
lea   r9, [rsp+78h+rcd_size];; size
lea   r8, [rsp+78h+rcd]    ;; dst
lea   rdx, [rsp+78h+rcd_id] ;; ID
call  VFSGetRecord
mov   ebx, eax
test  eax, eax
jnz   loc_18004C76A

;; Processing: decrypt the data
mov   r8d, dword ptr [rsp+78h+record_size] ;; size
mov   rdx, [rsp+78h+record_data]           ;; src
mov   rcx, cs:VFSModule ;; module
lea   rax, [rsp+78h+size]
lea   r9, VFSCryptoKey
          ;; key: 73231F4393E19F2F990C17815CFFB401
mov   [rsp+78h+psize], rax
lea   rax, [rsp+78h+written] ;; dst_written_size
mov   [rsp+78h+buffer], rax  ;; dst
mov   [rsp+78h+key_size], 10h ;; key size
call  CryptoDecryptBuffer
mov   ebx, eax
test  eax, eax
jnz   short loc_18004C75C

;; Marshalling of the returned value
mov   rax, [rdi+RPC.module]
mov   r8d, [rsp+78h+size]
mov   rdx, [rsp+78h+data]
mov   rcx, [rax+MODULE.regin]
mov   rax, [rcx+REGIN.helper]
mov   rcx, rdi
call  [rax+HELPER.out.append]
```

III. INSTRUMENTATION

The previous sections showed that Regin code is strongly structured and designed. Furthermore this malware is very large and features numerous functionalities. The usual response for such malware consists in developing a toolkit to decrypt and decode data storage in order to understand the role of the infected machines. This usually results in substantial development work.

We propose an alternative technique base on the sound structure of this malware. Instead of developing a toolkit to interpret Regin data, this section presents how to reuse Regin code. This approach uses instrumentation loading the core module and leveraging the RPC interface to decode data.

A. RPC Helper

The cornerstone of the instrumentation technique is the RPC helper structure that has been exhibited in the previous

```
#Src, Hdl, Dst,      0004, 32, 0001,      0012, 04, 0013,      0009, 14, 0009,      0008, 69, 0009,      0008, 1d, 0009,
0000, 21, 0001,      0004, 32, 0033,      0012, 05, 0013,      0009, 15, 0009,      0008, 6a, 0009,      0008, 1e, 0009,
0000, 23, 0001,      0004, 32, 0009,      0012, 06, 0013,      0009, 24, 0009,      0008, 6b, 0009,      0008, 79, 0009,
0000, 26, 0001,      000e, a0, 000f,      0012, 08, 0013,      0008, 25, 0009,      0008, 6c, 0009,      0008, 80, 0009,
0000, 2f, 0001,      000e, a1, 000f,      0012, 0b, 0013,      0008, 26, 0009,      0008, 6e, 0009,      0008, 7a, 0009,
0000, 3f, 0001,      000e, a2, 000f,      0012, e9, 0013,      0008, 90, 0009,      0008, 6f, 0009,      0008, 7b, 0009,
0000, 30, 0001,      000e, a3, 000f,      0010, 10, 0011,      0008, 60, 0009,      0008, 71, 0009,      0008, 81, 0009,
0000, 32, 0001,      000e, e9, 000f,      0010, 12, 0011,      0008, 61, 0009,      0008, 72, 0009,      0008, 82, 0009,
0000, 35, 0001,      000a, 04, 000b,      0010, 13, 0011,      0008, 62, 0009,      0008, 73, 0009,      0008, 83, 0009,
0000, 3b, 0001,      000a, 05, 000b,      0010, 21, 0011,      0008, 63, 0009,      0008, 74, 0009,      0008, 84, 0009,
0000, 43, 0001,      000a, 11, 000b,      0010, 30, 0011,      0008, 64, 0009,      0008, 75, 0009,      0008, 85, 0009,
0000, 44, 0001,      000a, 28, 000b,      c372, 04, c373,      0008, 65, 0009,      0008, 76, 0009,      0008, e9, 0009,
0000, 46, 0001,      000a, e9, 000b,      0009, 11, 0009,      0008, 66, 0009,      0008, 77, 0009,      c41e, e9, 0009,
0000, 48, 0001,      0032, 81, 0033,      0009, 12, 0009,      0008, 67, 0009,      0008, 78, 0009,
0000, 4a, 0001,      0032, e9, 0033,      0009, 13, 0009,      0008, 68, 0009,      0008, 1c, 0009,
```

Fig. 2. Module Access Control List

section. Figure 3 presents the methods that are used to manage the RPC processing steps. The RPC helper is an entrypoint into Regin instrumentation. The following examples presents how to leverage this helper to interact with Regin nodes. The following code registers a module with ID 7eca where regin_instance is the instance of the local node.

```
/* Get the helper BASE is the base address */
RPC_HELPER* HELPER = (RPC_HELPER*)(BASE + 0x669F0);

/* Create a module */
void* module7eca;
HELPER->modNew(&module7eca, 0x7eca, regin_instance)
```

In order to make module handlers accessible to the other modules managed by the local dispatcher, each handler need to be registered. For example the following code define the handler *writeMsg* and register it with the ID 23. The handler of Module 7eca can now be queried by all other local modules or by any remote nodes connected to this node via RPC.

