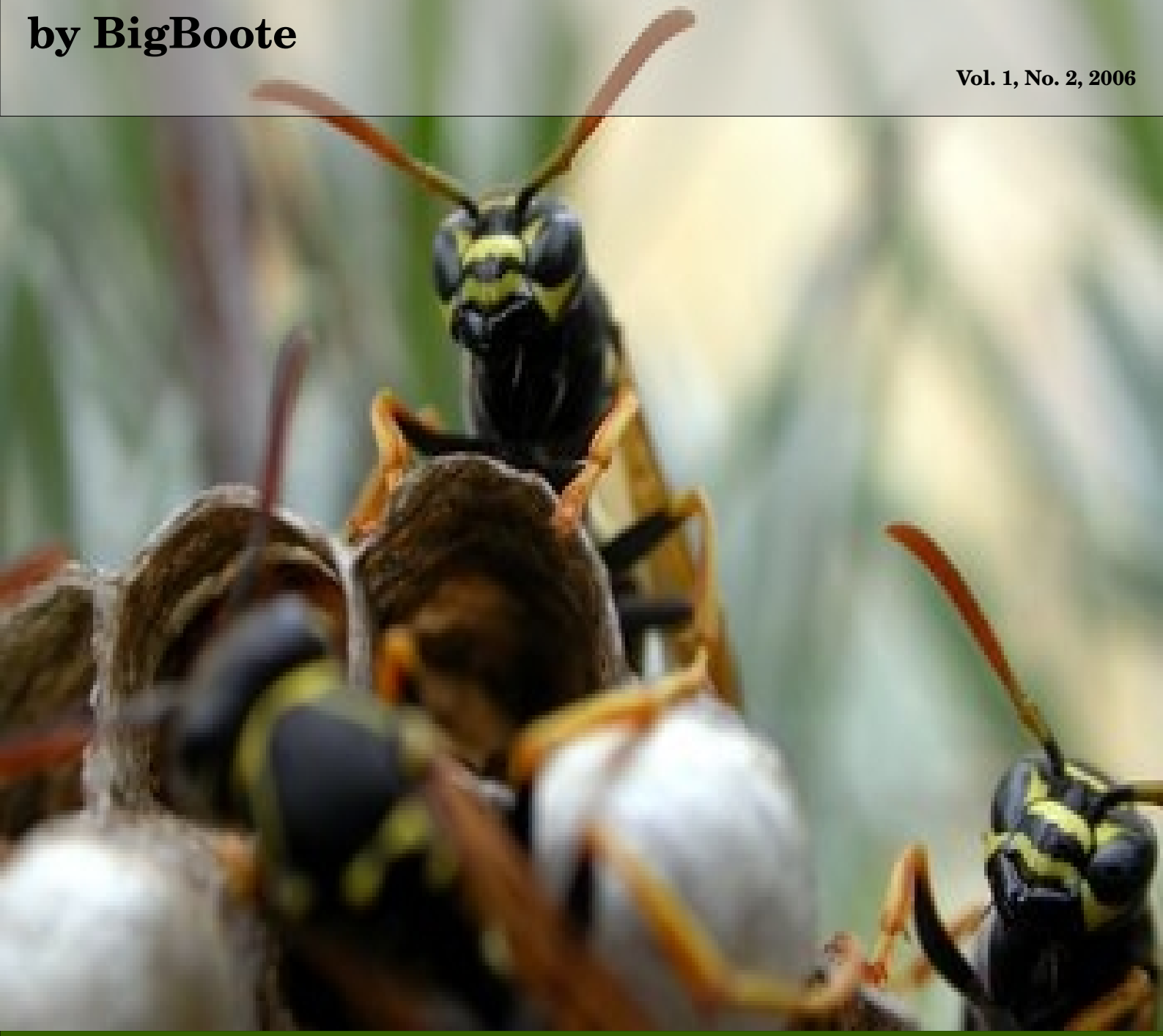
CodeBreakers Magazine
Security & Anti-Security - Attack & Defense

How to Write Your Own Packer

by BigBoote

Vol. 1, No. 2, 2006



Abstract:

Why write your own packer when there are so many existing ones to choose from? Well, aside from making your executables smaller, packing is a good way to quickly and easily obfuscate your work.

How to Write Your Own Packer

By BigBoote

Why write your own packer when there are so many existing ones to choose from? Well, aside from making your executables smaller, packing is a good way to quickly and easily obfuscate your work. Existing well-know packers either have an explicit 'unpack' function, or there are readily available procdump scripts for generating an unpacked version.

1 Intro

Why write your own packer when there are so many existing ones to choose from? Well, aside from making your executables smaller, packing is a good way to quickly and easily obfuscate your work. Existing well-know packers either have an explicit 'unpack' function, or there are readily available procdump scripts for generating an unpacked version.

Since this document has quickly exploded in length I'm going to break it up into separate installments. In this installment I will cover the qualitative aspects of producing a packer. I'll discuss what you're getting into and how the packer is structured in general. I'll briefly discuss some pitfalls, and I'll give some links to technical information you will need to be familiar with before going into the next installments.

In the next two installments I'll go into details of how to implement the components of the packer and how I usually go about producing them.

2 What You're Getting Into

It's not really hard, per se, but it is rather tedious code. Lots of pointer manipulation and translation to keep track of. Aside from that, if you can write code to add and subtract integers and do file IO, you've got all the skill needed! As mentioned, it is tedious code so you will probably do well to not attempt this coding on a hangover; trust me, I know.

FYI, the last packer I produced was fairly full-functioned (exes and dlls, several compression algorithms with debug capability and advanced support such as TLS (critical for Delphi apps)) and it weighed in at about 3700 lines for the packer tool and about 1000 lines for the decompression stub it embeds in the target. That's somewhere around 70 printed pages of code. So, not a huge app, but not a tiny one either. The first one I produced took about 1.5 weeks to produce including research and bug fixing. Subsequent ones took far less since I had already done the hard part, which is figuring out how. Hopefully this document will save you that time as well!

You do not have to use assembler for the most part. If you can part with supporting some esoteric features, you won't have to use it at all. All of that is relevant for the decompression stub only anyway. The packer can be in Logo or Object-Oriented COBOL if you like.

OK, enough of the blahblahblah, on to technical stuff....

3 Big Picture

Simple. Executable is analyzed, transformed, and an extra piece of code is attached which gets invoked instead of the original program. This piece is called a 'stub' and decompresses the image to its original location. Then it jumps to that original location. But you know this already.

Sounds simple, but there are pitfalls that await you. Some of these include:

- Support for simplified Thread Local Storage, which is key in supporting Delphi applications
  Support for code relocation fixups in dlls if you care about packing dlls. Recall ActiveX controls are dlls too, as are other common things you might be interested in packing

- Support for some stuff that must be available even in the compressed form. This includes some of your resources and export names in dlls
  Dealing with bound imports

- Support for stripping out directory entries that will confuse the Windows loader since the decompression won't have happened and they will point to nothing useful, like the IAT and debug info

- Support for doing relocation fixups manually on your decompression stub since it will certainly be in a different memory location than where the linker thought it would be when it was compiled

- Dealing with differences in interpretation of the PE spec between different vendor's linkers. Borland linkers interpret aspects of the spec differently from Microsoft's so you need to be ready for that.

- Working around bugs in Microsoft code. There is an infamous one relating to OLE and the resource section. Many packers do not accommodate this and this is important for ActiveX support.

4 First Step

OK, enough of the horror stories. The first step is to get painfully familiar with the file format of executables. This is called the 'Portable Executable' format, or PE for short. I will discuss it briefly here. You will need more detail in reality. Rather than attempting to duplicate that, here are some references you will find helpful:

> The Portable Executable File Format from Top to Bottom
>
> http://mup.anticrack.de/Randy%20Kath%20-%20PE%20Format.html/]http://mup.anticrack.de/Randy%20Kath%20-%20PE%20Format.html

a good and readable discussion, but not totally accurate when it comes to the import section. Dead wrong in implying that these sections always exist -- they easily can not exist. Still, a good read.

> An In-Depth Look into the Win32 Portable Executable File Format pts 1 and 2
> //h**p://www.msdnaa.net/Resources/Display.aspx?ResID=1083]http://www.msdnaa.net/Resources/Display.aspx?ResID=1083
> //h**p://www.msdnaa.net/Resources/display.aspx?ResID=1323]http://www.msdnaa.net/Resources/display.aspx?ResID=1323

great article, weak on discussion of resource section

> Microsoft Portable Executable and Common Object File Format Specification
> //h**p://www.microsoft.com/whdc/hwdev/hardware/pecoff.mspx]http://www.microsoft.com/whdc/hwdev/hardware/pecoff.mspx

horse's mouth. Dry. Accurate.

