HOW TO WRITE A SIMPLE EXECUTABLE PACKER IN C BY GUNTHER

FOREWORDS

The time has come to write a tutorial on “How to write a Simple Packer”. “How to write a simple Packer” will definitely open floodgates for script kiddies to pack their malware.

This tutorial is written to provide better understanding of a Packer and the development of a Packer in C. Before proceeding further, get the source code from here, so that you are able to see the code and my messy comments as you follow reading this through. Readers are required to know the basics of WIN32 api and C. Please note that this article and the source code provided has been written for learning purposes and not for complex functionality, thus extra features of a Packer depends entirely on the creativity of the developer.

This Packer is developed in such a way that it will function as a framework, allowing it to be further implemented by adding or changing functions.

After promising Shub-Nigurrath for a long time, I have decided finish this tutorial, writing “How to write a Simple Packer”. The tutorial will cover different issues:

- How does a Packer works.
- How to modify them
- Practical examples of what we can do

I hope that this could revive interest in Packing and UnPacking. Today’s topic will go over “How to write a Simple Packer”.

1 TABLE OF CONTENTS

DISCLAIMER/LICENSE

VERIFICATION

ARTeam.esfv can be opened in the ARTeamESFVChecker to verify all files have been released by ARTeam and are unaltered. The ARTeamESFVChecker can be obtained in the release section of the ARTeam site: http://releases.accessroot.com

1.1 BASIC CONCEPTS

In this tutorial, we are going to learn the basic concepts on the Packer Architecture and how does it work..

There has been a growing trend in software protection technology. Even thought software protection had been around for eons but many people are still puzzled at how does it work. Thus, i will go through with you on building your own Packer. In this tutorial, I will try to fill in required information about how it works and how to code one yourself.

Let’s start getting our hands dirty...

1.1.1 PACKER ARCHITECTURE

The concept of packing two or more files of any type together will result in a single executable file. This concept has been around for quiet some time now. Before we start programming, we need to answer a basic question: what is a Packer? As the name implies (Packer) - a Packer is a computer application that packs several files quickly together, thus obfuscating your application. Now we know what a Packer is and have an idea of its functions, lets move on to the next step. A Packer comprises of 2 important components, the main Packer component (Which takes care of adding the included files to the final executable and packing it), and the Stub component, is a compiled program that when opened will automatically extract the included files.

However, there are still a lot of people who are confused about how packing works. I hope that this is simple to understand. Basically, we can add files to the final executable either by adding it to the EOF (End Of File), or by adding it to the final executable’s resources using BeginUpdateResource, UpdateResource, EndUpdateResource APIs. For this particular tutorial, we shall use the EOF method.

Ok, now let’s plan our program, what should such Packer do and what it should not. We shall discuss the LOGIC flow of the application. The pseudo instructions that the application shall follow.
Packer will:

1.) Add included files and produce a resultant executable.
2.) Encrypt the included files.
3.) Compress the included files.
4.) Unpacked the included files once executed.

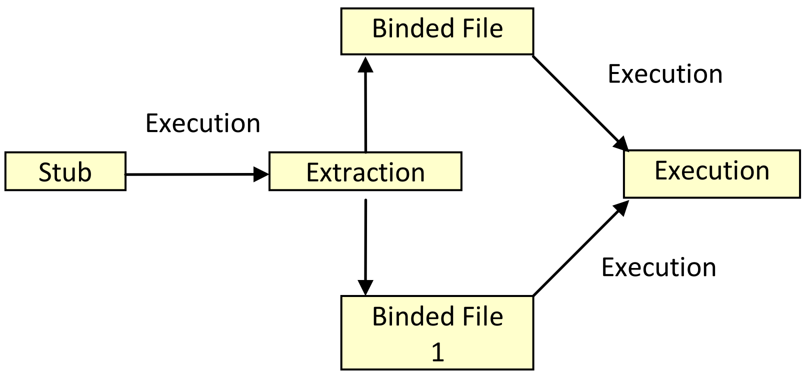

Figure 1 - A simple model of a stub

The stub is the crux of the program. It is the stub's mission to carry out file extraction & file execution or other custom option which the developer has included with his application. In the past, a lot of time was spent trying to keep the stub's size down so that the resultant executable may appear unnoticed if bound to other files.

In order to prevent Reverse Engineers, like the guys from ARTeam from having an easy time viewing the resultant executable file's settings. Nowadays, Packers have the option which allows users to add different data encryption algorithms, from a simple Xor to RC4, BlowFish or Camellia. This is supposedly to add another layer of security preventing them from reversing it and view the settings.

1.2 IMPLEMENTATION: CONVERTING LOGIC INTO CODE

We shall use Visual C because it is much easier and simple to understand as far as novice audience is concerned. Let's start the ball rolling and start your VC6 environment! Beam me up Scottie.