```
/* Add a module handler */
HELPER->modAddHdl(module7eca, 0x23, writeMsg);
```

Module handlers typically process data. The RPC model supports input and output via dedicated buffers where typed data can be marshaled. Figure 3 lists the supported data types: BYTE, WORD, DWORD, string, wide character strings and raw buffer. The previously referenced handler routine *writeMsg* is detailed below. It takes a raw buffer from the RPC input and print it.

```
/* Handler payload */
  NTSTATUS
  writeMsg(void* rpc){
    BYTE* msg = NULL;
    /* UnMarshalling */
    HELPER->in.readSizeStringBE(rpc, &msg, NULL);
    /* Processing */
    wprintf(L''\n >%s\n>'', msg);
    return STATUS_SUCCESS;
  }
```

Calling this handler would result in the printing of the message provided as argument. The following code present how to call this handler sending the string *"Hello world"* as input where 0b is the string length, 7eca is the destination module ID and 23 the destination handler ID.

```
/* Marshalling */
HELPER->in.writepSizeStringBE(rpc, Hello    World  , 0xb);
/* Queuing */
 status  = HELPER->queueRPC(rpc, 0x7f000001, 0x7eca, 0x23);
```

```
struct DATA_HELPER{
  BYTE(*writepSizeString)(void *rpc, void *src, size_t size, BYTE endianess);
  BYTE(*writepSizeStringBE)(void *rpc, void *src, size_t size);
  BYTE(*append)(void *rpc, void *src, size_t size);
  BYTE(*writeString)(void *rpc, char *s);
  BYTE(*writeWString)(void *rpc, wchar_t *s);
  BYTE(*writeByte)(void *rpc, BYTE b);
   [...] // Similar writers for WORD, DWORD, QWORD and Date structure
  BYTE(*SeekEoBuffer)(void *rpc, size_t *outSize);
  QWORD field;
  BYTE(*readByte)(void *rpc, BYTE* b);
   [...] // Similar readers for WORD, DWORD, QWORD and Date structure
  BYTE(*readSizeString)(void *rpc, BYTE **buff, DWORD *pSize, int endianess
       );
  BYTE(*readSizeStringBE)(void *rpc, BYTE **buff, DWORD *pSize);
  BYTE(*readString)(void *rpc, char **pBuff, size_t size);
  BYTE(*readWString)(void *rpc, wchar_t **pBuff);
  BYTE(*unReadByte)(void* rpc);
  BYTE(*readSize)(void *rpc, void**p);
};

struct RPC_HELPER{
  [...] // object internals
  NTSTATUS(*modNew)(void** mod, DWORD id, void* regin);
  NTSTATUS(*modFree)(void* rpc);
  NTSTATUS(*modAckIP)(void* rpc);
  NTSTATUS(*modAddHdl)(void* mod, WORD hdlID, void* payload1);
  NTSTATUS(*modApplyHdl)(void *mod, void *header, void *in, void *out);
  NTSTATUS(*rpcNew)(void *mod, void **rpc);
  NTSTATUS(*createAlternateRPC)(void *mod, void **rpc);
  NTSTATUS(*rpcFree)(void* rpc);
  NTSTATUS(*altRPCFree)(void* queue);
   [...] // rpc setters and getters
  NTSTATUS(*rpcQueue)(void* rpc, DWORD IP, WORD modID, BYTE hdlID);
  NTSTATUS(*rpcASyncQueue)(void *rpc, DWORD IP, WORD modID, BYTE
       hdlID);
   [...] // rpc queueing variants
  DATA_HELPER in;
  DATA_HELPER out;
   [...] // setters and getters
};
```

Fig. 3. RPC Helper

The second argument, 7f000001, is the address of the reception node. This is the virtual IP address in Regin virtual network: 7f000001 corresponds to the local loop IP address 127.0.0.0.1 representing the local node. RPC can be sent to remote nodes specifying destination virtual IP addresses. For example, the following code sends an RPC to the remote Regin node with virtual IP address 1.2.3.4.

```
/* Queue an RPC */
DWORD dstIP = \x01020304;
HELPER->in.writeSizeStringBE(rpc, "Hello_World", 12);
```

```
; Trace RPC queuing
bp disp+0x12ba7 ".echo -- RPC QUEUED --; .echo Dst IP:;dd (rbx + 0x14)
      L1; .echo Dst Module:; dw (rbx + 0x18) L1; .echo Dst Handler:; db (rbx +
      0x1a) L1; .echo Input:; dd poi(rbx + 0x48); g"
; Trace RPC return
bp disp+0x12bab ".echo -- RPC RETURNS --; .echo Dst IP:;dd (rbx + 0x14)
      L1; .echo Dst Module:; dw (rbx + 0x18) L1; .echo Dst Handler:; db (rbx +
      0x1a) L1; .echo Output:; dd poi(rbx + 0x68);g"
; Trace asynchronous RPC queuing
bp disp+0x12c6e ".echo -- RPC ASYNC QUEUED --:; .echo Dst IP:;dd (rbx
      + 0x14) L1; .echo Dst Module:; dw (rbx + 0x18) L1; .echo Dst Handler:;
      db (rbx + 0x1a) L1; .echo Input:; dd poi(rbx + 0x48);g"
; Trace asynchronous RPC return
bp disp+0x12c71 ".echo -- RPC ASYNC RETURNS --:; .echo Dst IP:;dd (
      rbx + 0x14) L1; .echo Dst Module:; dw (rbx + 0x18) L1; .echo Dst Handler
      :; db (rbx + 0x1a) L1; .echo Output:; dd poi(rbx + 0x68);g"
```

Fig. 4. *Windbg* Breakpoints for RPC Tracing

```
status = HELPER->queueRPC(rpc, dstIP, 0x7eca, 0xe0);
```

The source virtual IP address can be extracted from the processed RPC. For examplet, the handler *writeMsg* routine can be enhanced to print the source address and the received message.