5 Next Step

OK, after you've gotten familiar with those, we can start to write some code. I'm going to save that for the next installments (probably two). They will detail:

- Making the Unpacker Stub
  The stub has several responsibilities aside from the obvious decompression. It also has to perform duties normally done by the Windows loader.
- Making the Packer Application
  The packer application does all the hard work. This makes since when you realize the stub is supposed to do as little as possible to have a minimum impact on runtime.

I'll try to keep code examples to a minimum but there may be some reference to structure members when describing what's going on and maybe a snippet or two where code is clearer than human language. Most of the important structures can be found in WINNT.H for those who wish to read ahead.

6 Continuo

Last installment I mentioned some of the big-picture aspects of creating an exe packer. In this installment I am going to talk about a particular part of the packer, the decompression stub. This is the simpler part. In the next installment(s) I'll talk about the packer application itself. Again, this isn't going to be source for a packer, but I might do a straightforward one and publish it as an addendum to this series if folks are interested in having some working source as a starting point.

The decompression stub has several responsibilities:

- Find the packed data
- Restore data contents
- Perform relocation fixups
- Resolve all imports since the Windows loader couldn't do it
- Perform thread local storage duties since the Windows loader couldn't do it
- Boink over to the original program
- You may also have to handle being reentered if you are packing a dll

Oh, and it also has to run. So lets start with that...

7 A Stub That Runs

It's useful to remember that your decompression stub is actually a parasite onto a program that was never expecting for it to be there. As such, you should try to minimize your impact on the runtime environment in your packer. I had mentioned before that you could make a packer in Logo or Object-Oriented COBOL, and that really was only partially true. You can make the packer application that way fer sure -- and you might even be able to make the unpacker that way sometimes -- but you will really be much happier with C/C++/ASM for the stub part. I personally like C++. Anyway, it will be smaller. If you don't care about the size, still using stuff like Delphi or VB for the stub would be problematic because it hoists

in subtle stuff like TLS and runtimes, and they don't have primitives needed to thunk over to the original program. Plus it can hose COM stuff that the original app isn't expecting. So let's assume the unpacker will be in the lower-level languages I spoke of and take solace that this is pretty straightforward code, and that the packer still can be in whatever.

Since the stub is a parasite, and since it will have to be located in a spot at the original application's convenience, we will have to be relocating it dynamically in the packer application. To help with this we will make the stub with relocation records. These are usually used for dlls when they can't be loaded at their preferred address. We will make use of them when binding the stub to the original application.

If you're an avid ASM coder, many things are more straightforward since you can take care to produce position-independent code. This won't necessarily free you from all relocation concerns, however. The decompression library of choice may well not be position independent. Also, and references to global data will need to be fixed up.

8 Choice of Compressor

You can pretty much use whatever compressor library you want, so long as you build it in a way that doesn't spew out stuff via printf or other UI functions. There are plenty free compressors out there. You might want to start with something like zlib. It won't give you the best compression, but I know it works in this scenario. Also, another is UCL. This compresses better and is much smaller code-wise. It is GPL, however, and maybe you care about the licensing implications.

Check the docs to the compressor you want for configuration options and related stuff. For example, BZip2 requires BZ_NO_STDIO to be defined to have no printf stuff.

Configure the build to be compatible with the stub and compression library. For me, I disable RTTI and make sure I am linking the static runtime library, multithreaded. I optimize for size. The output should produce a static library, of course, rather than a dll, since the goal is to add no dependencies beyond the apps original ones.

Setting Up Projects -- and now for something completely different

OK, I am going to take a brief break from code and technological stuff and talk about project configuration. Normally I wouldn't since that's a personal choice, however this time I will because things I talk about later will be dependent upon some of the configuration assumptions. In real life you don't have to do it this way, but let's temporarily pretend we are and at the end of this series you'll know how you might like to do it different.

Big picture is that there will be two projects, producing two distinct executables -- the packer stub and the packer application. Their configuration will be significantly different.

We are going to do a bit of ledgerdemain with the stub project which will be explained later, but for now, configure a boiler plate project for your stub thusly:

- Produce a DLL
- Use static multithreaded runtime libraries
- Disable RTTI and exception support

If there are options for the boilerplate code it generates, make it simple, so that there is just DllMain being implemented. We're going to throw all that away anyway. Go ahead and build it as a sanity check, which should go through fine.

We're making the packer stub a DLL not because it will ultimately be a DLL -- it won't. We're doing this because we want the relocation records. You _can_ create it as an exe project and cause it to have relocation records (linker option /FIXED:no), but I find the Microsoft's linker will crash randomly in that configuration. Stick with the DLL config and you'll be OK.

Next, change the config thusly (this is for Microsoft's tools, you'll have to look up the equivalents for Borland's or gcc):

> Linker options:
>
> add any library paths your compressor lib will be needing
> /nodefaultlib don't use default libs
> /map(filename) DO generate a mapfile
> remove /debug don't generate debug info
> change /incremental:yes to /incremental:no disable incremental linking

> Compiler options:
>
> add any defines or header paths your compressor lib will be needing
> /FAcs generate listing files with source, assembly, and machine code
> /Fa(filename) specify where these listings go
> remove /GZ compiler-generated stack checks
> remove any debug options, it won't help us where we're going

> these options are probably available as checkboxes, so you won't have to manually add them.

The gist is that we are not going to have normal debug capabilites so we turn off that stuff. Instead, we will be relying on the listing of the compiler-generated assembly to see code and the linker-generated mapfile to see actual addresses. All this is interesting stuff in any project really, but it is all we have for debugging in this one.

If you build now you will should get a linker error complaining about an unresolved external symbol DllMainCRTStartup@12. This is good! If you don't get that then the default libs are coming in. The symbol is possible different for Borland stuff. Other errors probably mean something else needs to be fixed; this is the only one you should get for Microsoft's compiler.

9 Runtime dependencies

You cannot assume what runtime dependencies the original app has. Thus, you cannot make calls to funky dlls (vbrunX.dll, etc). You have no idea if they are there. You will do well to statically link your runtime library. You will do much (much) better, however, to not link any runtime libraries at all! ASM coders will take delight in this fact already, because they are hard-core, but this need not dissuade the C/C++ coders who are accustomed to malloc() strcmp() new std::vector<> or such. All this is doable. You will just have to provide your own implementation of these functions. Fortunately, this is pretty easy since you can call native functions exported by Kernel32.dll. /That/ dll is certainly present, and certainly one that is already used by the original app you are packing so feel free to use it when you like.

10 Making a Trivial C Runtime to Link Instead of the Proper One

Replacing the C Runtime might sound scary but remember we only want to implement what is necessary; this will turn out to be a small set of things. The linker will help you figure out what these are. Recall that we turned off default library searching with the /nodefault switch (or equivalent for your linker, that's for Microsoft's). If you configured as I suggested above, we've got a linker error already: DllMainCRTStartup@12 We'll fix that one first.

Discard your boiler-plate DllMain. Replace it with:

```
BOOL WINAPI _DllMainCRTStartup ( HANDLE,
DWORD, LPVOID )
{
//(program will go here)
return TRUE;
}
```

This should resolve the linker error and will be our entry point. The place our program will ultimately go is indicated by the comment. Ultimately we'll never hit the 'return TRUE' statement; it's just there to make the compiler happy, and the function signature is what it is to make the linker happy.

If you want to be more arty, you can do the following:

```
#pragma comment ( linker, "/entry:\"StubEntryPoint\"" )
void declspec ( naked ) StubEntryPoint() {
//(program will go here)
}
```

which is syntactically clearer.

This is cosmetic so don't feel bad if you find the equivalent pragmas for your compiler/linker. Also, this perverts what the compiler normally thinks about and I have seen it crash randomly. I have found when the compiler gets in a crashing mood, that putting in:

```
asm nop
```

in a couple places seems to get it back on track. Ain't that a laugh?! Whatever...

As code is added, you should periodically build. The linker will add more and more complaints like above and we will have to implement the underlying methods the compiler is emitting references to. Here's a tip: when you installed your dev tools, you may have had the option to install the source to the C Runtime. It will be helpful in some cases since you can cut and paste some parts. In particular, a function:

```
extern "C" declspec ( naked ) void _chkstk(void)
```

is sometimes emitted by the compiler quietly (if you have a large array on the stack, like for a buffer). Just cut-and-paste that one; it's funky.