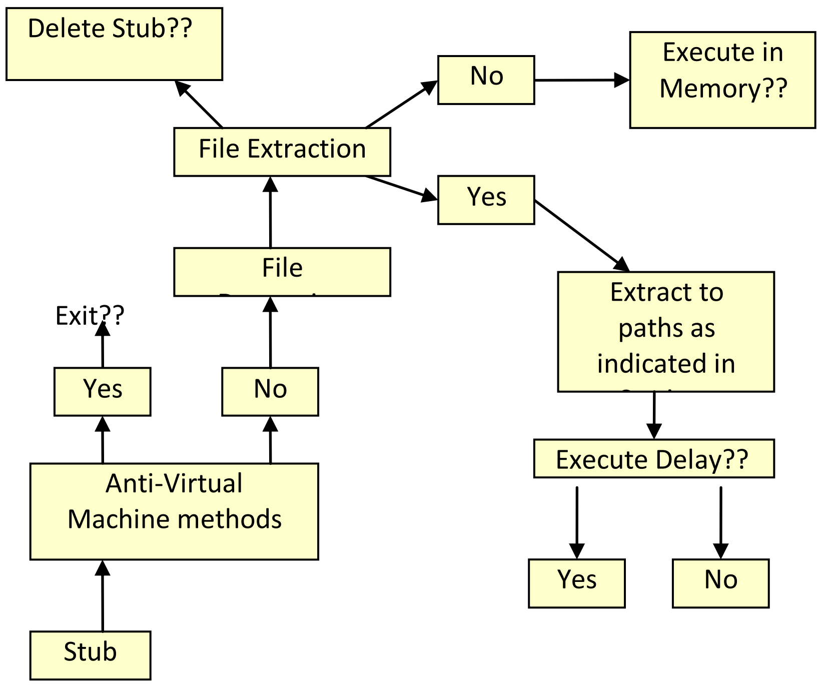

Figure 2 - A Stub design based on Y0da's Protector.

1.2.1 STRUCT OF FILE INFO

Firstly, we need to define a struct for each of the included file so that when we add it to the resultant executable, it is easier for us to extract it out to the location which we need to extract it to.

The above model supports binding of unlimited amount of files together into a single executable file. The figure above shows the stub supports the following features.

- Anti-Virtual machine - Detect whether the executable is being executed on a Virtualised Machine.
- Data Encryption – Encrypts the files to provide obfuscation.
- File extraction
- Memory execution - Does not require writing the file to disc in order to execute it.
- Delayed execution - Execute a file after a delay or on a specific time/date.
- Standard execution
- Melt Stub - Deletes the final executable.

```
struct file_info{
    char            szFileName[40];     // Filename.
    unsigned long   size;               // The compressed size of the file.
    unsigned long   actual;             // The actual file size of the file.
    short           path;               // The path to be extracted to.
    short           run;                // Whether the file need to run after extracting.
}
```

In the application, I have coded it to enable users to add and remove files simply by right-clicking on the list-view and select "Add" from the context menu. It will pop up a dialog box asking where the file would be extracted to once it is unpacked and whether it should execute.

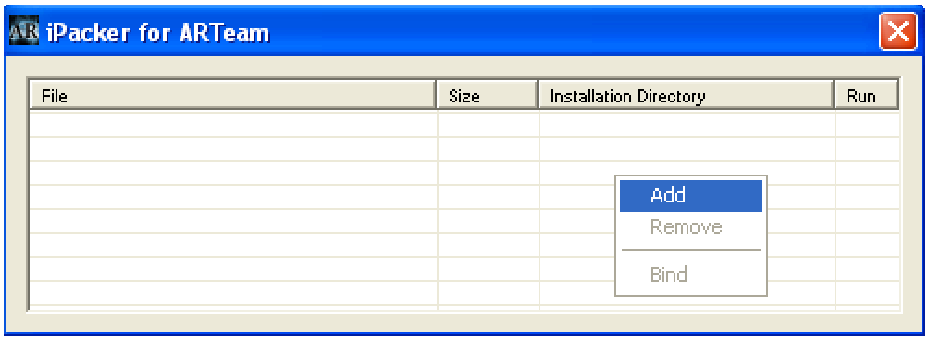

Figure 3 - Gui of Application.

After we had added at least 2 files, we can right-click on the list-view and select "Bind" from the context menu. A dialog box will pop up asking you where the final resultant executable will be saved to.