```
/* Handler payload */
NTSTATUS
writeMsg(void* rpc){
  /* Retrieve RPC context */
  BYTE node[4];
  HELPER->getNodeAndModuleID(rpc, (DWORD*)&node, NULL);
  /* Unmarshalling */
  BYTE* msg = NULL;
  HELPER->in.readSizeStringBE(rpc, &msg, NULL);
  /* Processing */
  wprintf(L''\n%i.%i.%i.%i>%s\n>'', node[3], node[2], node[1], node[0], msg);
  return STATUS_SUCCESS;
}
```

This section underlines the flexibility of the Regin programming framework where a chat program can be coded with a few lines registering a single handler. Networking and routing mechanisms are builtin the Regin framework so that Peer to Peer (P2P) networking can be easily programmed. The following section provide further details about this networking framework.

B. RPC Execution Trace

Such an architecture makes difficult to trace the execution. Since control is transferred via queues of RPC, the understanding of the data flow is necessary to study the control flow. Static reverse engineering is complex and debugging can be tricky because of the inherent multithreading. To overcome this difficulty, it is interesting to trace the queuing of RPC. This can be achieved via the *windbg* breakpoints presented in Figure 4. Those are inserted at the beginning and the ending of the routines `rpcQueue` and `rpcASyncQueue` from the helper structure. The breakpoints prints the destination node virtual IP, the destination module ID and the destination handler ID and the marshaled input of the queued RPC. At the end of the RPC processing the output is respectively printed.

For example, Figure 5 show the RPC trace resulting from the addition of a known host public key. This result in a sequence of three RPC. A call to Module 0007 Handler 02 launches the writing of the public key to the VFS container. This writing is proceeded by the compression, Module 000d Handler 01, and the encryption, Module 000f Handler 01, of the newly added public key.

IV. NETWORKING

A. Design

Regin is designed as a peer-to-peer network software, nodes can function either as clients or as servers to another infected host. However some slight evidences suggest that there is other Regin flavor that may act as super nodes internally referenced by local loop aliases: 127.0.0.2 and 127.0.0.3 are believed to reference master nodes, furthermore, 127.0.0.4 and 127.0.0.5 are believed to reference monitoring nodes.

The Regin network implements a virtual overlay on top of the physical network of infected host. Regin nodes are assigned virtual IP addresses stored in Container 01 of Module 0009. This overlay build a VPN over the infected physical network. Modules 0009 and 0013 manage communication inside the virtual network while relying on transport module such as 0019 (UDP) and c373 (TCP) to exchange data over the underlying physical network.

The structure of the virtual network is stored in Container 03 of Module 0009. This configuration, define a list of records assocting virtual IP of remote nodes to a transport modules. Two records can associate a single IP address to several transport modules for resilience purpose. There is no imposed structure upon the virtual network. This makes Regin botnet easy to build with the ability to define several pivots points that are sometime necessary to exploit segmented victim networks.

There is a second overlay on top of the virtual network defined by trusted links. Similarly to a ssh server, each Regin node features a list of trusted hosts associated to public keys, Module 000f container 01. Signed RPC messages from trusted nodes are directly executed. In this model some nodes only acts as proxy: they receive messages which are routed to another node inside the virtual network.

Figure 6 presents the topology of a Regin infection. The solid arrows represent the virtual network overlay and dashed arrows represent the trust overlay. The network is organized into clusters where some nodes concentrate the traffic with multiple incoming network connections such as XX15f814 and XX15f90b. There are also relays, such as XX15bd99, which interconnect clusters. This is a typical topology of a data collection network with probes distributed over a victim network.

B. Transport

The routing of RPC over the botnet is managed by Module 0009 with the help of Module 0013. Connection channels are implemented by dedicated modules such as Module 0019 responsible for UDP channels and Module c373 responsible for TCP channels. Channels module need to implement a common interface for managing incoming and outgoing connections. As long as such an interface is provided, any kind of transport channel can be supported by Regin such as ICMP, steganography, HTTP cookies...