FYI, I typically have to implement:

```
memcpy
memset
memcmp
malloc
free
realloc
```

```
calloc
operator new ( unsigned int nSize )
operator delete ( void* pv )
```

To get you going on what it means to do this sort of roll-your-own-C-runtime, please see the following article. It's good and will save me from repeating the infomation here. There's sample implementation as well.

> Reduce EXE and DLL Size with LIBCTINY.LIB
> http://msdn.microsoft.com/msdnmag/issues/01/01/hood/default.aspx]http://msdn.microsoft.com/msdnmag/issues/0...od/default.aspx

OK, we're now setup to do the work!

11 Unpacking Stub Responsibilities

I mentioned way back that the stub has the following duties:

- Find the packed data
- Restore data contents
- Perform relocation fixups, if needed
- Resolve all imports since the Windows loader couldn't do it
- Perform thread local storage duties since the
- Windows loader couldn't do it
- Boink over to the original program
- You may also have to handle being reentered if you are packing a dll

It's important that the stub restore the original data to it's exact original location. This is because we don't know about what references are in the original code to things like global data structures and functions pointers in things like vtables.

Recall that the format of the PE file (links to good discussions were provided in the previous installment) is organized into sections, which have a location and size. This information is stored in the section headers, which describe where the sections go in memory (relative to the load address).

To do this properly, we will be needing to know our load address. If we are a stub for an exe we can simply do a GetModuleHandle(NULL) and the returned module handle is the base load address. This won't work for a dll however. The module handle for the dll is on the stack. We can write some code to get it, or we can choose not to do the 'arty entry point' and it is referenceble as a parameter (do not attempt to reference those parameters if it is the stub for an exe unless you are fond of crashes).

My preferred technique, however, is to get the packer application to help me out. That way the same stub works for exes and dlls and in the same way. It involves a global variable, and there are going to be several of those, so let me discuss that first.

12 Packer Parameter Globals

There are going to be parameters that are computed by the packing application and that will be embedded in the stub so it can do it's work at runtime. These require a bit of special handling because the packer application needs to find these items at pack time. You could hard-code in the addresses into the packer. You would get these addresses from the mapfile generated by the linker. This is a bit tacky because you will have to double check it each time you alter the stub, which will be quite frequently while developing. Instead, I prefer to do a bit of legerdemain with structures, sections, and segments. This only needs to be done for the variables published to the packer. Regular globals you might want to have can be done as usual without concern.

First, simple stuff. I make one structure with all the important globals. Then one global instance of that structure. Thus there is only one object the packer has to locate at pack time. Let's call that structure:

```
//in a header, say GlobalExternVars.h
struct GlobalExternVars
{
//stuff goes here
};
```

Now we will do some kookiness in our main .cpp file:

```
#pragma data_seg ( ".A$A" )
declspec ( allocate(".A$A") ) extern struct
GlobalExternVars gev =
{
//initializers go here
};
#pragma data_seg ()
```

What the Hell is that? Well, it creates a special data section for our global variables. Dirty little secret about the linker is that it sorts the section names lexically, and discards the portion at the '$' and after. By naming the section '.A$A' we will be forcing the global vars structure to be at the very beginning of the data section, which will be easy for the packing application to locate. Next, we will merge some sections with the following linker options. You can put these on the link line, or you can be fancier and place them in the code with a pragma (if your tools support such). I think putting them in the pragma makes it more obvious from the code standpoint that the stuff is fragile and should be handled carefully if changes

are needed.

```
#pragma comment(linker, "/MERGE:.rdata=.A")
#pragma comment(linker, "/MERGE:.data=.A")
#pragma comment(linker, "/MERGE:.bss=.A")
```

So the global data (and don't forget your compression lib might have some too) will all be merged into one section, with the external variable structure at the very beginning. Oh, notice that I merged .bss in too. This has a subtle simplifying effect. .bss is used to hold _uninitialized_ globals. These don't normally take up file space (since they are uninitialized) but they do take up memory. The packer will have to take this in consideration when laying out the actual stub it builds. By merging it into the data section, it will take up actual file space and thus the packer won't have to worry about it. There will be very little .bss at all so don't be disturbed about it taking up space; we're talking bytes.

13 Computing the Load Address

OK, regardless of whether you have used my technique for publishing packer globals or rolled your own, let's assume that it is done. Now, the original point was that we would be needing the the base address at runtime in the stub so that we can convert Relative Virtual Addresses (RVAs) to actual Virtual Addresses (VAs). Recall the VA = RVA + base address.

My technique is to have a published global which is the RVA of the stub entry point. The packer sets this up. The stub then takes the address of the actual entry point, subtracts the RVA computed and stored by the packer, and the result is the load location of the packed executable. I store this result in a 'regular' global (which doesn't need to be part of the GlobalExternVars). I do this first thing in the main stub entry point thusly:

```
//global var
DWORD load_address = 0; //computed actual load
address for convenience

//in the stub entry point function
load_address = (DWORD) StubEntryPoint-
gev.RVA_stub_entry;
```

Note, if you did not do my entry point rename trick, you would use the name of your funtion instead, possibly _DllMainCRTStartup. This technique always works regardless of wether the appliaton is a DLL or EXE.

Once you have the load address you are all set up to decompress to the proper location.

14 Decompressing the Original Data

The compressed data is stuff attached by the packer. Like the stub, it will have stuck it somewhere. It can located most anywhere you like. A popular choice is to locate it at the _end_ of where the original data was located. Then, decompressing that data from start to finish to it's original location causes the data to be ultimately be overwritten. Fancy. This will only work of course if the compressed data is smaller than the original, but we generally hope that our compressor actually, uh, compresses, and makes things smaller.

The compressed data is located somewhere placed by the packing application. Where? Who knows. There will be needed a published external global specifying where and setup at pack time. So add a DWORD RVA_compressed_data_start; DWORD compressed_data_size;

to the GlobalExternVars struct. Transforming the RVA to the VA by adding the load_addresss previously computed will tell you where the compressed data is located at runtime.

The specific format of your compressed data is completely up to you. Since essentially we will be restoring data to original locations, which are chunks (the sections of the original PE file), the simple stream format of:

```
struct original_directory_information
dword section_count
section 1 header
{
dword RVA_location
dword size
}
(section 1 compressed data)
...
```

The original_directory_information is the stuff in the DataDirectory member of the IMAGE_OPTIONAL_HEADER of the PE headers of the original app. The packer will have changed these values to be suitable for the stub, so it will need to stick the original in the compressed stream so we can get to those values at runtime. This will suffice for the stream. Feel free to add whatever you might like to it as well. The decompression routine pseudo-code is:

```
struct section_header {
DWORD RVA_location;
DWORD size;
};


//'regular' non-published global
IMAGE_DATA_DIRECTORY origdirinfo
IMAGE_NUMBEROF_DIRECTORY_ENTRIES;
```

```
void decompress_original_data() {
void* pvCompData = (void*)
( gev.RVA_compressed_data_start + load_address );
initialize_compressor ( pvCompData,
gev.compressed_data_size;);

decompress_data ( &origdirinfo, sizeof(origdirinfo) );

int section_count;
decompress_data ( &section_count, sizeof(section_count) )
;

for ( int i = 0; i < section_count; ++i ) {
section_header hdr;
decompress_data ( &hdr, sizeof(hdr) );
void* pvOrigLoc = (void*) ( hdr.RVA_location +
load_address );
decompress_data ( pvOrigLoc, hdr.size );
}

cleanup_compressor();
}
```

This will be called in the main entry point of the stub right after computing the actual load address.

That's it! What could be easier? Well, notice that we're using a stream model for our compressor. Most compression libraries come pretty close to implementing that but you have to do ever so slightly more to make it that simple. I wrap up my compressors in a class so that they all implement the above interface to make things simple like above. Swaping out compressors then just means making a new adaptor class. The rest of the stub need not be touched to put in different compressors/encryptors.

Now that all the original data is decompressed into it's original location, we have to do stuff that the Windows loader normally does. This includes relocation fixups, imports lookup, and TLS initialization/thunking.

15 Performing Relocation Fixups

This is really only necessary for packed DLLs since EXEs are supposed to be always loaded at their preferred base address. In fact, relocation records are usually stripped from EXEs so there's nothing to process.