1.2.2 EXTRACTING STUB AND PLACING IT ONTO THE NEW EXECUTABLE

Inside the source code, before binding the included files, we will first read in our existing Stub and write it to the beginning of the new resultant executable.

```
BOOL ExtractStub( char *szLoc ){
    HRSRC               rc;
    HGLOBAL     hGlobal;
    HMODULE     hThisProc;
    DWORD       dwSize, dwBytesWritten;
    unsigned char       *lpszData;

    hThisProc= GetModuleHandle(NULL);
    rc= FindResource(hThisProc, MAKEINTRESOURCE(IDR_RT_EXE), "RT_EXE");

    if( hGlobal= LoadResource(hThisProc, rc) ){
            lpszData= (unsigned char *)LockResource(hGlobal);
            dwSize= SizeofResource(hThisProc, rc);
            hLoader=    CreateFile(szLoc,    GENERIC_READ    |    GENERIC_WRITE,    FILE_SHARE_READ    |
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

            if( hLoader==INVALID_HANDLE_VALUE ){
                    return FALSE;
            }else{
                    WriteFile(hLoader, lpszData, dwSize, &dwBytesWritten, NULL);
            }
    }
    if( dwBytesWritten!=dwSize ){
            MessageBox(NULL, "Error writing stub file.", NULL, MB_ICONERROR | MB_OK);
            return FALSE;
    }else{
            return TRUE;
    }
}
```

After writing the Stub into the resultant executable, we shall go through the list from list-view and add in the included files and report errors when there is difficulties writing the included files into the final executable.

```
if( !WriteFiles(iIndex) ){
    MessageBox(hDlg, "Error writing files.", NULL, MB_OK);
    CloseHandle(hLoader);
    return FALSE;
}
```

Using my own WriteFiles function, I will add in the included files by first encrypting them using Camellia 128bit. There are no special reasons as to why i chose Camellia, you can even use RC4, BLOWFISH, Triple DES or any other algorithm.

The following will show roughly what I will do to each included file. Basically for all the files in the ListView, i need to process it and packed it. I ran them thru a loop and read every single file off the ListView and compress them using aPLib. Next, i encrypt the files using Camellia. As Camellia is a block cipher algorithm, i need to pad the files.

```
int WriteFiles( int nFileNum ){
        ………………………………………………………………..
        ………………………………………………………………..
        for( i=0; i<nFileNum; i++ ){
                ZeroMemory(&fd, sizeof fd);
                ListView_GetItemText(hwndList, i, 0, szPath, MAX_PATH);

                hFile= CreateFile(szPath, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL);
                if( hFile==INVALID_HANDLE_VALUE ){
                        return 0;
                }
                dwSize= GetFileSize(hFile, NULL);
                // Start of aPLib compression.
                /* allocate memory */
                if( (data= (byte *)malloc(dwSize))==NULL || (packed= (byte
*)malloc(aP_max_packed_size(dwSize)))==NULL || (workmem= (byte
*)malloc(aP_workmem_size(dwSize)))==NULL ){
                        MessageBox(NULL, "ERROR: Not enough memory.", "Memory not Enough Error.",
MB_OK);
                        return 1;
                }
        ………………………………………………………………..
                // Compress data block.
                outsize= aPsafe_pack(data, packed, dwSize, workmem, callback, NULL);

                if( outsize==APLIB_ERROR ){
                        MessageBox(NULL, "ERROR: An error occured while compressing.", "Compression
Error.", MB_OK);
                        return 1;
                }
                dwSize= outsize;
                // End of aPLib compression.
                pfile_data->actual= dwSize;

                temp= dwSize%(ENC_LEN/8);
                dwSize += (ENC_LEN/8 - temp);

                pfile_data->size= dwSize;
                strcpy(pfile_data->szFileName, PathFindFileName(szPath));

                ListView_GetItemText(hwndList, i, 2, szDir, sizeof szDir);

                if( !strcmp(szDir, "System") ){
                        pfile_data->path= 1;
                }else if( !strcmp(szDir, "Temporary") ){
                        pfile_data->path= 2;
                }else{
                        pfile_data->path= 3;
                }
        ……………………………………………………………………..
                Camellia_Ekeygen(128, rawKey, keyTable);

                SetFilePointer(hLoader, 0, NULL, FILE_END);
                WriteFile(hLoader, pfile_data, sizeof fd, &dwBytesWritten, NULL);

                temp= 0;

                while( temp<dwSize  ){
                        // Start  of Camellia.
                        char *plaintext= packed;
                        unsigned char *enc_data= malloc(ENC_LEN/8);
                        enc_data[0]= 0;
                        plaintext += temp;

                        Camellia_EncryptBlock(ENC_LEN, plaintext, keyTable, enc_data);
                        WriteFile(hLoader, enc_data, ENC_LEN/8, &dwBytesWritten, NULL);
                        // End of Camellia.
                        if( dwBytesWritten!=dwBytesRead ){
                                //return 0;
                        }
                        free(enc_data);
                        enc_data= NULL;
                        temp += ENC_LEN /8;
                }
        ……………………………………………………………………..
                CloseHandle(hFile);
        }
        ……………………………………………………………………..
        CloseHandle(hLoader);
        return i;
}
```