```
-- RPC QUEUED --
Dst IP:
00000000'00187fe4  7f000001;; Local node
Dst Module:
00000000'00187fe8  000f    ;; Module Crypto
Dst Handler:
00000000'00187fea  53      ;; Add a known host
Input:
00000000'0208e310  32000007 00000088 01000000 01000000
00000000'0208e320  552570fb 50659c12 de78301f 0ead5594
-- RPC QUEUED --
Dst IP:
00000000'0018808c  7f000001;; Local node
Dst Module:
00000000'00188090  0007    ;; Module VFS
Dst Handler:
00000000'00188092  02      ;; Write VFS Container
Input:                     ;; Data to write
00000000'02088060  0001c001 00000700 00008832 00000000
00000000'02088070  00000001 2570fb01 659c1255 78301f50
-- RPC QUEUED --
Dst IP:
00000000'00187e94  7f000001;; Local node
Dst Module:
00000000'00187e98  000d    ;; Module Compression
Dst Handler:
00000000'00187e9a  01      ;; Deflate
Input:                     ;; Data to deflate
00000000'02051490  32000007 00000088 01000000 01000000
00000000'020514a0  552570fb 50659c12 de78301f 0ead5594
-- RPC RETURNS --
Dst IP:
00000000'00187e94  7f000001;; Local node
Dst Module:
00000000'00187e98  000d    ;; Module Compression
Dst Handler:
00000000'00187e9a  01      ;; Deflate
Output:                    ;; Deflated data
00000000'0204f730  000007fc bb008832 fbbf0701 12552570
00000000'0204f740  5065ff9c de78301f adff5594 8028c10e
-- RPC QUEUED --
Dst IP:
00000000'00187e94  7f000001;; Local node
Dst Module:
00000000'00187e98  000f    ;; Module Crypto
Dst Handler:
00000000'00187e9a  01      ;; Symetric encryption (RC5)
Input:                     ;; Data to encrypt
00000000'02051490  00000010 431f2373 2f9fe193 81170c99
00000000'020514a0  01b4ff5c 00000149 45e96f43 c0010000
-- RPC RETURNS --
Dst IP:
00000000'00187e94  7f000001;; Local node
Dst Module:
00000000'00187e98  000f    ;; Module Crypto
Dst Handler:
00000000'00187e9a  01      ;; Symetric encryption
Output:                    ;; Encrypted data
00000000'03890390  0601199b 6984694a cb1a09cb 5c8efc9e
00000000'038903a0  91fd1d2f 1919c50c f771a307 1168bd6b
-- RPC RETURNS --
Dst IP:
00000000'0018808c  7f000001;; Local node
Dst Module:
00000000'00188090  0007    ;; Module VFS
Dst Handler:
00000000'00188092  02      ;; Write to container
Output:                    ;; No output
-- RPC RETURNS --
Dst IP:
00000000'00187fe4  7f000001;; Local node
Dst Module:
00000000'00187fe8  000f    ;; Module Crypto
Dst Handler:
00000000'00187fea  53      ;; Add a known host
Output:                    ;; No output
```

Fig. 5. RPC Trace of a Know Host Registration

Transport channels must feature two phases: initiation and data. Those phases can be supported by distinct transport modules, for example initiation can be achieve over UDP port 53 and transport over TCP port 443. Only the initiation channel needs to be specified on the reception node, the data channel is dynamically defined in the initiation message. Incoming connections are managed by Module 0009 Handler 06. This handler take three parameters as input: a BYTE specifying the action 0 to add a channel, 1 to remove a channel, a WORD specifying the transport module managing the incoming connection and a raw buffer which provides connection details such as the listening network port. For example the following code set an incoming UDP channel on port 53 via Module 0019.

```
/* Configure listening connection on UDP 53 */
/* Marshalling mode 0, Module 0019 (UDP), port 53 */
HELPER->in.writeByte(rpc, 0);
HELPER->in.writeWord(rpc, 0x19);
size_t len = wcslen(data) / 2;
HELPER->in.writeSizeStringBE(rpc, 53 , 3);
/* Queue the RPC to start listening connections */
status = HELPER->queueRPC(rpc, DST_IP, 9, 6);
```

Outgoing connections can be defined on the emitting node via Module 0009 Handler 04. This handler takes more parameters to configure the connection as presented in the following example.