Details of the relocation record format are sufficiently detailed in the articles reference in the first installment. For us to process them we:

- compute an offset of the preferred base address and the actual load address
- find the relocation records from the original directory information we just decompressed
- whiz through the records getting the DWORD at the address they indicate and add the offset

Pretty straightforward. The format of the relocation records is a little bit odd and is structured the way it is presumably for size considerations. The records are organized as a series of chunks of records, one chunk per page. The records in the chunk reference an offset into the page. Additionally, for padding consideration there are records that are essentially no-ops and should be ignored. Pseudo-code follows:

```
void perform_relocations () {
//see if no relocation records
if
( origdirinfoIMAGE_DIRECTORY_ENTRY_BASERELO
C.VirtualAddress == 0 )
return;

//compute offset
IMAGE_DOS_HEADER* dos_header =
(IMAGE_DOS_HEADER*) load_address;
IMAGE_NT_HEADERS32* nt_hdr =
(IMAGE_NT_HEADERS32*)
&((unsigned char*)load_address)dos_header->e_lfanew;
DWORD reloc_offset = load_address - nt_hdr-
>OptionalHeader.ImageBase;

//if we're where we want to be, nothing further to do
if ( reloc_offset == 0 )
return;

//gotta do it, compute the start
IMAGE_BASE_RELOCATION* ibr_current =
(IMAGE_BASE_RELOCATION*)
(origdirinfoIMAGE_DIRECTORY_ENTRY_BASERELOC
.VirtualAddress + load_address );

//compute the end
IMAGE_BASE_RELOCATION* ibr_end =
(IMAGE_BASE_RELOCATION*)
&((unsigned
char*)ibr_current)origdirinfo[IMAGE_DIRECTORY_EN
TRY_BASERELOC.Size];

//loop through the chunks
while ( ibr_current < ibr_end && ibr_current-
>VirtualAddress ) {
DWORD RVA_page = ibr_current->VirtualAddress;
int count_reloc = ( ibr_current->SizeOfBlock -
IMAGE_SIZEOF_BASE_RELOCATION ) /
sizeof(WORD);
WORD* awRelType = (WORD*)((unsigned
char*)ibr_current +
IMAGE_SIZEOF_BASE_RELOCATION);
for ( int i = 0; i < nCountReloc; ++i ) {
WORD wType = awRelTypenIdx >> 12;
WORD wValue = awRelTypenIdx & 0x0fff;
if ( wType == IMAGE_REL_BASED_HIGHLOW ) { //do it
*((DWORD*)(RVA_page + wValue + load_address)) +=
reloc_offset;
}
```

```
ibr_current = (IMAGE_BASE_RELOCATION*)
&((unsigned char*)ibr_current)ibr_current->SizeOfBlock;
}
}
```

This is the majority of what is needed to support DLLs. There is a little bit more discussed later. Given that this is so straightforward, I'm a little surprised at the number of packers out there that do not support DLLs.

The next major thing we have to do is to resolve all the imports. This is only a little more involved that the relocation records.

16 Resolving Imports

Resolving the imports consists of walking through the Import Address Table of the original application and doing GetProcAddress to resolve the imports. This is very similar to the relocation record logic that I won't do a pseudo-code example. Details of these structures are given in the links provided in the first installment. The structures all start at:

```
origdirinfoIMAGE_DIRECTORY_ENTRY_IMPORT.Virtual
Address
```

There are a couple caveats I should mention however:

- The structures are wired together via RVA pointers. These need to have the load_address added to make a real pointer
- The pointers in the structure to strings are real pointers. These _do_not_ need the load_address added. Relocation processing will have already fixed these up.

Don't forget about importing by ordinal. You will know this is happening because the pointer to the string will have the high bit set ((ptr & 0x8000000) != 0). Borland and Microsoft linkers do different things, so you have to be prepared to get the string from either of different spots. Basically, there are two parallel arrays, the ImportNameTable which you get from:

```
IMAGE_IMPORT_MODULE_DIRECTORY.dwImportNa
meListRVA
```

and the ImportAddressTable which you get from:

```
IMAGE_IMPORT_MODULE_DIRECTORY.dwIATPortio
nRVA
```

The ImportNameTable is optional. Borland doesn't use it. If it is present, you should use it to get the name of the function and GetProcAddress() it's pointer (the IMAGE_IMPORT_MODULE_DIRECTORY.dwModuleNameRVA has the name of the dll you will need to LoadLibrary() on). Once you get the address, you stick it in the parallel location in the ImportAddressTable array. You do this for each member.

In the case when the ImportNameTable is not present, however, as with Borland's linker, you must get the address of the function name from the ImportAddressTable itself. Then you overwrite it with the function address.

It is important to use the ImportNameTable in preference to the ImportAddressTable because of a thing called 'bound executables'. If you want to test your work on a bound executable, consider that notepad.exe is bound.

After processing each DLL you may or may not wish to do a FreeLibrary. It's going to depend on how you implement your packer application. We'll discuss that in the next installment, and it relates to 'merged imports'. For now, suffice it to say that if you perform merged imports, you can call FreeLibrary, but if you do not, you must not call it. You might want to put the call in and comment it out while developing until you have merged imports implemented. Merged imports is important for properly supporting TLS that potentially exist in implicitly loaded DLLs. This leads into the final responsibility for the stub, which is handling TLS support.

17 Supporting TLS

Thread Local Storage, or TLS, is a handy programming mechanism. We don't care mostly, since we're not using it, but the original application to be packed might be using it indeed. In fact, Delphi always uses it, and so if we're going to support packing Delphi apps, we better accomodate it.

TLS fundamentally is done via API calls. In general, you allocate an 'index' which you store in a global variable. With this index you can get a DWORD value specific to each thread. Normally you use this value to store a pointer to a hunk of memory you allocate once per thread. Because people thought this was tedious, a special mechanism was created to make it easier. Consequently, you can write code like this:

```
declspec ( thread ) int tls_int_value = 0;
```

and each thread can access it's distinct instance by name like any other variable. I don't know if there is an official name for this form of TLS, so I'll call it 'simplified TLS'. This is done in cooperation of the operating system, and there are structures within the PE file that makes it happen. Those structures are contained in a chunk that

is pointed to by yet another directory entry:

```
origdirinfoIMAGE_DIRECTORY_ENTRY_TLS.VirtualA
ddress
```

The problem is that the processing of this information happens by the OS on the creation of every thread prior to execution being passed to the thread start address. This would not normally be a concern for us, except that at least one thread has been started before we can unpack the data: our thread! What we have to do is set up a fake TLS management section to capture what the OS has done before we started, then manually copy this information to the original app as our last step.

For this, I add two items to the external global packer data structure:

```
GlobalExternVars
{
//(other stuff we previously described)
IMAGE_TLS_DIRECTORY tls_original;
IMAGE_TLS_DIRECTORY tls_proxy;
};
```

The packer application will copy the original data to tls_original for our use at runtime. tls_proxy will be almost an exact copy, except two items will not be modified from the stub:

```
tls_proxy.AddressOfIndex
tls_proxy.AddressOfCallBacks
```

In the stub we will inialize the AddressOfIndex to point to a normal global DWORD variable, and we will initialize AddressOfCallBacks to point to an array of function pointers in the stub. The function pointers array is a list of things that is called whenever a new thread is created. It is intended to be used for user defined initialization of the TLS objects. Alas, no compiler I have seen has ever used them. Moreover, on the Windows 9x line, these functions are not even called. Still, we support it in case one day they are used. We point the AddressOfCallbacks to an array of two items, one pointing to a function of our implementation, and the second being NULL to indicate the end of the list.

There will be a global DWORD for the TLS slot:

```
DWORD TLS_slot_index;
```

The TLS callback function must be of the form:

```
extern "C" void NTAPI TLS_callback ( PVOID DllHandle,
DWORD Reason, PVOID Reserved );
```

also you add two global booleans indicating that it is safe to invoke the original callbacks, and to indicated that there is a deferred call. Initialize these globals thusly:

```
bool safe_to_callback_tls = false;
bool delayed_tls_callback = false;
```

and provide some auxilliary globals to hold data that is delayed:

```
PVOID TLS_dll_handle = NULL;
DWORD TLS_reason = 0;
PVOID TLS_reserved = NULL;
```

the thunk implementation proceeds as such:

```
extern "C" void NTAPI TLS_callback ( PVOID DllHandle,
DWORD Reason, PVOID Reserved ) {
if ( safe_to_callback_tls ) {
PIMAGE_TLS_CALLBACK* ppfn =
g_pkrdat.m_tlsdirOrig.AddressOfCallBacks;
if ( ppfn ) {
while ( *ppfn ) {
(*ppfn) ( DllHandle, Reason, Reserved );
++ppfn;
}
}
} else {
delayed_tls_callback = true;
TLS_dll_handle = DllHandle;
TLS_reason = Reason;
TLS_reserved = Reserved;
}
}
```

This will provide a place for the OS to store the slot info, which we will later restore, and if it does call thunks then we will capture the parameters for later when we will invoke the original thunks after decompression. Again, this is all done because the OS will be doing this stuff before we have a chance to decompress. After we decompress, we pass the call straight to the original application.

