Insights from the Analysis of the Mariposa Botnet

Prosenjit Sinha, Amine Boukhtouta, Victor Heber Belarde, Mourad Debbabi
NCFTA Canada & the Computer Security Laboratory
Concordia University
Montreal, Canada

***Abstract*—Nowadays, botnets are among the topmost network threats by combining innovative hacking capabilities. This is due to the fact that they are constantly improved by hackers to become more resilient against detection and debugging techniques. In this respect, we analyze one of the most prominent botnets, namely Mariposa, which infected more than 13 million computers that are located in more than 190 countries. In this regard, we analyze the botnet architecture, components, commands and communication. In this setting, we detail the obfuscation and anti-debugging techniques it uses. Moreover, we detail the infection and code-injection techniques into legitimate processes. In addition, we explain the spreading mechanisms that are employed in Mariposa as well as the underlying communication protocols. More importantly, we analyze the injected bot code. This is accomplished by a reverse engineering exercise that uses both a network analysis together with reverse-engineering analysis. The insights from this work are meant to illustrate the know-how used in current botnet technologies and enable the elaboration of analysis, detection and prevention techniques.**

Keywords: Botnet, Mariposa, Malware, Reverse-Engineering, Obfuscation, Code Injection, Spreading, Communication Protocols, Encryption and Decryption.

I. Introduction

A botnet is a term that designates a network of autonomous software robots (bots) compromising computers that are controlled by a botmaster running a command-and-control center. Nowadays, botnets constitute the delivery platform of choice for the execution of a wide variety of cybercrime attacks targeting identity theft, denial of service, spamming, etc. Although the existence of botnets has been a known fact for a long time, it is the recent growth of cyber crimes and cyber warefare, mediated by botnets, which has attracted the attention of IT security researchers. Accordingly, a surge of interest has been expressed in understanding, analyzing, detecting, eradicating and preventing botnet attacks. In this context, the fight between hackers/cyber criminals on one side and IT security experts on the other side, takes the allure of a non-terminating cat and mouse fight. In order to counter the escalation of hackers ideas and innovations, security experts absolutely have to understand the threat and the employed technologies and then design and implement techniques to mitigate the risk underlying this threat.

One of the creative thoughts compiled by hackers, is the integration of spreading components within botnets. These components use existing technologies to widely distribute malicious code and create an exponentiation of infection within remote hosts. The spreading components are generally threefold: P2P technologies, storage devices (USB keys, SD card, etc.) and instant messaging tools (eg. MSN messenger). The reasons underlying the use of these spreading mechanisms are as follow:

- New generation botnets tend to employ automated propagation of bots that result into wider infected networks.
- P2P technologies as well as instant messaging tools are good candidates to spread malicious bots in the Internet. In [13], the authors claim that organizations can be targeted by bots that make use of spreaders. They stated that among 363 sampled organizations, 85% of them have at least one susceptible bot spreader (MSN messenger, bittorent, emule, etc).
- Storage devices can also be targeted by bots for the purpose of propagation. For instance, USB keys are widely used by people to save their digital contents. Their usage can vary from professional to personal, which implies the occurrence of infections within both corporate and home networks.

Mariposa is a new generation botnet, which embeds spreading mechanisms. It was claimed that 13 million machines got infected around 190 countries in the world by this botnet once it appeared in May 2009 [9]. In addition to the spreading capabilities, Mariposa bots are changed frequently to evade antivirus detection. Mariposa is able to download and execute malicious code on the fly, which makes the botnet extremely harmful. Moreover, it can be associated with other botnets since it has the capability to infect machines with other bots. Furthermore, Mariposa botnet communication uses its own protocol (Iserdo Transport Protocol), which is based on UDP.

It is the level of spreading together with the technical sophistication that brought us to the analysis of the Mariposa botnet. In this paper, we extend the work provided in [9]. We present an in-depth analysis of Mariposa bot by conducting a reverse engineering exercise. The primary intent of our work on Mariposa is to: (1) uncover and remove its obfuscation techniques, (2) circumvent its anti-debugging techniques, (3) understand how the infection and code injection are performed, (4) reveal the functionality of the bot, (5) understand and divulge the inner working of the communication protocol used by the bot and the botmaster. It is our firm belief that this type of work will improve our understanding of botnet technologies and pave the way for a better detection, eradication and prevention techniques.

The remainder of this paper is organized as follows: In section II, we present an overview of the Mariposa botnet. Section III gives an account of Mariposa botnet network behavior. In section IV, we report on the results of the analysis that was done on the bot client. Section V puts forward a description of the different modules that constitute the Mariposa bot. In section VI, we discuss the related work. In Section VII, we give some concluding remarks on this work together with a discussion of future work.