```
/* Configure outgoing connection on UDP 53 */
// mode 0: new connection with default parameters
// mode 1: delete connection
// ...
// mode 7: new connection with custom parameters
HELPER->in.writeByte(rpc, 1);
// Virtual IP address on the \regin virtual network
HELPER->in.writeDWord(rpc, 0x00000002);
// Id of the connection
HELPER->in.writeByte(rpc, 1);
// Initiation connection string 192.168.226.235 port 53
HELPER->in.writepSizeStringBE(rpc,
    \x00\x00192.168.226.235\x00\x35\x00\x01\x02\x01\x01\x01\x01,26);
// Data connection string 192.168.226.235 port 443
HELPER->in.writepSizeStringBE(rpc,
    \x00\x00192.168.226.235\x01\xBB\x00\x01\x02\x01\x01\x01\x01,26);
// delay between initiation retries
HELPER->in.writeByte(rpc, 0x96);
// module for initiation channel 0x0019: UDP
HELPER->in.writeWord(rpc, 0x19); // initiation chan module
// module for data channel 0xc373: TCP
HELPER->in.writeWord(rpc, 0xc373); // data chan module
// connection mode
NTSTATUS status = HELPER->queueRPC(rpc, DST_IP, 0x9, 0x4);
```

After this preliminary configuration, RPC can be remotely sent to the node virtual IP 0.0.0.2 residing on the physical host 192.168.226.235 and listening on UDP port 53.

C. Protocol

The present section describes the network protocol on top of the transport protocol. Communications are established by an initialization message. This message defines the data communication channel specified by a module and its parameters for data transport. The initialization message is watermarked

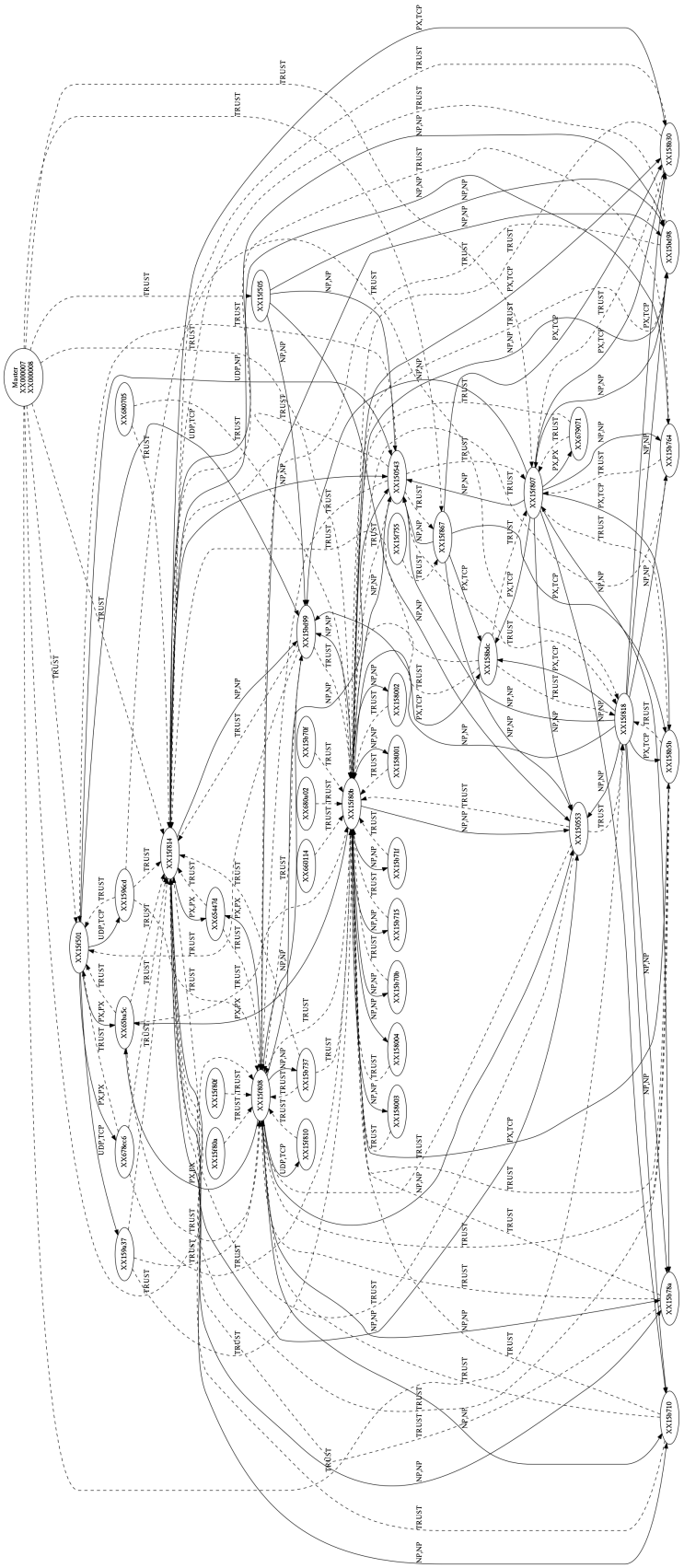

Fig. 6. Virtual Network and Trust Overlays

with the letters "s h i t" respectively located at index 8, 11, 1a and 23. Optionally the initialization message can be RC5 encrypted.

Below we dissect communications between Regin nodes. The first one has 0.0.0.1 as virtual IP and 192.168.226.171 as physical IP. It is configured to communicate via Module c373 (TCP) on port 80 for initialization and port 443 for data. The virtual IP of the destination node is 0.0.0.2 and its physical IP is 192.168.226.235.

The recipient node listens on TCP port 80. The communication starts with an initialization message instructing the recipient node to open a listening thread on TCP port 443 via Module c373. The initialization message typically looks like the packet of Figure 7.

Data is base64 encoded, on decoding we observe marshaled data with an endianess flag, followed by the message size and encrypted data watermarked with the string "s h i t" as presented in the hexadecimal dump in Figure 8

After the watermark removal, the data can be decrypted with the following hardcoded key:

```
71 9b b5 05 c8 69 9b 9f f8 6a 38 92 1f de 02 7e
```

The decrypted message presented in Figure 9 is salted with a random 2-byte word, watermarked with the 16bit integer 7a69 (31337d) and controlled by a CRC32 check-sum. Then follows the connection string that is served to Module 0009 to launch the listening thread. The author does not have a full understanding of all parameters but it features timeout (like 2800) and number of retries (02). The main parameters are the recipient IP address 192.168.226.235 and the port 443.

The receiving node parses the message and starts a thread for data according to the channel defined in the initiation message. Subsequent data communications are similarly encrypted with recipient public keys. If the digital signature is verified against a known trusted public key, then the RPC is queued for processing. On completion the result is sent over the same data channel. The data would typically looks like the dump of Figure 10.

Decryption with the same hardcoded RC5 key yield the data presented in Figure 11 with a routing header and encrypted payload.

The routing header is in clear text so that messages can be relayed over the botnet while enabling end to end encryption. Figure 12 presents the configuration of a relay. The node XX30bf15 has only a very few known host, in particular it does not know the public key of XX9e0112. This nodes act as a bridge between the networks of Victim A and C. As such it only needs the public key of the node from which it was installed, XX020119. So intercepting this node does not compromise the communication between Victim A and C because only routing header are processed by this node.

D. Digital Signature

Regin botnet configurations are secured via asymmetric cryptography. Indeed one cannot connect to a Regin node without proper authentication. A Regin node accept an RPC from a remote node only if this RPC is signed with a known asymmetric key. Authentication and asymmetric cryptography are managed by Module 000f. The following example presents how to list public keys of authorized nodes.

