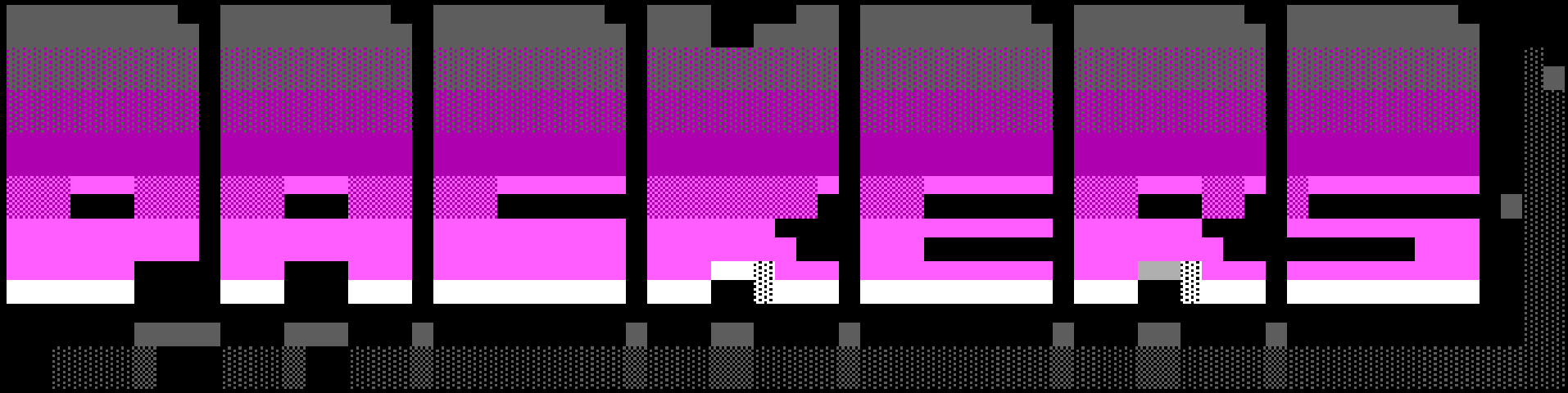
PARSERS

PACKERS

WHAT IS A PACKER?

- A *packer* is an executable compression tool that's primarily used for a few things:
  - Compression, expectedly
  - Obfuscation
  - Evasion

PACKERS

WHAT IS A PACKER?

- Fundamentally, a packer only has a few main steps:
  - **Compress** an executable image.
  - **Decompress** that image at runtime.
  - **Load** that image in accordance with the platform's executable loader.
  - **Execute** the image.

PACKERS

WHAT IS A PACKER?

- Equally simple, a packer consists of two major moving pieces:
  - The **packer**, which is responsible for producing the packed executable.
  - The **stub**, which is responsible for performing the unpacking.

PACKERS

WHAT IS A PACKER?

- This presentation aims to teach the following:
  - How to make an automated build system for a packer, including testing the packer.
  - How to parse and manipulate a Windows executable.
  - How to mimic parts of the executable loader process to load arbitrary binaries.

PACKERS

WHAT IS A PACKER?

- You will need the following things:
  - Knowledge of C++
  - A copy of Visual Studio with C++ installed.
  - A copy of CMake
    - Source code and more can be found on GitHub: https://github.com/frank2/packer-tutorial

PACKERS

USING CMAKE

- We need three main CMake projects:
  - The packer
  - The stub
  - The dummy (for testing)
- We also need a copy of zlib, a compression library

PACKERS

USING CMAKE

```
packer/
+---+ CMakeLists.txt
    + dummy/
    | |
    | +---+ CMakeLists.txt
    |     + src/
    |       |
    |       +---+ main.cpp
    |
    + stub/
    | |
    | +---+ CMakeLists.txt
    |     + src/
    |       |
    |       +---+ main.cpp
    |
```

```
+ src/
| |
| +---+ main.cpp
|
+ zlib-1.2.13/
  |
  +---+ CMakeLists.txt
      + ...
```