II. Overview of the Mariposa Botnet

In this section, we provide an overview of the Mariposa Botnet. We describe how the botnet works as well as the various features of the bot. Different Variants that constitute Mariposa botnet mainly evolved from the so-called butterfly bot [17]. The authors of Mariposa variants enhance the capabilities of the butterfly bot to make it more robust, resilient, stealthy and threatening. The botnet architecture consists of a set of clients, a server and a master. The architecture is connectionless because it is based on the UDP protocol [18] (no guarantee to the upper layer protocols of message delivery). The server plays the role of the relay between the master and the clients. The UDP protocol is used due to its covertness: The UDP connections are not generally logged in firewalls and gateways, which is not the case with TCP connections. In order to check the presence of bot clients, the server pings clients periodically in a predefined time gap. If it does not receive any reply from the bot, the server marks it as a time-out bot. Further details about the communication protocol are described in the next section that reports on the network analysis of the botnet. We summarize Mariposa's features as follows:

- *Bot client*: The bot has innovative capabilities comparing to the majority of bots that exist in the wild. It has the ability to make direct code injection into remote processes. This injected code corresponds to the entry point of all activities that are done by the bot. Mariposa is capable to download any extra modules like the Zeus botnet and execute them on the fly. Besides, it is capable of performing UDP and TCP flooding, and can tune the flood strength by acting on the data and packet size, and send random data to the victim host. In addition, the bot has mechanisms to spread through the infection of USB keys or using MSN messenger and P2P applications. Moreover, the Mariposa bot contains a module that tracks the visited websites and a grabber that catches all the posted data that are sent from Internet Explorer 6, 7, and 8. On the other hand, the bot is endowed with two downloaders: The first one can download via HTTP, HTTPS and FTP protocols whereas the second downloads files via the ButterFly Network Protocol. Additionally, it has a built-in cookie stuffer for IE and Mozilla Firefox. Recently, Mariposa authors added new features like a slowloris flooder and a reverse proxy module, which can turn all bots into proxy servers.
- *Server*: The server is a mediator between the master and the bot clients. As such, it allows to control the traffic between them by setting the number of frames per second to diminish the CPU usage and the communication latency ratio. We can also set up the maximum upload on the server. The latter localizes the bots using GEOIP localization.
- *Master*: The master represents the core of all operations. It can get multiple server connections and has the ability to enable and disable servers and clients. The master sends commands to bot clients through servers. These commands are various and can be used to customize the operations that are done by clients.

The next section reports on the results of our network analysis.

III. Behavioral Network Analysis

Before digging into the inner details of the analysis of the bot code, we analyze the network behaviors of Mariposa in a controlled environment to grasp the botnet behaviors. First, let us explain the experimental setup for the network analysis. The controlled environment is based on VMware Server 2.0.3 [20] running on a Windows XP system. This software allows running multiple virtual machines in an isolated environment and gives a certain flexibility to create different types of network architectures. The network consists of a default virtual network, which behaves as a stub network. In our analysis, we use four hosts to build a virtual network. These hosts are used to set up the botnet. We installed a master, a C&C server and a host, which is infected by a Mariposa bot. The fourth host is used as sniffing box. It runs a live-CD for network security analysts [16]. The utility of this live-CD resides in logging all communications promiscuously in order to correlate events and monitor the network activities of the botnet. It also allows to verify whether backdoors are set or not. In addition, it can bind to any Linux DNS server. As a result, network records can be created to simulate an Internet-connected network. For this intent, we used $c:\backslash windows\backslash system32\backslash drivers\backslash etc\backslash hosts$ file as a source of a domain name resolution. The communications within the botnet breaks into three phases: initialization phase, bot liveness phase and action phase. All the phases involve the participation of the master, server and the bot client.

The initialization phase takes place after an infection. Once a bot infects a machine, it sends a *join server command*. This command allows a bot to register the IP address of the bot within the server. The latter acknowledges the registration by sending a *join acknowledgement* packet. By receiving this command, the bot sends an acknowledgement to the server and *command\response* packet. The latest message contains the bot information like system information and the country code. The server sends an acknowledgement to the bot and forwards the *command\response* to the master, which acknowledges the reception of this message to the server.

The second phase aims at checking the liveness of bot clients. The server keeps sending *command\response* packet to the bot client in a frequency of four minutes. If a given bot is alive, it replies with an *acknowledgement* packet.

The action phase aims to instruct the bots to make actions at the infected hosts. The master sends *command\response* packet to the server. The server forwards this packet to the bot. By receiving the packet, the bot performs the action that is mentioned in the packet. It acknowledges its action by sending an *acknowledgement* packet. The server sends an *acknowledgement* packet to the master. Figure 1 summarizes the different involved behaviors of the 3 phases of the Mariposa botnet communication.