```
/* List known hosts */
HELPER->in.writeDWord(rpc, 1);
NTSTATUS status = HELPER->queueRPC(rpc, DST_IP, 0xf, 0x60);
```

However there is a whitelist access control that enable to bypass the signature validation process. The whitelist is composed of entries specifying a source module, a destination handler and a destination module. A raw version of this whitelist is presented in Figure 2. Figure 13 lists the routines that can be called by a foreign module without signature.

We observe that those functionalities are related to monitoring or debugging. This support the hypothesis of the existence of other Regin flavor with greater control over standard nodes. This whitelist can be obtained via Handler 1f of Module 0009.

```
/* Get the whitelist */
HELPER->queueRPC(rpc, DST_IP, 0x9, 0x1f);
```

V. CONCLUSIONS

Regin is developed with built-in access control and authentication. This can be compared to ssh clients, where security lies in the keys of the users and not in the implementation details. Uncovering the Regin framework does not directly impact the adversary security. Indeed the security of the Regin infrastructure is tied to node secret keys. On the other hand, victims should be in possession of private keys enabling connection to the Regin network. This aspect might be an explanation for a recurrently observed TTP in Regin supported attacks: numerous victims nodes are quickly disinfected by the adversary when the attack is discovered.

Regin is built on a convenient SOA framework aiming at rapid development of remote services. This suggest that Regin operations might be supported by tiered services where a team or a contractor is responsible for providing this framework. Thus several entities may have contracted "Regin kits" from the same provider, making final attribution difficult. Furthermore, such a framework is likely to be delivered as source code or intermediate language. As a result compilation timestamps might be an indicator about the operator rather that the development contractor.

A. Defense Perspective

Regin dose not supports the first steps of an infection, it is a post-exploitation kit likely installed via an initial implant. So Regin infection cannot be efficiently prevented and defense strategy should rather focus on detection and hunting.

On IOCs: The subject malware cannot be easily detected via the usual IOC strategy. For example Section IV-C shows that botnet is organized with pivots where a third victim can bridge two first victims. In this context IOC like IP addresses or domain names are specific to each infection making IOC sharing less efficient. Similarly, [2], [1] underlined the staging mechanism implemented to load the main Regin components.

```
;; Ethernet II, Scr: Vmware_27:40:9e, Dst: Vmware_2f:e5:34
;; Internet Protocol V4, Src: 192.168.226.171, Dst: 192.168.226.235
;; Transmission Control Protocol, Src Port: 49209, Dst Port: http (80),
;; Seq: 1, Ack: 1, Len: 127
  0000  00 50 56 2f e5 34 00 50 56 27 40 9e 08 00 45 00   .PV/.4.PV'@...E.
  0010  00 a7 0c 9a 40 00 80 06 a6 ce c0 a8 e2 ab c0 a8   ....@...........
  0020  e2 eb c0 39 00 50 52 24 4c 48 61 b8 15 8b 50 18   ...9.PR.LHa...P.
  0030  40 29 cc e9 00 00                                @)....
;; Data
  0036                    41 56 38 41 41 41 42 4f 79 4c         AV8AAABOyL
  0040  35 7a 61 4b 62 6a 4e 4d 48 69 42 51 68 6f 61 76   5zaKbjNMHiBQhoav
  [...]
  00a0  43 78 76 5a 51 72 33 4f 59 73 6c 5f 64 36 70 43   CxvZQr3OYsl_d6pC
  00b0  75 57 63 46 59                                    uWcFY
```

Fig. 7. Initialization Message Encapsulation

```
  0000  01                                                .
;; Length
  0001     5f 00 00 00                                     _...
;; Encrypted Data with "shit" watermark
  0005                 4e c8 be  73 68 a6 e3 34 c1 e2 05       N..sh..4...
                                    ^^                          s
  0010  08 68 6a f2 cc ac ff 78  24 23 69 0b e3 fc 39 5d  .hj....x.#i...9]
          ^^                       ^^                      h        i
  0020  60 b0 f4 74 6d 46 e7 f0  fc ed c3 20 16 d7 e7 80  `..tmF..... ....
                ^^                                          t
  0030  fa 42 6e aa 67 f6 62 1c  76 47 8c 73 16 31 07 27  .Bn.g.b.vG.s.1.'
  0040  fe 11 a9 fc ca d6 82 c6  50 48 c2 c5 ae 0e 8c 30  ........PH.....0
  0050  b1 bd 94 2b dc e6 2c 97  f7 7a a4 2b 96 70 56     ...+..,..z.+.pV
```

Fig. 8. Initialization Message Watermark