PACKERS

USING CMAKE

- We have a dependency chain that CMake will handle nicely:
  - *Packer* and *stub* depend on *zlib*
  - *Packer* depends on *stub*
  - *Dummy* depends on *packer*
- Our packer’s CMake project will be our example

PACKERS

USING CMAKE

- In CMakeLists.txt, we need to declare a version of CMake we want to support and the packer project itself.

```
# target a cmake version, you can target a lower version if you like
cmake_minimum_required(VERSION 3.24)

# declare our packer as a C++ project (since zlib is a C project and the compilation
# detection might get confused)
project(packer CXX)
```

PACKERS

USING CMAKE

- We also want to remove DLL runtime dependencies. This is equivalent to /MT.

```
# this line will mark our packer as MultiThreaded, instead of MultiThreadedDLL
set(CMAKE_MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>")
```

- The angle brackets are a CMake generator expression that emits “Debug” when in Debug mode and nothing otherwise. For more about it, see the README in the Git repo!

PACKERS

USING CMAKE

- Next, we need to collect our source and resource script files.

```
# this will collect header, source and resource files into convenient variables
file(GLOB_RECURSE SRC_FILES ${PROJECT_SOURCE_DIR}/src/*.cpp)
file(GLOB_RECURSE HDR_FILES ${PROJECT_SOURCE_DIR}/src/*.hpp)
file(GLOB_RECURSE RC_FILES ${PROJECT_SOURCE_DIR}/src/*.rc)

# this will give you source groups in the resulting Visual Studio project
source_group(TREE "${PROJECT_SOURCE_DIR}" PREFIX "Header Files" FILES ${HDR_FILES})
source_group(TREE "${PROJECT_SOURCE_DIR}" PREFIX "Source Files" FILES ${SRC_FILES})
source_group(TREE "${PROJECT_SOURCE_DIR}" PREFIX "Resource Files" FILES ${RC_FILES})
```

PACKERS

USING CMAKE

- Then, add our subprojects, including zlib:

```
# this will add zlib as a build target
add_subdirectory(${PROJECT_SOURCE_DIR}/zlib-1.2.13)

# this will add our stub project
add_subdirectory(${PROJECT_SOURCE_DIR}/stub)

# this will add our test dummy project
add_subdirectory(${PROJECT_SOURCE_DIR}/dummy)
```

PACKERS

USING CMAKE

- We can use CMake to generate files for us.
- Since CMake will intelligently resolve our dependency chain, we can create a resource script that always points at the freshly built stub file. This will include our stub in the packer's resources for ease of access.

```
# this will make sure our stub data will be included in the resources of our packer
# despite where it may reside in cmake's build system
file(GENERATE OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/$<CONFIG>/stub.hpp"
  CONTENT "#pragma once\n#define IDB_STUB 1000\n")
file(GENERATE OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/$<CONFIG>/stub.rc"
  CONTENT "#include <winresrc.h>\n#include \"stub.hpp\"\nIDB_STUB STUB \"$<TARGET_FILE:stub>\"\n")
```

PACKERS

USING CMAKE

- Next we declare our packer executable.

```
# this will create our packer executable
add_executable(packer ${HDR_FILES} ${SRC_FILES})
target_sources(packer PRIVATE ${RC_FILES} "${CMAKE_CURRENT_BINARY_DIR}/$<CONFIG>/stub.rc")
```

- And add our 3rd-party dependencies.

```
# this will link zlib to our packer
target_link_libraries(packer zlibstatic)
```

PACKERS

USING CMAKE

- We have to make our binaries aware of some imports.
  - For the packer, we're using generated files to eventually be included in the project. So does zlib!
  - For the stub, we're resolving zlib includes in this CMake file due to a requirement that a dependency (zlib, in this case) be in the same directory where it's included. zlib is required by both projects, so it resides *outside* the stub directory.