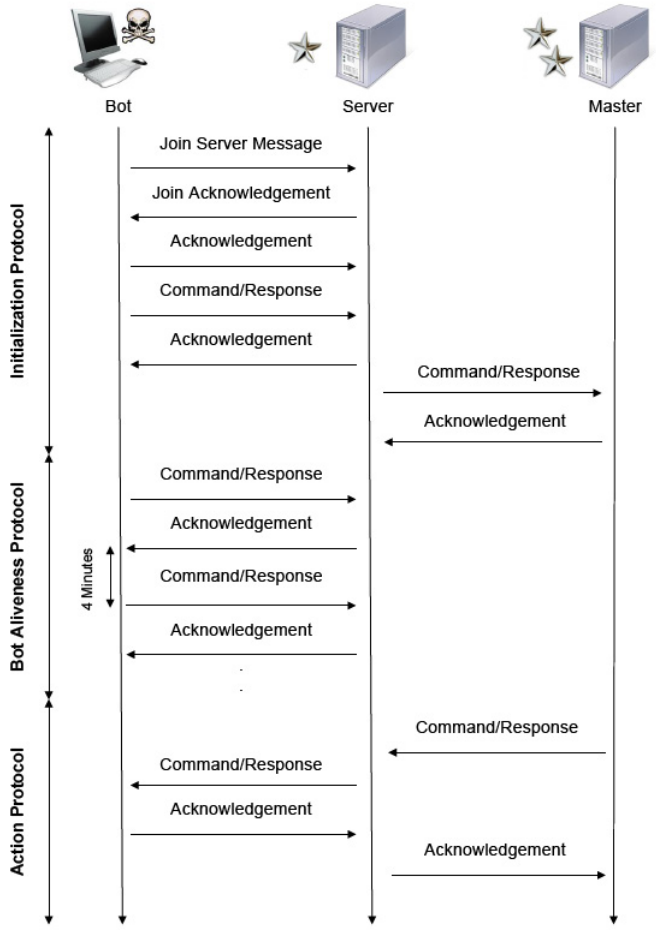

Fig. 1. Mariposa Botnet Protocol

The next section is devoted to the results of the static & dynamic analysis.

IV. Static & Dynamic Analysis

The static & dynamic analysis constitute a must when it comes to reverse engineering of malware. Actually, it allows digging into the inner-secrets of the malware code. In our analysis, we used IDA pro [3] disassembler and decompiler to analyze the Mariposa bot client. The MD5 hash of the malware variant is *3E3F7D8873985DE888CE320092ED99C5*. Before digging into the details of the analysis, we used *SysAnalyzer* [8] to get an initial insight about the client. After running this tool, we noticed that Mariposa infects the *explorer.exe* process. This process opens a UDP port, which has the identifier 1055. Moreover, *SysAnalyzer* reveals the registry keys and external references that are accessed by the Mariposa bot.

The analysis consists of getting over the obfuscation and anti-debugging techniques that employed by Mariposa as well as reaching the susceptible parts of the code that execute Mariposa bot features that we previously described in Section II. The Mariposa binary has a metamorphic code [19], comprised of various obfuscation and anti-debugging techniques. Figure 2 depicts the different phases of Mariposa bot metamorphose. The execution of the bot client has three phases: the obfuscation phase, the decryption phase and the injection phase.

In the sequel, we introduce the different phases that are related to the de-obfuscation, anti-debugging traps and different decryption layers.

A. De-Obfuscation & First Decryption Layer

Code obfuscation is nowadays a standard practice within Malware. It constitutes the concealment of the intended meaning of an integrated malicious code. It makes the code confusing and intentionally ambiguous and more difficult to interpret. In the Mariposa bot, the obfuscation starts with useless computations. These computations are done within a loop that iterates 889,976,605 times. At the end of this loop, a jump to an address is loaded into *EAX* register. As a consequence, a jump is initiated to start a routine that *XOR*s the range of data that is located between the addresses *0x41D000* and *0x41D4C0* with the constant *0x0CA1A51E5*. Afterwards, the address *0x41D047* is pushed onto the stack. As a result, the control flow is transferred to this address. The latter corresponds to an entry point of the anti-debugging traps.

B. Anti-Debugging Traps

Anti-Debugging techniques are ways for a program to detect if it runs within a controlled environment or a debugger. They are used by commercial executable protectors, packers and malicious software to prevent or slow-down the process of reverse-engineering [15]. The Mariposa bot client uses several anti-debugging techniques. These techniques make the reverse-engineering tasks as strenuous and difficult as possible. They increase the time that is required for a full analysis of the bot binary. The address *0x41D047* constitutes the entry point of the code that employs anti-debugging traps. The most important anti-debugging techniques that have been encountered in the analyzed variant of the Mariposa bot are:

- *ICE* Breakpoint (In Circuit Emulator): It is one of the Intel's undocumented instructions with opcode *0xF1*. The execution of this instruction generates a single step exception. This instruction pushes a debugger to think that a normal exception is generated by the program. It sets the single step bit in the flag register. Thus, the associated exception handler is not executed. In order to bypass this trap, we avoid the use of single step execution of the code segments that contain ICE breakpoints.
- *Stack Segment Register*: The idea is to exploit a property of the Intel x86 hardware debugging system. According to Intel x86 architecture, hardware breakpoints are not

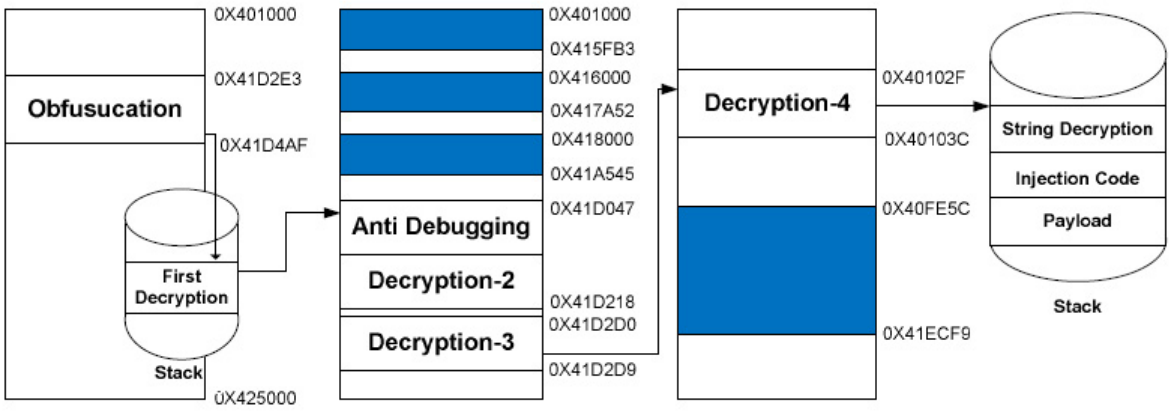

Fig. 2. Overview of Mariposa Bot

effective when it used after *POP SS* instruction. If the program traced over *POP SS* instruction, the next instruction will be executed covertly. As a consequence, the trap flag remains set. Malware checks the trap flag to detect the presence of debugger.