```
;; Random word
  0000  5b e0                                             [.
;; Watermark 31337 in decimal
  0002        7a 69                                         zi
;; CRC32 checksum
  0004              38 61 f4 42                               8a.B
;; Connection parameters (delay, timeout...)
  0008                           08 02 00 00 00 01 00 00         ........
  0010  00 01 00 00 28 00 00 46  00 00 00 2c 01 00 00 28  ....(..F...,...(
  0020  96                                                .
;; Transport Module c373 (TCP)
  0021  73 c3                                              s.
;; Connection string parameters
  0023           01 24 00 00 00  00 00
;; Destination 192.168.226.235 on port 443 (0x1bb)
  002a                                 31 39 32 2e 31 36  .s........192.16
  0030  38 2e 32 32 36 2e 32 33  35 00 01 bb              8.226.235...
;; Connection string parameters
  003c                                       00 00 00 00              ....
  0040  01 00 00 00 02 01 00 00  00 01 01 01 00 00        ..............
```

Fig. 9. Decrypted Initialization Message

```
;; Ethernet II, Scr: Vmware_27:40:9e, Dst: Vmware_2f:e5:34
;; Internet Protocol V4, Src: 192.168.226.171, Dst: 192.168.226.235
;; Transmission Control Protocol, Src Port: 49210, Dst Port: https (443),
;; Seq: 1, Ack: 1, Len: 397
  0000  00 50 56 2f e5 34 00 50 56 27 40 9e 08 00 45 00   .PV/.4.PV'@...E.
  0010  01 b5 0c a3 40 00 80 06 a5 b7 c0 a8 e2 ab c0 a8   ....@...........
  0020  e2 eb c0 3a 01 bb 56 d5 be a1 cf cf 8a e1 50 18   ...:..V.......P.
  0030  40 29 4d a3 00 00                                @)M...
;; Encrypted
  0036                    72 90 fc 0d 72 90 fd 84 73 b9         r...r...s.
  [...]
  01b0  65 33 18 a7 ae f6 35 f7 32 08 d0 a2 b4 81 7e e3   e3....5.2.....~.
  01c0  6e 92 7e                                          n.~
```

Fig. 10. Data Message

```
;; Destination virtual IP (0.0.0.2), destination module (0009), destination
;; handler (11)
  0000  02 00 00 00 09 00 11                                .......
;; Source handler (13), source IP (0.0.0.1) and source module (0009)
  0007                       13  01 00 00 00 09 00               .......
  [...]
;; Timestamp Fri Nov 28 06:05:56.044 2014
  0020  a2 75 80 5f d1 0a d0 01                             .u._....
  [...]
;; Checksum
  0038                           b2 79 65 fd                     .ye.
;; Signing virtual IP (0.0.0.1)
  003c                                       01 00 00 00             ....
  [...]
;; Message length (0x119) signature version (0x1) and signature length (0x113)
  0050  19 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
  0060  01 00 13 01 00 00                                   ......
;; Encrypted data
  0066                     91 e0  16 77 d6 7b 60 af f4 27        ...w.{`..'
  [...]
  0170  65 c2 cf 98 ec c3 33 ff  cb                         e.....3..
```

Fig. 11. Decrypted Data Message

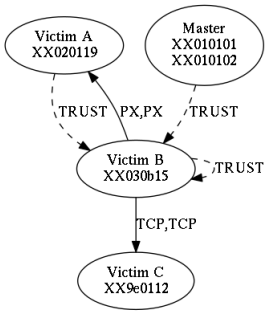

Fig. 12. Pivot Between Victims

Module ID	Whitelisted routines
0000	Manage running modules: start, stop, list
0004	List modules, neighbors or connections
000e	Edit crytographic parameters
000a	Clean logs
0032	Configure inactivity triggers
0012	Neighborhood management
0010	RPC dump management
c372	Open TCP listener
0009	Authenticate, handshake
0008	Manage timeout, manage queued RPC
c41e	Unknown

Fig. 13. RPC Whitelist

The first two stages are disposable and may regularly change so that IOC on filenames and registry are quickly obsolete.

Detection strategy should rather focus on design structures. Indeed, Regin has a strong design with specific protocol and data formats. Such a detection strategy is more sustainable than IOCs because changes in protocol or data structures would likely cause backward compatibility issue on the adversary side.

Structural Detection: An example of such a detection strategy is presented in [4] where the proposed heuristics target the VFS file structure. However, the detection domain should be extended from files to memory and to lower storage level such as inter-partition spaces on disks.

Another structural detection can target network protocol. As presented in Section IV-C the watermark "s h i t" in the initialization message is a low hanging fruit for network detection, [5] propose such an IDS rule. However, this watermark is not used on the data channel and it can be removed from the initialization packet without much impact on the the communication protocol.

On the other hand, Section IV-C underlined that the protocol implement clear text routing header enabling end-to-end encryption. A change in this header would impose to upgrade all routing Regin nodes which may be difficult to manage for the adversary. This routing header is encrypted with a hardcoded key, so an IDS targeting this structure should implement RC5 decryption over the first bytes of the packet. This is feasible even on high traffic but dedicated computational power may be necessary.

Hunting: Figure 6 shows that infections may be very large. Such network maps are valuable to track all infected nodes. This network diagram is obtained compiling the network connections in Container 01 of Module 0009 and the public key list in Container 01 of Module 000f.

Backtracking the network structure with timeline correlation would provide interesting information about how the botnet was deployed so that potential entry points and implant are identified and removed too. Indeed, Regin is a post-exploitation malware installed from an initial implant, so containment must target such implants too.

B. Attacker Perspective

Backdoor: Regin is a very mature malware however it features several weaknesses. First the digital signature verification bypass presented in Section IV-D looks like a backdoor for master nodes. The security of this bypass relies on the module identifier which is a very weak control. Indeed even if standard nodes are distributed with odd number identifier

only, module identifiers can be impersonated providing wider control to a counter-attacker. It is understandable that wider control is necessary for specific nodes. This should rather be implemented via specific public key distribution similarly to the master nodes public keys. Ideally role base access control should be considered on top of the authentication mechanism.

Watermark: The watermark "s h i t" in the initialization message is superfluous and it is a low hanging fruit for IDS detection. It is superfluous because a second verification is achieved on the header after RC5 decryption.

C. Counter-Intelligence Perspective

Observation: The first step toward the understanding of a Regin infection is to identify the scale of the infection. Section V-A presented how to build a network map of the botnet. Additionally, network flow can be identified activating logging mechanisms built-in the malware. Logs are controlled by Module 000b and Module 0009 handlers aa-ac, b4-bd. RPC dump are controlled by Module 0011. Logs and dump are stored in specific configurable VFS. Such an observation mechanism has the benefits of stealthiness as the adversary own tool are leveraged in the process.

Adversary Intent: In order to anticipate new infections it is important to identify the mobile of the adversary so that the related assets are specifically monitored. For this purpose all the containers in the virtual file system need to be extracted and analyzed. For example, Microsoft Exchange e-mail collection filters are stored compressed in Container 02 of Module d9d6. The following code allows to access the content of this container.