We handle this last step like so:

```
void FinalizeTLSStuff() {
if
( origdirinfoIMAGE_DIRECTORY_ENTRY_TLS.Virtual
Address != 0 ) {
*gev.tls_original.AddressOfIndex = TLS_slot_index;
void* TLS_data;
asm
{
mov ecx, DWORD PTR TLS_slot_index;
mov edx, DWORD PTR fs:02ch
mov ecx, DWORD PTR edx+ecx*4
mov pvTLSData, ecx
}
int size = gev.tls_original.EndAddressOfRawData -
gev.tls_original.StartAddressOfRawData;
memcpy ( pvTLSData, (void*)
```

```
gev.tls_original.StartAddressOfRawData, size );
memset ( (void*) gev.tls_original.EndAddressOfRawData,
0,
gev.tls_original.SizeOfZeroFill );
}

safe_to_callback_tls = true;
if ( delayed_tls_callback ) {
TLSCallbackThunk ( TLS_dll_handle TLS_reason
TLS_reserved );
}
}
```

Once you have done that, it is finally safe to call over to the original program. You should have a published external global that will be set up by the packing application that specifies the original program's entry point. I will call it

```
DWORD orig_entry;
```

which will be a member of GlobalExternVars. It will be initialized to an RVA and we will fix it up to a VA by adding the load_address. This done only once on the first pass, of course.

For EXEs, the entry point will never return. For DLLs it will. Moreover for DLLs there are the original parameters which must be pushed. This brings us to the final topic, the last bit needed for DLL support.

18 Last Bit for DLL Support

EXEs go into their entry point only once, and with no parameters (remember, this is not main(), but well before that). DLLs, on the other hand enter at least twice and perhaps once per thread. Obviously, the stuff we did before (the decompression, relocs, imports, TLS) only needs to be done once. Easy enough, add a global boolean that indicates that stuff was done and set it to true after the first pass.

The slightly more tedious thing is producing a stub that works for DLLs and exes, since you will want to return the value.

What I like to do is make use of the declspec (naked) attibute I applied to the StubEntryPoint. This causes the compiler to emit no prolog and epilog code. Consequently, if we don't mess with the stack, we can do and assembly jmp to the original entry point, and the right behaviour will happen if we are an EXE or a DLL. Thusly:

```
asm jmp gevt.orig_entry;
```

And all should be running.

19 Afterthoughts on Stubs

Looking at other packers, I have seen some slightly different stub techniques. I think the most interesting is UPX, where the packer actually acts somewhat like a linker, building the stub code dynamically and including only what is necessary at pack time.

You can implement the stub in the fashion of your choosing, and you can omit features you don't think will be necessary in your particular application.

20 What's Next

OK, this was a good bit longer than I expected. Still, I wanted to communicate as much as possible the details so that others won't have had to spend as much time in the debugger as I had. Debugging a compressed exe is a major pain because the debugging info is all useless so you have to do it in assembly.

Next installment will cover the packer application, which will be much more straightforward from the standpoint of configuration, but will have much more work to do than the stub.

21 Continuo

This series is about creating exe packers for Windows 32-bit PE files.

In the previous installment I described how to create a decompression stub that would be bound to an existing executable. In this (final?) installment I'm going to describe the actual packer application, which binds the stub to an arbitrary executable and sets up parameters the stub will need at runtime. Additionally, it will perform some duties normally done by the OS loader.

The packer application will wind up being the biggest hunk of code for the project. Fortunately, it will be fairly straightforward.

22 First Things

There are some basic things to setup or consider before we get moving with the actual packer.

22.1 Project Configuration

As mentioned in the previous installment, configuration is a function of your particular design. For the sake of discussion in this article we are assuming a design where the decompression stub is produced as a dll. The binary of that dll will be incorporated into the packer application as a binary resource. None of this is strictly necessary. The stub 'dll' will never exist in the real world

as such since we are going to snip out interesting pieces. You could just as easily use a tool to spew just the interesting pieces to binary resources, or encode them as static data in a C source file. This choice is per taste and we are going to choose the resource approach. We are also going to be a command-line app. So...

Configure your project as a command-line (console) application. Create a RC file and include a resource that is the stub 'dll' produced by your previous project. That's really it for configuration. I'm sure that will be a welcome simplification after having set the stub project!

22.2 Utility Code

There are going to be some things that are simple, but very tedious, and you will probably like to produce some machinery to tend to these tasks.

One such task relates to translating addresses. We have to do this in a couple places for different reasons, so you might consider making some sort of general purpose address translator. It will need to handle several distinct ranges of addresses being mapped independently to other ranges. In practical terms, there won't be a huge number of range mappings (like about 5), so if you want to just keep a list of range mappings and do a linear search no one will chastise you.

Another tedious thing (I find) is reading little bits and pieces from the original executable file. This is particularly true when navigating a network of objects since you have to run along pointer paths. To make this much more bearable I use a memory-mapped file for the original executable. Read-only access is fine since we won't be altering the original (BTW, if for some reason you do want to write to the mapped image, but not disrupt your original file, remember you can map it copy-on-write. I've done this for some protectors.) I don't use this approach for the output file, however, because most of that will be sequential write.

Lastly, I would like to reiterate that the pointers in the executable are RVA's. This means you will need to do _two_ things to transform them to real pointers. First, if you've mapped the image to an address, you will need to add that base address. The stub 'dll' compiled in as a resource will be accessed through a memory address once we LockResource() on it. That address is the base address. Now, that's all you have to do on a running module (i.e. one the OS loader mapped in), but that's not all we have to do. The second thing we have to do is consider the file and section alignment of the executable (do _not_ assume they are they same). The net result of this is that there will need to be an adjustment on a per-section basis to the resultant pointer. Again, this is not necessary for a module loaded by the OS loader into memory since it has mapped the sections appropriately.

So, I would further suggest creating a utility class that incorporates the address translator mentioned earlier (along with logic to initialize it) that can provide translated access from RVA to physical pointer for regions within a PE file. Stick in an RVA, get out a physical memory pointer. We can use this device for both the memory-mapped original, and also for the resource-loaded stub. You don't have to do this but it will make your life easier. This is a plus because it's already going to get a little harder as it is wink.gif. You may wish to throw in a couple other convenient PE-specific items, like pointers to the image headers. We'll be using various fields in these headers at several points throughout the packing process.

One other thing that will make you happier in the long run is to produce some sort of wrapper for your compression library of choice. In doing so you can both simplify use of the library and also be able to swap out a different compressor should you choose. For example:

```
class Compressor {
public:
Compressor ( HANDLE hFile ); //create; write to given
file at current file pos
void InsertData ( const void *pv, int size ); //stick some
uncompressed data in
void Finish(); //finish any pending writes
DWORD CompressedCount(); //count of output
(compressed) data
};
```

This sort of interface I have found to be suitable for all compression libraries I have considered, though of course I wouldn't use it for things other than this exe packer.

Other than that you might like to make a general-purpose resource tree walker, but we'll discuss that later in the implementation. Making this part generic is mostly useful if you wish to reuse it in other projects.

With that being said, we are ready to move onto the...

23 Basic Tasks

Here are the fundamental things the packer will need to do.

- Determine Size of original
- Setup new section(s); modify originals
- Create and add stub outside this region
- Preserve export info
- Fixup TLS stuff
- Relocate the Relocations

- Compress and stow original data
- Process the resource directory
- write out the results

and a couple minor fixups like changing the entry point and some of the directory entries.

24 Details

Here are the details of each of these tasks.

24.1 Determine Size of Original

This is the easiest task as it is indicated in the PE header of the original. It is located at:

```
IMAGE_NT_HEADERS::OptionalHeader.SizeOfImage
```

This is important, because this determines the start of where we will bind our stub. After we bind our stub we will update this value to include the stub's additional size.

24.2 Setup New Section(s); Modify Originals

Sections, which are basically areas of the file that the loader allocates and possible memory-maps to regions in the running process' address space, are described in the PE header. They can take up zero disk space, when tells the loader to allocate the memory, but not to map part of the file in (e.g. this is routinely done for sections like uninitialized data.)

Since we are going to pack the application, and since we will have to initialize it ourselves (i.e. the loader can't do it for us) we will need to modify the existing section headers. In particular We will need to modify the 'characteristics' of the section to convert them all to writeable since we will be writing when decompressing (IMAGE_SECTION_HEADER::Characteristics). Also, we need to modify the size of compressed sections to 0 (IMAGE_SECTION_HEADER::SizeOfRawData). The PointerToRawData need not be modified, but I usually set it to 0 anyway.