- *QueryPerformanceCounter* Function: It is used to compute the hardware performance. It reads the values of performance counters that are stored in some processor registers[1]. Mariposa uses this function to compare hardware activities with a threshold value and checks if a process is running under debugging mode or not.
- *GetTickCount* Function: It is located in *kernel32.dll*. It returns the number of milliseconds that the system has elapsed since its last reboot. The highest return value is *49.7* days. Malware calls the *GetTickCount* function consecutively to calculate the difference between two function calls. It allows the malware to detect the presence of a debugger.
- *OutputDebugString* Function: It is generally used by encryption programs. The function receives a string as a parameter. If a program runs under a debugger, then, the returned value of this function (value of EAX register) corresponds to the address of the string that is passed as parameter. Otherwise, it returns the value 1. To escape *QueryPerformanceCounter*, *GetTickCount* and *OutputDebugString*traps, we used an IDA Pro plugin, namely, IDAStealth [10].

C. Second, Third & Fourth Decryption Layers

This section describes all the decryption routines that are executed. After unveiling and removing the obfuscation and the anti-debugging routines, we reach the part of code that contains the decryption routines. The second layer of the decryption corresponds to an iteration of a *XOR* operation with a 32 bytes key. Each byte within the data is *XOR*ed with a byte from the key. This byte corresponds to the modulo result of data byte position with the size of the key (32 bytes). This algorithm is iterated three times for three different chunks of data. The first location of data corresponds to the range [*0x401000*, *0x415FB3*], the second location of data resides in the range [*0x416000*, *0x417A52*] and the third location of data is within the range [*0x418000*, *0x41A545*]. There exist three 32 bytes keys; each one is used in the algorithm for each chunk of data. These keys are located at the following addresses: *0x41D015*, *0x41D155* and *0x41D1B4*. Figure 3 illustrates the pseudo code of the second decryption layer. The value *x* corresponds to the key location, whereas *r1* and *r2* are the start and the end addresses of data respectively.

```
Second_Decryption_layer()
{
  Key_size=32 byte;
  Key_location = x;
  Key[]=getKey(x);//For first decryption layer.
  Start_address=r1;
  End_address=r2;
  Enc_data[]=getData(Start_address, End_address);
  for(i=0;i<Enc_data.size();i++){
    Dec_data=Enc_data[i] XOR key[i % 32];
  }
}
```

Fig. 3. Pseudo Code of the Second Decryption Layer

After executing the second layer decryption, the control flow reaches the part that is responsible of loading the imported functions. The next step consists of running another decryption routine. It *XOR*s each byte of data in the range [*0x41D000*, *0x41D21E*] with a constant key *0x39*.

After executing the second and third layer decryption, the program loads its process and thread identifiers by calling *GetCurrentProcessID* and *GetCurrentThreadID* functions. It uses some anti-debugging traps using the *QueryPerformaceCounter* and *GetTickCount* functions. The intent behind this is to check again whether the current process runs under a debugger or not. In order to check whether it runs in a sandbox technology, it verifies the presence of *sbiedll.dll* in the system. By getting over these traps, we notice that the program allocates 60,925 bytes of space from the stack. It decrypts the

[1]Contemporary processors use registers that act like performance counters. They count performance of hardware activities within the processor.

data in the range [*0x40FE5C, 0x41EC59*] by using the fourth decryption routine that is illustrated in Figure 4. The latter loads the decrypted data into the allocated space of the stack. Afterwards, Mariposa transfers its control to the stack.

```
Third_layer_decryption()
{
  Key1=getByte(0x418CA2);
  Key2=getByte(0x418CA3);
  Key1=((! key1) + key2) / 2;
  Source_address= 40FE5C;
  Enc_data[0xEDFD] = getData(Source_address,
      Source_address +0xEDFD );
  Dec_data[0xEDFD]=null;
  Dest_address = 0xXXXX;//in the stack.
  for(i=o; i<Enc_data.length ; i++){
    Dec_data[i]= (Enc_data[i] + key1) XOR key2;
    If(key1==0xFF){
      Key2= (Key2+1)  %  0xFF;
    }
    Key1= (Key1+1) %0xFF;
  }
}
```

Fig. 4. Pseudo Code of the Fourth Decryption Layer

Until this point, Mariposa code passes several phases of decryption. However, all the strings are encrypted. These strings represent API functions and magic words that will be used by the injected process. Once into the stack, the program runs another decryption routine three times. This routine decrypts all the strings that are located in *.data* section. Figure 5 illustrates the pseudo code of the string decryption.

```
Decrypt_Strings ()
{
  Start_add=0x4197E0;
  Size=0xD65;
  Enc_data[]=Get_data(Start_add,Start_add+Size);
    Key1=Get_byte(0x418CA2);
  Key2=Get_byte(0x418CA3);
  key=(key2+ ~Key1) >> 1;
  for(i=Size; i >= 0; --i){
        Dec_data[i]=(Enc_data[i]+key) XOR key2;
    key=(key++)%255;
  }
}
```