```
  /* Create the RPC structure */
HELPER->createStream(module0001->instance, &stream);

  /* Specify the VFS ID: 1 here */
HELPER->in.writeByte(stream, 1);

  /* Specify container ID ModuleID and ContaineID:
     respectively d6d9 and 02*/
HELPER->in.writeSizeStringBE(stream, "\xd6\xd9\x02",3);

  /* Read from VFS */
HELPER->queueStream(stream, DST_IP, 7, 0xc);

  /* Retrieve compressed output */
BYTE* buff;
size_t size = 0;
HELPER->getOut(stream, (void**)&buff, &size);

  /* Inflate , the end is sigaled by the status code */
DWORD sizeout = 0x100;
do{
  HELPER->seekBuffer1(stream, 0);
  HELPER->in.writeDWord(stream, sizeout);
  HELPER->in.writepSizeStringBE(stream, buff, size);
  status = HELPER->queueStream(stream, DST_IP, 0xd, 0x6);
  sizeout *= 2;
}while ( status == 0x1011 && sizeout != 0);
```

Those filters includes mail addresses and keywords. Mail addresses can be directly used to get insight on the infection objectives. Filters also includes a blacklist of keywords used to filter out mails. Such information are valuable in a counter-intelligence strategy. Figure 14 presents some blacklisted keywords, the purpose is obviously SPAM filtering. Such filtering is a double edged sword as explained in [6].

ACKNOWLEDGMENT

I'd like to thank Alexandre, Andrzej, Bruce, Christophe, Damien, Michal, Sergiusz and Yann for their awesome work on that malware.

REFERENCES

[1] Symantec Security Response, "Regin: Top-tier espionage tool enables stealthy surveillance," https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/regin-analysis.pdf, 2014.
[2] Kaspersky Lab Report, "The regin platform nation-state ownage of gsm networks," https://securelist.com/files/2014/11/Kaspersky_Lab_whitepaper_Regin_platform_eng.pdf, 2014.
[3] Ömer Coskun, "Why nation-state malwares target telco networks," https://www.slideshare.net/merCokun1/defcon23-why-nationstatemalwaretargettelcoomercoskun-51440112, 2015.
[4] Paul Rascagnères and Eddy Willems, "Regin, an old but sophisticated cyber espionage toolkit platform," https://blog.gdatasoftware.com/blog/article/regin-an-old-but-sophisticated-cyber-espionage-toolkit-platform.html, 2014.
[5] EmergingThreats, "Regin rules (requries apr module) and flash detection updates," https://github.com/EmergingThreats/et-luajit-scripts/blob/master/luajit.rules.
[6] Paul Ducklin, "Do terrorists use spam to shroud their secrets?" https://nakedsecurity.sophos.com/2015/01/19/do-terrorists-use-spam-to-shroud-their-secrets, 2014.

```
remember me
ride you
asian
autocad
banged
bed
bedroom
being larter
blowjob
breast
camel toe
cock
courtship
cum
delivery failure
delivery notification
delivery status notification
designer
dialost
discount
dreams
drugs
ecard
ejaculation
exposed herself
facsimile
flaccid
for health
gay
girth
greeting
hilarious
horny
hot babes
hot rod
huge
impotence
inches
invincible
jessica alba
longer
macho
make her
manhood
manliness
med
mightier
naked
orgasm
party
pleasure
porn
prada
pussy
rolex
satisfaction
sex
she will
slut
spam
supplement
teat
the person
tight
timepieces
undeliverable
viagra
vyagra
watche
xmas
weight
chicks
dirty
pharmacy
hot deals
walmart
Google Alert
Press Review
Sources Say
Russian Headlines
Wires at
Delivery Status Notification
Radio News
```

Fig. 14. E-Mail Filters: Blacklisted Keywords