It's worth noting that the section names have not meaning whatsoever (with one exception I shall note), and you can change them at-will. They are purely mnemonic. The important bits of data that may be broken into sections (or combined with existing sections) are all located through the 'directories' located at IMAGE_NT_HEADERS::OptionalHeader.DataDirectory.

Now for the exception: due to a defect in the internal implementation of OLE automation, one section, the resource section, must preserve its name. The defective implementation finds the resources via section name (.rsrc) rather than looking up in the directory (at IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryIMAGE_DIRECTORY_ENTRY_RESOURCE.VirtualAddress). The result of this is that you take care in handling this one. More details on that when we discuss resources.

There are a variety of choices in determining how you want the new sections to be laid out. Some packers keep all the original ones, making them 0 size, and adding the new sections. Other packers consolidate the original sections into one uninitialized section and append the new ones. This is largely a matter of personal choice.

For example UPX consolidates the sections and splits the consolidation in two. The lower-addressed part is named UPX0 and is uninitialized. The higher-addressed part is named UPX1 and contains the compressed data and the stub. The reasoning behind this choice is apparently that it has less runtime impact since the compressed data will ultimately overwrite itself. ASPack on the other hand leaves the original sections in place and adds two new ones, one for the stub and one for the stub data (compressed data presumably). Many packers allow you to give arbitrary names to the sections as a minor method of hiding what packer was used. Amusingly, ASPack allows you to do this for the stub code section (by default .aspack) but the data section has a fixed name (.adata). Go figure.

If you're making a new packer then for development purposes you may wish to simply keep all the sections and append your new one. Later you can tweak the section handling stuff since its trivial.

In our example, we're going to stick all the stub code and compressed data into one section which we will append to the end. If you're going to do some resource preservation (like to preserve icons and stuff needed for COM.OLE.ActiveX like registration stuff) there will be yet another section added after the stub (because of the Microsoft OLE bug).

I keep a list of the section headers, keeping a reference to the stub section on hand. The original sections I setup once and forget about. The stub section will be manipulated as we go since we really don't know how big it's going to be yet until we compress the data. I setup the name, characteristics and virtual address now since we know them. I use the following for characteristics

```
IMAGE_SCN_CNT_INITIALIZED_DATA|
IMAGE_SCN_CNT_CODE|
IMAGE_SCN_MEM_EXECUTE|
IMAGE_SCN_MEM_READ|
IMAGE_SCN_MEM_WRITE
```

which pretty much turns on all access. The IMAGE_SECTION_HEADER::VirtualAddress I initialize to:

```
IMAGE_NT_HEADERS::ImageBase +
IMAGE_NT_HEADERS::OptionalHeader.SizeOfImage
```

which sticks it at the end of the original exe's PE-mapped address space. (We'll have to fixup SizeOfImage for the result later, when we know how be the stub and data is).

This will leave until later the need to fix up the fields: VirtualSize - how big in memory PointerToRawData - where in file SizeOfRawData - how much in file is mapped into memory.

24.3 Create and Add Stub Outside this Region

OK, this is the twistiest part, mostly because there is a lot of pointers to follow and also a bit of translation of those pointers (in one case we will have to translate the pointers twice!). Hope you implemented some of that utility code we mentioned! You can take solace in the fact that the big picture is quite simple, and so all the complexity is in managing the indirection.

The big picture of stub creation is appending chunks of memory, then whizzing through that memory and fixing up pointers. We have to do the fixups because when we built the stub project the linker calculated where items were. We are going to be moving stuff to a place appropriate for the particular exe we are packing, so the original calculations will be invalid and must be corrected.

The way we described the decompression stub project in installment 2 of this series possibly made you want to wretch because of the funky linker options and various pragmas. Well, all that was done to make this operation more manageable. If you set up the project as described your resulting stub 'dll' (which I shall call 'stub.dll' for convenience) should have four sections. This was achieved via the various /MERGE options for the linker. The names are not really important, but we want four to make it easier to find the important stuff. You can use DUMPBIN.EXE to see what the sections are. I am going to assume that you have four sections named:

```
.text - code section
.A - specially organized data section in stub
.idata - import section
.reloc - relocation data
```

Ultimately we will merge all these together, but we want them separate in the original stub 'dll' for special reasons.

The data section should be distinct because we took pains to put the public stub data (the stuff we will be fixing-up for the stub to use at runtime) located at the very beginning. Having it broken out in the source stub 'dll' makes it easy to find this important area.

The .idata section usually can be part of the data section, but we want it separate because we are going to completely regenerate it. Having it in a separate section makes it easier to throw out the original (after processing) and replace it with our new one.

The .reloc section contains the relocation data. Similar to .idata, we are going to process the original and replace it with the new contents.

The .text section is not special in itself. It is just what is left over from not being in the other sections.

If you don't have three sections containing the above information you may want to revisit your stub configurations. Again, the particular name is not important, just the contents.

24.4 Starting to Process the Original Stub

The stub section will be small, so I build it up completely in memory before transferring it to the final disk image. You can use whatever technique for managing the memory you like, but you might have to do some reallocations as the section grows. If you use C++ you can free yourself from this minor chore by using a std::vector<unsigned char> as your buffer. That way you can append with push_back() or insert() or resize() as you chose.

We mentioned earlier in Utility Code how you could create a class for handling the details of accessing portions of PE files. Both the original application and the stub 'dll' are PE files and you can use an instance of this utility class for each of these.

We are going to create the new stub section by appending the code (.text in the cited example), the data (.A in the example), then imports (.idata) and finally the relocations (.reloc). As mentioned earlier, since these sections will be in a location than what the linker thought, we must fixup internal pointers to reference the new location. Happily, the linker provided what is essentially a list of 32-bit values that are virtual addresses (_not_ RVA's) of all such pointers. We just add a delta to that value that we compute. To compute that delta you will need to know where the section it originally pointed to came from, and where it moved to.

You then add this to the 32-bit value located at the place specified in the relocation record. Tedious? Yes.

To simplify this relocation task I suggest using the Address Translator utility class mentioned earlier. Then you just stick your address in and get back what it translates to. To use this, however, it must be setup. You setup the translator as you append your sections. Here is some pseudo-code of how to do it for this example packer:

Merging the code and data sections given:

- buffer of bytes for destination stub section (empty)
- translator (empty)
- original stub 'dll' w/ mechanism to access sections by RVA
- list of sections in stub 'dll'
- RVA of start of stub section in destination exe (computed earlier)
- preferred load address of destination exe

then do the following:

- for .text section in stub 'dll'
- append all .text section to buffer
- add entry in translator translating from (original stub .text RVA start, original stub .text length) to (RVA dest + buffer.size(), original stub .text length)
- resize buffer as needed to align on 32-bit boundary
- remember current size of buffer; this will be the index to public data
- for .A (data) section in stub 'dll'
  append all .A section to buffer
- add entry in translator translating from (original stub .A RVA start,
  original stub .A length) to (RVA dest + buffer.size(), original stub .A length)
- resize buffer as needed to align on 32-bit boundary

OK, at this point we have merged our code and data. We also have an index that corresponds to where in the buffer the packer public data is located. Keep that as an index rather than pointer because as we grow the buffer, the pointer will become invalid whereas the index will not. We also have set up the first two section entries in the translator that will allow us to transform (stub) original pointers into (stub) destination pointers.

You can see the process is pretty much the same for the code and data portions of the stub. Really, it can also be the same for the import section. That is, unless you want to support TLS in DLLs. This is an obscure feature (common in EXEs though) with some subtle problems. The problems are subtle enough that even Microsoft advises against using it, and I have never seen it done in production code.

The problem is that the OS loader allocates TLS at load time and stuffs pointers in appropriate places. However, this is a one-shot opportunity and it does not perform this action if a DLL is loaded later, like when the application calls LoadLibrary() and such. Consequently folks are cautioned against using it in a DLL unless you absolutely, positively, know the DLL will only be loaded implicitly, not explicitly.

Well, guess what? Unless we take pains to change affairs, _all_ the original applications DLLs will be loaded explicitly (by the stub) and thus TLS in the DLLs will fail. We can change the affairs by manipulating the imports section to load the application's original DLLs. We do this by adding bogus import structures that makes it look like the stub is going to use all the dlls the original application did.

24.5 Merging Imports Data

If you're developing a new packer, I advise initially doing the straight-forward append of the imports like you did for the code and data for starters. This will work for every real application I have ever seen. After your packer is working, then you might consider adding import merging to support the non-existent-but-possible TLS-in-a-DLL clients.