Fig. 5. Pseudo Code of String Decryption Algorithm

D. Code Injection

This section describes the process of code injection that is employed by Mariposa. Despite substantial improvement in host-based security, the code injection technique still sustains as the favorite method to compromise operating systems. The method of code injection is used to conceal evil processes inside legitimate processes. The execution of process inside another address space can be achieved in several ways. We can enumerate windows hooks [7], dll injection and direct code injection [21]. The Mariposa bot uses the Direct Code Injection (DCI) technique to inject malicious code inside the address space of *explorer.exe*. Instead of writing a separate DLL, the DCI technique copies the malicious code to the remote process directly via *WriteProcessMemory* function and starts its execution with an invocation of the *createRemoteThread* function. The direct code injection (DCI) technique can be summarized into the following steps:

- Retrieval of the handle of the remote process by calling the *OpenProcess* function
- Allocation of memory in the remote process address space in order to inject code. This is achieved by calling the *VirtualAllocEx* function
- Writing a copy of the initialized *INJDATA* structure to the allocated memory by invoking the *WriteProcessMemory* function
- Execution of the injected code via the *CreateRemoteThread* function

Before code injection, Mariposa creates some directories and files. The created directories and files are:

- Directory Path: $C:\backslash Recycler\backslash s-1-5-21$.
- Directory Path: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454$. The directory access control is set to read,write and execution permissions.
- File Name: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash Desktop.ini$.
- File Name: $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$.

Afterwards, the program calls the *GetVersion* function to get the version of the operating system. The reason behind this call resides in checking whether the operating system is a Windows NT or not. If so, it uses the *CreateRomoteThread* function [2]. At the beginning of the injection process, the program calls the *CreateToolhelp32Snapshot* function to take snapshot of the running processes in the system. It enumerates the existing processes by calling *Process32First* and *Process32Next* functions. Once *explorer.exe* process is found, it retrieves its process identifier (process ID).

After getting the process ID, the program calls *OpenProcess* function to open *explorer.exe* process. Then, it calls *VirtualAllocEX* function to allocate memory within the targeted process and *NtWriteVirtualMemory* function to write into *explorer.exe* process. Once the code is written in a virtual memory location, the program calls the *CreateRemoteThread* function in order to run the injected code.

E. Injected Thread Activity

The code that is injected into *explorer.exe* is the pivotal part of Mariposa bot. In this section, we discuss the behaviors of the injected code. To this end, we attached the process *explorer.exe* to IDA pro debugger and set a breakpoint at the entry point of the newly created thread to get full control of the execution. The thread creates a *mutex* object namely *c__kdjcpeoij*. The *mutex* object is used to ensure singular execution of the bot. The intent is to avoid a possible running of multiple bot instances, which can crash

[2] *CreateRemoteThread* function works only in Windows NT versions.

the system, or at best slow down the machine. It uses the *WaitForSingleObject* function with a predefined waiting time to ensure singular execution. Once the single instance checking is ensured, it creates two files: $C:\backslash Recycler\backslash S-1-5-21-7344526690-8558129233-739613093-1787\backslash windll.exe$ and $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash Desktop.ini$. After the file creation, the thread copies the whole bot code to $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$. At this point of execution, Mariposa uses the *WsaStartup* function to initiate the use of *Winsock DLL*, which is responsible for the socket communication. It also opens the registry key $software\backslash Microsoft\backslash WindowsNT\backslash CurrentVersion\backslash Winlogon$, and creates a new entry , namely, *Taskman*. It sets the value of this entry to $C:\backslash Recycler\backslash S-1-5-21-7524899924-6962119414-608760223-8454\backslash windll.exe$ in order to make a direct injection of code when the machine reboots. It also creates another entry named *shell* with the value $C:\backslash Recycler\backslash S-1-5-21-7344526690-8558129233-739613093-1787\backslash windll.exe$.

At this stage, the bot creates two pipes. The first one is $\backslash\backslash.\backslash pipe\backslash cdcpr55$ whereas the second is an anonymous pipe. The first pipe is created in *pipe_access_inbound* mode, which supports client to server transfer only. Once the pipes are set, the program calls the *InternetOpen* function in order to use the *WinInet* library functions. Mariposa bot uses three hard-coded domain names to resolve the IP address of the C&C server. It picks the first domain name and sends the encrypted magic word to the resolved IP address, and waits for the reply from the server. If the server does not respond, it picks the second or third domain name and tries to connect to the server using the resolved IP address. The domain names that are used for this Mariposa variant are:

- Shv4.no-ip.biz
- Shv4b.getmyip.com
- Booster.estr.es

The sequence of actions that are taken by the Mariposa bot to reach the server and receive commands are:

- *Inet_addr* function is called to convert the domain names into a proper address.
- The bot retrieves the host information from the corresponding host name using the *gethostbyname* function.
- It calls the *htons* function, which converts a unsigned short number from host to a TCP/IP network byte order [3].
- It encrypts the magic word (*bpr1* is the magic word in this variant of Mariposa). The encryption/decryption algorithm is detailed in [9].
- It sends the magic word using the *sendto* function.
- It receives a reply from the server using the *recvfrom* function.