What we have covered so far is for the Binder component. Now we shall talk a bit on the Stub component. Basically, the Stub will just decrypts the files and extract them to the location as we have chosen.

```
int StubMain( void ){
        ……
        GetModuleFileName(NULL, szThisFile, _MAX_FNAME);

        hStub= CreateFile(szThisFile, GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
NULL);

        SetFilePointer(hStub, STUB_EOF, NULL, FILE_BEGIN);

        while( ReadFile(hStub, pfile_data, sizeof fd, &dwBytesRead, NULL) && dwBytesRead ){
        …………………………………………………………………..
                Camellia_Ekeygen(ENC_LEN, rawKey, keyTable);

                if( pfile_data->path==1 ){
                        GetSystemDirectory(szPath, sizeof szPath);
                }else if( pfile_data->path==2 ){
                        GetTempPath(sizeof szPath, szPath);
                }else{
                        GetWindowsDirectory(szPath, sizeof szPath);
                }
        …………………………………………………………………..
                hFile= CreateFile(szTempPath, GENERIC_WRITE | GENERIC_READ, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
                if( hFile==INVALID_HANDLE_VALUE ){
                        return 1;
                }

                buf= malloc(pfile_data->size);
                if( !buf ){
                        return 2;
                }
                // Start of Camellia decryption.
                ReadFile(hStub, buf, pfile_data->size, &dwBytesRead, NULL);
                temp= 0;
                while( temp<pfile_data->size ){
                        char *crypt= buf+temp;
                        Camellia_DecryptBlock(ENC_LEN, crypt, keyTable, dec_data);
                        temp += ENC_LEN/8;

                        if( temp>pfile_data->actual ){
                                WriteFile(hFile, dec_data, pfile_data->actual % (ENC_LEN/8),
&dwBytesWritten, NULL);
                        }else{
                                WriteFile(hFile, dec_data, ENC_LEN/8, &dwBytesWritten, NULL);
                        }
                }
                FlushFileBuffers(hFile);
                CloseHandle(hFile);

                hFile= CreateFile(szTempPath,GENERIC_READ, 0, NULL, OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);

                // Start of aPLib decompression.
                hActualFile= CreateFile(szPath, GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
                if( hFile==INVALID_HANDLE_VALUE ){
                        return 1;
                }
                // Allocate memory.
                if( (packed= (byte *)malloc(GetFileSize(hFile,NULL)))==NULL ){
                        MessageBox(NULL, "ERROR: Not enough memory.", "Memory not enough Error.",
MB_OK);
                        return 1;
                }
                ReadFile(hFile,packed,GetFileSize(hFile,NULL),&dwBytesRead,NULL);
                depackedsize= aPsafe_get_orig_size(packed);

                if( depackedsize==APLIB_ERROR ){
                        MessageBox(NULL, "ERROR: Compressed data error.", "Error in Compressing
Data..", MB_OK);
                        return 1;
                }
                // Allocate memory.
                if( (data= (byte *) malloc(depackedsize))==NULL ){
                        sprintf(dec_data,"%ld",depackedsize);
                        MessageBox(NULL,dec_data,"",MB_OK);
                        MessageBox(NULL, "ERROR2: Not enough memory.", "Memory not enough Error2.",
MB_OK);
                        return 1;
                }
                // Decompress data.
                outsize= aPsafe_depack(packed, pfile_data->actual, data, depackedsize);

                // check for decompression error.
                if( outsize!=depackedsize ){
                        MessageBox(NULL, "ERROR: An error occured while decompressing.", "Error in
decompressing.", MB_OK);
                        return 1;
                }
                WriteFile(hActualFile, data, outsize, &dwBytesWritten, NULL);
                // End of aPLib decompression.
                CloseHandle(hFile);
                CloseHandle(hActualFile);
                free(dec_data);
                free(buf);

                if( pfile_data->run ){
                        ShellExecute(NULL, "open", szPath, NULL, NULL, SW_HIDE);
                }
        }
        CloseHandle(hStub);
        return 0;

}
```

1.3 CONCLUSIONS

In this tutorial, we had a look at how a Packer works in principle and how to make a really simple one. We have successfully built our very own Packer, which was the goal of this tutorial.

If you have any further information regarding making your own Packer or how to improve on the design. I would be more than happy to hear from you.

1.4 GREETINGS

This tutorial is dedicated to all the components of ARTeam and Shub for editing it. Big thanks to Shub.

And of course, to you, who decided to read this document, because without your support and contribution this task wouldn't be worth to be done.