I'm going to hand-wave through this because it's so excruciatingly boring and virtually never needs to be done. I will, however, tell you what you need to do and you can sift through the headers. If you get the rest of your packer working, performing this task will involve no new technology -- just more pointers, translations, and appends. Briefly, to do this you must:

- Go through the stub's imports; collect this information
- Go through the application's imports; merge this information (selectively)
- synthesize a new import section
- append it with limited fixups

Going through the stub's imports we only really care about the module name. This list of names will form a 'stop list' which will inhibit merging the original application stuff. No need to force an import of a module that is already coming in, and who wants to fixup all the pointers anyway.

Going through the application's imports, we ignore modules that come in through the stub. For all others we

arbitrarily select an import (I usually just choose the first) and create a new import descriptor, Import Name Table, Import Address Table, and strings for such. Your address translator will be of invaluable help in keeping track of where all the individual descriptors moved. The major issue is that you will have to insert data _into the middle_ of the original stub's import table. This comes from the extra import module descriptors for the bogus imports. The result is all the pointers from the stub's original descriptors become skewed by the amount additional descriptors. If you stuff in two translation records for each half you will be OK.

Regardless of whether or not you do the import merging or simple append, you must still perform a special relocation pass on the imports data. The reason is that the pointers in the import section do not have relocation records! These pointers are RVAs, and thus relative and thus don't need to be fixed up at load time. Unfortunately, the thing to which they were originally relative has changed, and so we must fix them up. It's pretty straightforward.

Fixing up the import's RVAs means whizzing through the structures, using the translator to get the translated address, and saving back the result.

The structure of the imports section is adequately described in the articles I referenced in installment 1, and I refer you there for details, however there are a couple items I would like to point out:

I have never found the declaration of the Import Module Directory structure in the headers. If anyone finds the 'official' declaration I would like to know its name and location. Anyway, it's a simple struct, and here is the hand-crafted version I use:

```
struct IMP_MOD_DIR {
DWORD dwINTRVA; /*name table; may not exist*/
DWORD dwTimeDateStamp; /*for bound exes, ignore*/
DWORD dwForwarderChainRVA; /*for bound exes,
ignore*/
DWORD dwModNameRVA; /*name of dll*/
DWORD dwIATRVA; /*import address table, must exist*/
};
```

The import section consists of an array of these terminated by an empty one.

- The INT contains a list of the ordinal, or name and hint, of an imported symbol.
- The INT may not exist. Borland shuns the INT apparently, whereas Microsoft embraces
- The IAT, for an unbound exe, contains the same information as the INT for an unbound exe. For Borland (which shuns the INT) the IAT must contain this information. The net effect is that if the INT exists you must process it and copy the result to the parallel item in the IAT, or you must process the IAT only.

Some of the items are an ordinal, which means you do nothing since it is not a pointer. Don't forget to check for this.

After you have appended the Import section (either the easy way or the hard) and fixed up the pointers, set the:

```
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_IMPORT.VirtualAddress
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_IMPORT.Size
```

to refer to the newly appended area. Go ahead and align up the size of the buffer to a DWORD boundary for the next set of appends. Now we are ready to move on to exports.

24.6 Exports

If you're just packing exe's (and not dlls) you won't have to worry about this since exe don't typically export anything. On the other hand, if you _do_ intend to pack dlls, you will definitely have to deal with it. The exports section needs to be available even before the stub has a chance to decompress the original data.

This would be a straightforward append except we have to fixup RVAs, so we have to traverse the structures anyway. Fortunately, this is much simpler than what we (potentially) did for import merging.

There is one root structure -- IMAGE_EXPORT_DIRECTORY -- which is indicated in the directory of the original exe at:

```
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_EXPORT.VirtualAddress
```

After copying that structure over we will need to fixup the three members:

```
AddressOfFunctions
AddressOfNames
AddressOfNameOrdinals
```

to reflect the RVAs of where they will be copied. Append them verbatim over from the original, one after the other, following the IMAGE_EXPORT_DIRECTORY structure. The contents are largely OK as-is except for AddressOfNames and some special cases of AddressOfFunctions.

First, we will need to travel across the original application's AddressOfNames array, appending the

name over to the destination and setting the corresponding entry in the destination's copy of AddressOfNames to refer to this copy. This is straightforward.

Second, we will need to do something a bit odd. We will travel across the original AddressOfFunctions array and look for pointers (that are RVAs) that are within the export section. What is this for? Forwarded functions! Wack! Anyway, these are not addresses of exported objects (functions, data) but are strings that must be copied. In this special case, do like we did for the AddressOfNames array and copy the string and set the pointer to point to that copy.

Setup:

```
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_EXPORT.VirtualAddress
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_EXPORT.Size
```

to where we stuck it and how big it became. Finally align up the buffer to a DWORD boundary for further appends, and you're done with this part.

FYI, we are 50% through our to-do list. And we haven't compressed any data yet! It's all downhill from here...

24.7 Do Stub Fixups and Relocating the Relocations

At this point most of the stub's stuff has been built up and we can fixup it's pointers to reflect the fact that we have extracted and moved it's original components. This task is very similar to the relocation fixups performed in the stub. The difference is in computing the delta to apply.

In normal relocation, like what the stub performs, there is only one delta. This is because the image as a whole moves, and all items are relocated by the same amount. In our case, different sections have moved differently, and thus each item must be treated as having it's own delta.

The delta computation in this case is computed as the change between the RVA of the original item to be fixed up (RVAFixupOrig) and the RVA of the item after it has been moved (RVAFixupDest). The item at RVAFixupDest must then be adjusted by this delta.

Since this translated RVADest is a relocated relocation, I save it into an array of DWORDs for the next step. This saves me from going through the relocation structure twice.

After having performed stub relocations we can decide whether we need to make the resultant executable relocatable. That decision should be made on the basis of whether the original is relocatable. There are two ways to tell this; the presence of a relocation directory entry is a good one. There is also a characteristics bit that indicates that relocation records have been stripped. Suffice it to say that if the original is not relocatable, then we don't need to make the result relocatable. If it is relocatable we need to create a relocation section for the stub. The stub will handle doing the application at runtime.

To create a relocation section for the stub we first sort the array of fixed-up relocation addresses we created while doing stub fixups. The sorting is needed to handle the quirky format of the relocation section.

Recall from installment 2 that the relocation records are stored in chunks, one chunk per page, and as 16-bit records that are essentially offsets into the page. I refer you to installment 2 for details, and to the references in installment 1. Suffice it to say, we travel along our now-sorted array, emitting chunk headers whenever a page change is detected and emitting 16-bit records otherwise. A page is on a 4096 boundary for 32-bit PE files so you can AND the address with 0xfffff000 to find it's page value, and you can AND the address with 0x00000fff to find it's offset for the relocation record. Also take care that when you detect a page change, you will possibly need to pad to a 32-bit boundary by adding a no-op relocation record (IMAGE_REL_BASED_ABSOLUTE).

After processing all records set the

```
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_BASERELOC.VirtualAdd
ress
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_BASERELOC.Size
```

to reflect this new chunk that we added. We should already be aligned to a 32-bit boundary.

24.8 Setup for TLS Stuff

If the original application used TLS we need to set some things up so that the stub can help out. This is fairly straightforward. Especially if there is none!

TLS information is communicated to the stub through the public data. Way back, when we were appending the data section took note of the index that starts the data section. Also, since when we build the stub to have that structure at the beginning, now we can cast the address of the buffer offset by the index to a pointer to the public structure. Again, we can't stow this pointer since

whenever we append to the buffer we risk reallocating memory, but we can recompute the pointer as needed from the index between appends.

Anyway, if there is no TLS, as evidenced by:

```
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_TLS.VirtualAddress
```

being 0, then we can simply clear out the copy of the tls directory in the public data (we called it tls_original in installment 2).

If there _is_ TLS, then we copy the original TLS directory structure to the tls_original in the public data, and copy over a few items to the tls_proxy:

```
SizeOfZeroFill
Characteristics
StartAddressOfRawData
EndAddressOfRawData
```

Note, the addresses do not need to be translated (shock-of-shocks) because they reference data in the original application, which we have not moved. The stub only accesses that data _after_ it decompresses it.

Setup:

```
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_TLS.VirtualAddress
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_TLS.Size
```

to refer to the tls_proxy structure. You compute the VirtualAddress with something like:

```
Stub Section RVA + dwIdxStubPublicData + offsetof
( GlobalExternVars, tls_original )
```

Nothing was appended here, no need to align up the buffer.