[3]Network byte order defines the bit-order of network addresses as they pass through the network. The TCP/IP standard network byte order is big-endian. In order to participate in a TCP/IP network, little-endian systems usually bear the burden of conversion to network byte order [6].

- It decrypts and decodes the received command. The bot can then trigger appropriate actions that are instructed by the master.

V. Modules

A. Spreader Module

The Mariposa bot comes with a spreader module. This module breaks into three different components, namely, USB spreader, MSN spreader, P2P spreader. In the Mariposa botnet, the master can send commands to enable and disable the spreaders. In the sequel, we introduce these different components:

- *USB spreader*: At the beginning, the program creates a new top-level window by executing *CreateWindowEx* function. The returned handle is used by the *RegisterDeviceNotification* function in order to receive notification from the system when a flash drive is inserted. Once a user inserts a USB key, it locks the *autorun.inf* file and modifies the file accordingly. As a result, no software or malware can launch an auto-run. The file stays locked until a user decides to remove the USB key. Mariposa makes a copy of itself into the USB key. Figure 6 shows the content of the *autorun.inf* file.

```
debug048 : 00FE0040 aAutorunOpenRec db '[autorun]',0Dh,0Ah
debug048 : 00FE0040 db 'open=RECYCLER\autorun.exe',0Dh,0Ah
debug048 : 00FE0040 db 'icon=%SystemRoot%\system32\SHELL32.dll,4',0Dh,0Ah
debug048 : 00FE0040 db 'action=Open folder to view files',0Dh,0Ah
debug048 : 00FE0040 db 'shell\open=Open',0Dh,0Ah
debug048 : 00FE0040 db 'shell\open\command=RECYCLER\autorun.exe',0Dh,0Ah
debug048 : 00FE0040 db 'shell\open\default=1',0
```

Fig. 6. The *autorun.inf* Content

- *MSN spreader*: The Mariposa bot infects MSN messenger by hooking sending and receiving functions. The MSN spreader is activated if a bot receives an enabling command. This command contains a custom link, which is used to download a bot in the user's machine.
  After receiving the MSN spreader activation command, the bot looks for the *msnmsgr.exe* process. This operation is done periodically if the process is not running in the system. Once the *msnmsgr.exe* process is found, the Mariposa bot retrieves its process identifier. Then, it calls the *OpenProcess* function to get the handle of this process. Afterwards, it creates a duplicate handle of the current process by calling *GetCurrentProcess* and *DuplicateHandle* functions. At this point, the Mariposa bot starts a new routine, which is responsible for injecting code inside the virtual address space of *msnmsgr.exe* process. This routine is called twice. In the first call, it allocates 256 bytes of space by calling *VirtualAllocEX* function and injects code using *NtWriteVirtualMemory* function. In the second call, it injects string utility functions and the custom link that is sent by the master. It creates a thread by calling *CreateRemoteThread* function.

After the injection process, the bot hooks *ws2_32_send* function in order to make the injected code executed for each message that is sent from a user to a recipient. This is done by calling the *VirtualProtectEx* function to allows writing in the virtual memory. At the end, it calls the *NtWritevirtualMemory* function to overwrite with the address of injected code.

- *P2P spreader*: When the bot receives a command that enables the P2P spreader, the program calls the *GetEnvironmentVariable* function in order to get the registry entry for the current user. The intent behind this resides in checking if P2P applications are installed or not. The Mariposa bot looks for the following P2P applications in the system: Ares, BearShare, iMesh, Shareaza, Kazaa, DC++, eMule and LimeWire. Once, it detects the presence of a P2P application, it copies itself into the shared folder with a fake name that is issued by the master. Figure 7 shows a screenshot of P2P registry keys that are accessed by the Mariposa bot.

```
debug046 : 00FC0A8C aLocalSettingsA db '\Local Settings\Application Data\Ares\My
debug046 : 00FC0AC3 aDownloaddir db 'DownloadDir',0
debug046 : 00FC0ACF aSoftwareBearsh db 'Software\BearShare\General',0
debug046 : 00FC0AEA aSoftwareImeshG db 'Software\iMesh\General',0
debug046 : 00FC0B01 aSoftwareSharea db 'Software\Shareaza\Shareaza\Downloads',0
debug046 : 00FC0B26 aCompletepath db 'CompletePath',0
debug046 : 00FC0B33 aSoftwareKazaaL db 'Software\Kazaa\LocalContent',0
debug046 : 00FC0B4F aSoftwareDc db 'Software\DC++',0
```

Fig. 7. P2P Registry Keys

B. Uploader and Downloader Modules

During the analysis of the main thread activity, we noticed that when the bot receives update/download commands, it triggers two new threads. To debug these threads in IDA pro, we set a breakpoint at the beginning of each thread. When Mariposa bot transfers its control to one of these threads, we suspended the original thread in IDA pro and continued debugging with the new thread.

a) Thread 1: Mariposa starts this thread when the bot receives a download command. After receiving this command, the bot checks the command. If the latter corresponds to *descargar*[4], the thread launches the following activities:

- It targets the temporary location in the system to download a new executable.
- It calls the *InternetOpenUrl* function with the supplied *url*.
- If the *InternetOpenUrl* function succeeds, the bot creates a file in the temporary location by calling the *CreateFile* function.
- It downloads the file using the *InternetReadFile* function.
- It writes the file onto the disk by invoking the *WriteFile* function.
- It uses the *CreateFile* function again to create the file.

After downloading the file, the bot checks the first two bytes to ensure whether the downloaded file is an executable or not. If so, it runs the file by calling the *CreateProcess* function and exits the thread by calling the *ExitThread* function.

b) Thread 2: This thread starts when the bot receives an upload command. After receiving the command, the bot checks the command and compares it with *subir*[5]. If the comparison is successful, the thread executes the following activities:

- It calls the *InternetCrackUrl* function to read different *url* components.
- By getting the *url* components, it calls the *InterConnect* function to set a connection with the *url*.
- It uses the *HttpOpenRequest* function to create an *HTTP* request.
- It invokes the *InternetReadFile* function to read data to be sent.
- It sends the data using the *HttpSendRequest* function.
- Finally, it closes the connection handle using the *InternetCloseHandle* function.

After uploading the file, the thread calls the *exitthread* function to close the thread.

C. Components Diagram

By conducting a thorough reverse-engineering task, we noticed that Mariposa bot has complex interactions between its functional components. Figure 8 illustrates the different interactions between the different functional components. For the space constraint, we show only the key components of the Mariposa bot.

VI. Related Work

The analysis of botnets is a worthwhile exercise. It aims at uncovering the employed technologies in terms of obfuscation, encryption, injection and communication. It is only with the insights gained from such analysis that we can design and implement efficient detection, eradication and prevention techniques. In the sequel, we discuss the state of the art proposals in this area of research that is botnet analysis.

In [12], the authors presented the analysis of an HTTP botnet, namely, BlackEnergy. The analysis provided a detailed information about the botnet architecture, commands and communication patterns. BlackEnergy is a web-based tool that allows to build bot binaries. The main threat of this botnet is Distributed Denial of Service (DDoS). Chiang and Lloyd studied the Rustock rootkit in [1]. This rootkit contains a spam bot module. The authors studied the network traces and noticed that the traffic is encrypted by RC4 algorithm. The Rustock rootkit has multiple levels of obfuscation, which makes it hard to be detected. The main usage of this tool resides in mail spamming. In addition to the network analysis, the authors were able to extract the encryption key of the communication. Daswani et al. [2] put forward a detailed case study of clickbot.A. This bot is responsible of click fraud attacks.Their analysis covered the main components of this botnet as well as the commands and configuration. Porras et al. reverse-engineered the Storm botnet in [14]. They detailed

[4]Descargar is a Spanish word, which means download

[5]Subir is a Spanish word, which means upload

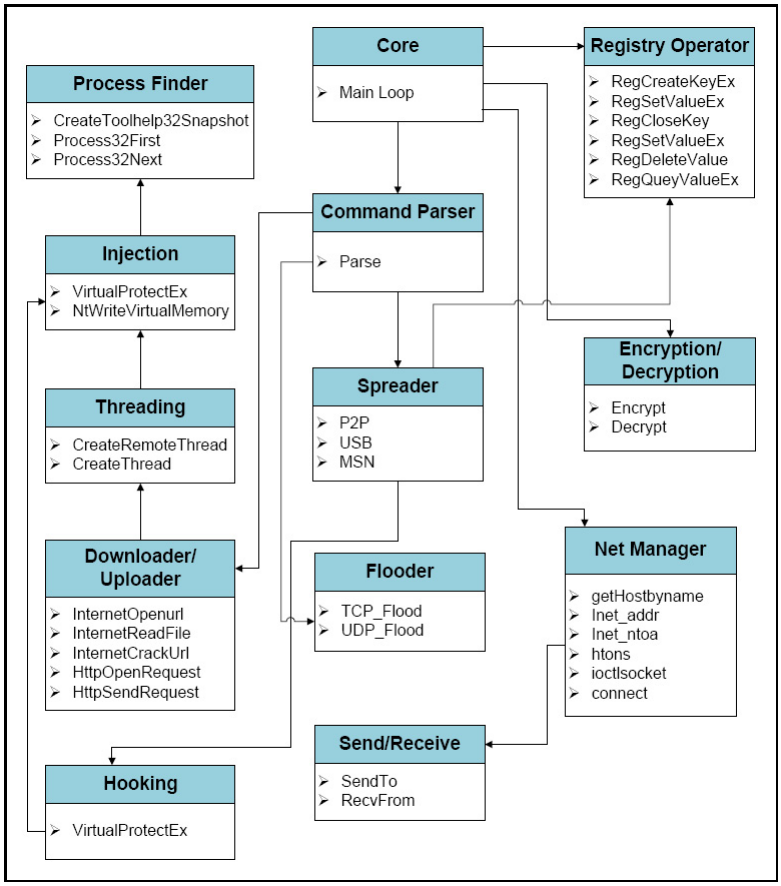

Fig. 8. Component Diagram

the techniques that were used to hide the binary and how it has been obfuscated. Storm botnet uses the Overnet protocol for the communication. This botnet is used to send email spams and DDoS attacks. In [5], [11], the authors investigated the the Storm botnet by studying the encryption key generation algorithm that is used for communication between different peers. In [4], the authors reported their analysis of the Nugache instance.They analyzed the communication pattern between different principals. The communication is based on a key exchange protocol. In Nugache botnets, the bot herder instructs bots to listen to a specific IRC channel in order to initiate a DDoS attack. The authors addressed extra aspects of their initial analysis and estimated the size of the Nugache botnet using a bot client crawler.