24.9 Compressing the Original Data

Finally! We compress data! There are many compression libraries to choose from, take your pick so long as you can use in the decompression stub. Recall that means operating with a minimal C runtime (which we produced ourselves). The old standby of zlib works just fine for this purpose, but don't expect spectacular compression.

You may also choose to implement a dummy compressor that does no compression at all. This is useful during development in order to isolate problems. Not useful otherwise.

OK, assuming you have implemented the wrapper interface I suggested in Utility Code, above, we are ready to do some compressing! Well almost. The compression of the original data could be large, so I prefer not to do it to memory and rather directly compress to the output file (ergo the HANDLE constructor argument in the Compressor class). So we must compute the file position of where this data goes.

We zeroed the size of the original PE sections, so the first real one is our new stub section. We need to compute the file offset to this new section (PointerToRawData).

You should make a copy of the original IMAGE_NT_HEADERS if you haven't already. We will manipulate it to reflect our output. Let's call it nthdrDest and initialize it to the original exe's values. Then calculate:

```
nthdrDest .FileHeader.NumberOfSections = (new section
count)
int nSectionHeadersPos =
IMAGE_DOS_HEADER::e_lfanew +
sizeof(IMAGE_NT_HEADERS);
int nFirstSectionPos = nSectionHeadersPos +
(new section count) *
sizeof(IMAGE_SECTION_HEADER);
```

Align up the nFirstSectionPos according to IMAGE_DOS_HEADER::OptionalHeader.FileAlignment

This is the PointerToRawData for our stub data. Stick that value into the section information that we created way back in the beginning (it was the last item in the list).

Do a seek to this position + the sizeof the buffer we have been building up. The net effect of this is to cause the compressed data to be appended to the stub section without having to stow it in memory.

Instantiate the compressor on the file handle (now properly positioned).

As we mentioned in installment 2, the exact format of the data stream is a matter of design. I had made a suggestion of using:

```
struct original_directory_information
dword section_count
section 1 header
{
dword RVA_location
dword size
```

```
}
(section 1 compressed data)
...
```

if we were to use that, then we would invoke the following using the _original_ exe's NT headers:

```
InsertData
( IMAGE_NT_HEADERS::OptionalHeader.DataDirectory
,
sizeof(IMAGE_NT_HEADERS::OptionalHeader.DataDire
ctory) );
DWORD dwSectionCount =
IMAGE_NT_HEADERS::FileHeader.NumberOfSections;
InsertData ( &dwSectionCount, sizeof(dwSectionCount) )
;
for each section IMAGE_SECTION_HEADER
InsertData ( &
IMAGE_SECTION_HEADER::VirtualAddress,
sizeof(IMAGE_SECTION_HEADER::VirtualAddress) );
InsertData ( &
IMAGE_SECTION_HEADER::SizeOfRawData,
sizeof(IMAGE_SECTION_HEADER::SizeOfRawData) );
InsertData ( (actual pointer to original data),
IMAGE_SECTION_HEADER::SizeOfRawData );
```

In other words, we are pushing the RVA of where the data goes, the physical (uncompressed) size, and then the physical data. We do this for each section of the original.

When we are done we invoke Finish() on the compressor to flush any remaining data not written.

We get the number of actual compressed bytes with CompressedCount(). This we add to the size of the buffer we were building and store it in the SizeOfRawData field of the section header for the stub.

Finally, get a pointer to the structure containing the public data (this is why we didn't write out this until now). Set the value of the stub entry point (after translating, of course), the RVA of the start of the compressed data (which is the RVA of the stub + the size of the stub buffer) and the size of the compressed data (which we got from the compressor when done).

Then seek back to the position PointerToRawData we just computed and write out the stub buffer. Basically we just concatenated the two in reverse order.

Finished with generating and writing out the stub!

24.10 Processing the Resource Directory

Processing the resource directory is a strictly optional task. It is a bit tedious. Benefits of processing include preserving the ever-important application icon and version information so that one's experience with Explorer can be gratifying and fulfilling, but also so we can support various OLE features.

If you don't care about these things simply carry on. If you do care, then more 'fun' awaits.

The 'fun' that awaits is similar to what we did for exports earlier in that we walk a structure and optionally copy stuff over, adjusting the pointer when we do and leaving it pointing to the original data in the compressed section otherwise.

The difference is that this structure is more complex, with more objects and a more complex decision on what to keep. First let me briefly tell you what you want to keep uncompressed because that's the easy part to know and tedious part to figure out experimentally. You will want to keep uncompressed the following resources:

- first RT_ICON should be kept
- first RT_GROUP_ICON should be kept
- first RT_VERSION should be kept
- first "TYPELIB" should be kept
- all "REGISTRY" should be kept

OK, that being said, keep in mind that resources are a multi-level tree of directories. You need to keep track of at what level you are to make your comparisons in order to determine whether to keep a resource or not. Also, as a perceived convenience, all the fixed sized structures are coalesced at the beginning with variable length ones afterwards. This means all the directory structures are at the beginning, with things like string identifiers and resource data afterwards.

I do a similar thing as with the stub and build this section in memory with a managed array of bytes. Once it is constructed I write it out later.

You can walk the tree once to find where this boundary between fixed and variable sized data lays, then copy the fixed data verbatim. It's interesting to not that most of the pointers in this section are relative to the section itself, and thus do not require translation. The exception to this is the pointers to the actual resource data, which is an RVA.

Walk the tree a second time and append all the string identifiers. Adjust the pointers to these strings keeping in mind that they are _not_ RVAs, but are rather relative offsets into the resource section.

Walk the tree a third time and copy over the resource chunks for the resources types of interest described above. Keep in mind that these actually _are_ RVAs, so

you will need to add the RVA of the beginning this section. What is that? Well, it is the RVA of the last section, plus its size, aligned up to the NT_HEADERS::OptionalHeader.SectionAlignment. The resource chunks should be aligned between appends.

Setup the section header for this additional section. It _must_ have the name .rsrc. Setup the VirtualAddress of this section to the RVA we just computed. Setup the PointerToRawData in a similar manner, except use the last sections PointerToRawData + SizeOfRawData and align the result up by the value of IMAGE_NT_HEADERS::OptionalHeader.FileAlignment instead. Set the SizeOfRawData to the size of the resulting chunk, and the VirtualSize to the same. You can align these values up if you like.

Similar to what we did with the stub, seek to the PointerToRawData and write out the data in the buffer we've been building.

Finally, set:

```
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_RESOURCE.VirtualAddr
ess
IMAGE_NT_HEADERS::OptionalHeader.DataDirectoryI
MAGE_DIRECTORY_ENTRY_RESOURCE.Size
```

and we are done with that.

24.11 Dotting I's and Crossing T's

There are some details that will need to be fixed up before writing the rest of the stuff out. Mostly this has to do with the various directory entries, but let's not forget the entry point address!

The entry point is computed as the stub 'dll's entry point after being translated with the translation device I hope you created.

The image size needs to be recomputed as the last section's VirtualAddress plus its VirtualSize.

Most of the directory entries need to be copied over from the stub 'dll' after being passed through the translator. Exceptions include the Resource directory. If you processed resources you should point it to the new section you created. If you did not leave it as it was in the original. Resources will be available at runtime, but not to explorer or OLE (or ResHacker).

If you made exports/relocations, setup those entries (that was discussed earlier).

Some directory entries should definitely be zeroed out:

- IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT -- kiss it goodbye
- IMAGE_DIRECTORY_ENTRY_IAT -- expunge it
- IMAGE_DIRECTORY_ENTRY_DEBUG -- (we don't really have bugs, anyway)

Seriously, though, the first two are used by the loader and will cause crashing behaviour. Removing them harms nothing. The last one might be nice, but the debugger can't get to the data until after the application is running, which is too late.

- Writing out the Remainder
- Copy over the original DOS stub.
- Write out the modified PE header.

Position to the section header offset we computed (nSectionHeadersPos) Loop through the section headers we have been keeping on-hand and write them out. If you have a modified resource section, take care to rename the original and make the new one be named .rsrc to work around the Microsoft OLE automation bug.

Close your file.

25 Beyond Packers

I think it's useful to consider from a big picture of what a packer is, because subsets of the technology can be used for different applications. For instance, we bound new code and data to an arbitrary executable that was not designed to host it, without damaging the original program. This is like an exe binder. Discard the compression and a lot of the manipulation of directories and you can produce one. Similarly, one could retain some of the directory manipulations, like with the imports, and fashion a protector of sorts to resist reverse engineering. Other extended applications may come to mind as well.

26 conclusions

I hoped you found some useful information in this article. I enjoyed having the opportunity to write it.