VII. Conclusion

The Mariposa toolkit is a one of the most prominent botnet technologies that are being used nowadays. This toolkit provides a lot of features, which make it interesting to investigate. The bot employs direct code injection into *explorer.exe* process, which makes it resilient against firewalls that consider system processes legitimate and avoid to check for their integrity. In addition, the bot shows some characteristics that are atypical to the majority of other bots. It incorporates spreading mechanisms, it infects USB keys and uses MSN messenger and P2P applications to distribute itself widely through the Internet. Furthermore, the Mariposa bot has the ability to install different malware on the compromised machine. The capabilities of the Mariposa botnet together with its wide propagation attracted us to investigate its inner-secrets. Accordingly, we conducted a twofold analysis: First, we set up a closed environment to analyze the network behaviors of the botnet. This practice allowed us to grasp how the different botnet components interact with each other at the network level. By monitoring the traffic, we concluded the existence of three protocols, namely, the initialization protocol, the beaconing protocol and the action protocol. The second phase of the analysis consisted of reverse-engineering the code. This phase breaks into getting over the obfuscation and encryption layers and understanding how the bot injects itself into the *explorer.exe* process. This code represents the entry point to different malicious activities that are performed by the bot. By isolating the injected code, we managed to debug it and reach the fragments of code that malicious actions. By sending different commands, we grasped the behaviors that these commands involve. Our general observation is that botnets are becoming blended threats since they combine the capabilities of worms, viruses and trojan horses. In addition, from this exercise we learn that some sequences of API calls can be a good source to detect nefarious bot activities. For

instance, a process that does not need to P2P registry entries and do so by calling some API functions, can be suspicious. Finally, the rise of UDP traffic in the network can give a clue about the presence of a Mariposa infection in the network. As future work, we intend to study other bots that are of interest to the security community and also to elaborate detection, eradication and prevention techniques.

ACKNOWLEDGEMENTS

We wish to express our appreciation to National Cyber-Forensics Training Alliance-Canada for the fruitful collaboration that we had with different members of the organization. We would also like to thank Ms. Lynne Perrault for her support. We would like also to acknowledge the precious insights that we gained from the initial publications of Defence Intelligence on the Analysis of Mariposa.

REFERENCES

[1] Ken Chiang and Levi Lloyd. A case study of the rustock rootkit and spam bot. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007. USENIX Association.

[2] Neil Daswani and Michael Stoppelman. The anatomy of clickbot.a. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007. USENIX Association.

[3] DataRescue. Idapro - multi-processor disassembler and debugger. http://www.hex-rays.com/idapro/, 2009.

[4] David Dittrich and Sven Dietrich. P2p as botnet command and control: a deeper insight. In *3rd International Conference on Malicious and Unwanted Software (MALWARE)*, pages 41–48, Piscataway, NJ, USA, 7-8 Oct. 2008 2008. Appl. Phys. Lab., Univ. of Washington, Washington, DC, USA, IEEE.

[5] Thorsten Holz, Moritz Steiner, Frederic Dahl, Ernst Biersack, and Felix Freiling. Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm. In *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 1–9, Berkeley, CA, USA, 2008. USENIX Association.

[6] IBM. Communications server. http://publib.boulder.ibm.com/infocenter/zos/v1r9/index.jsp?topic=/com.ibm.zos.r9.halc001/oawshs.htm, 2009.

[7] Iczelion. Tutorial 24: Windows hooks. http://win32assembly.online.fr/tut24.html, 2009.

[8] IDEFENCE. Sysanalyzer overview. http://labs.idefense.com/files/labs/releases/previews/SysAnalyzer/.

[9] Defence Intelligence. Mariposa botnet analysis. Technical report, October 2009.

[10] jan newger. Idastealth plugin. http://newgre.net/idastealth, 2010.

[11] Brian Krebs. Storm worm dwarfs world's top supercomputers, August 2007.

[12] Jose Nazario. Blackenergy ddos bot analysis. Technical report, Arbor Networks, 2007.

[13] Palo Alto Networks. Palo alto networks. http://www.paloaltonetworks.com/researchcenter/2009/11/mariposa-how-at-exposed-are-we/, 2009.

[14] Phillip Porras, Hassen Sadi, and Vinod Yegneswaran. A multi-perspective analysis of the storm (peacomm)worm. Technical report, Computer Science Laboratory, SRI International, 2007.

[15] SecurityFocus. Windows anti-debug reference. http://www.securityfocus.com/infocus/1893, 2009.

[16] securixLive. Securix-nsm project page. http://www.securixlive.com/knoppix-nsm/, 2005-2009.

[17] Butterfly Network Solutions. Butterfly network solutions. http://bfsystems.net/index.php?page=9&subpage=4, 2009.

[18] Butterfly Network Solutions. Butterfly network solutions. http://bfsystems.net/index.php?page=6&subpage=4, 2009.

[19] P. Szr and P. Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, pages 123–144. Virus Bulletin, 2001.

[20] VMware. Vmware server. http://www.vmware.com/products/server/, 2009.

[21] Wikipedia. Dll injection. http://en.wikipedia.org/wiki/DLL_injection#cite_note-Waddington-9, 2009.