PACKERS

USING CMAKE

```
# zlib, as part of its build step, drops a config header in the build directory.
# we do this too, so make sure to include everything for the build!
target_include_directories(packer PUBLIC
  "${PROJECT_SOURCE_DIR}/src"
  "${CMAKE_CURRENT_BINARY_DIR}/$<CONFIG>"
  "${PROJECT_SOURCE_DIR}/zlib-1.2.13"
  "${CMAKE_CURRENT_BINARY_DIR}/zlib-1.2.13"
)

# also set the includes for the stub from here.
# we can't set this in the stub CMake file because CMake requires includes to be in the same
# directory as the build target. for this file, our build target is packer, so this sets
# up includes relative to the packer executable.
target_include_directories(stub PUBLIC
  "${PROJECT_SOURCE_DIR}/zlib-1.2.13"
  "${CMAKE_CURRENT_BINARY_DIR}/zlib-1.2.13"
)
```

PACKERS

USING CMAKE

- Finally, we declare our dependencies:

```
# this will add our stub as a dependency and our dummy as being dependent on the packer.
add_dependencies(packer stub)
add_dependencies(dummy packer)
```

- And enable testing of the packer and the dummy:

```
# enable testing to verify our packer works
enable_testing()
add_test(NAME test_pack
  COMMAND "$<TARGET_FILE:packer>"
  "$<TARGET_FILE:dummy>")
```

```
add_test(NAME test_unpack
  COMMAND "packed.exe")
```

PACKERS

USING CMAKE

- The stub and dummy CMake files should be easy to understand now, see the GitHub repo for those files.
- We now have a CMake project that does the following:
  - Manage our various dependencies and build them in the right order
  - Build the stub binary and add it into our packer binary automatically
  - Unit tests that test our packer on a binary and verifies its unpack routine functions

PACKERS

USING CMAKE

- We can see how all this works starting at the root of the repository:

```
$ mkdir build
$ cd build
$ cmake ../
```

```
$ cmake --build ./ --config Release

$ ctest -C Release ./
```

```
$ ./packed.exe
I'm just a little guy!
```

PACKERS

PACKING

- We now have to add an arbitrary executable to our stub binary. What the stub does right now isn't relevant-- we just need to worry about the vectors by which the binary eventually gets delivered into the stub.
- For the packer, our chosen vector is a feature of Windows executables called the *resource directory*. This contains arbitrary file data, such as icons and bitmaps, for use within an executable.

PACKERS

PACKING

- Accessing resources is simple:
  - Find the resource handle, given an ID and a type string. In our case, IDB_STUB and the string “STUB”.
  - Get the resource size and load the given resource handle.
  - Lock the byte pointer to the resource data.

PACKERS

PACKING

```
auto resource = FindResourceA(nullptr, name, type);

if (resource == nullptr) {
   std::cerr << "Error: couldn't find resource." << std::endl;
   ExitProcess(6);
}

auto rsrc_size = SizeofResource(GetModuleHandleA(nullptr), resource);
auto handle = LoadResource(nullptr, resource);

if (handle == nullptr) {
   std::cerr << "Error: couldn't load resource." << std::endl;
   ExitProcess(7);
}

auto byte_buffer = reinterpret_cast<std::uint8_t *>(LockResource(handle));
```

PACKERS

PACKING

- A Windows executable starts with what's called the DOS header, or how it's referred to in the Windows API, IMAGE_DOS_HEADER.
- It's a large structure, but we're only concerned with two members: e_magic and e_lfanew
  - e_magic is the magic value that signifies the DOS header, which is 0x5A4D or "MZ" in ASCII.
  - e_lfanew is the offset into the image that contains the NT headers, which contains most of the metadata needed for our executable

PACKERS

PACKING

```
auto dos_header = reinterpret_cast<const IMAGE_DOS_HEADER *>(target.data());

// IMAGE_DOS_SIGNATURE is 0x5A4D (for "MZ")
if (dos_header->e_magic != IMAGE_DOS_SIGNATURE)
{
   std::cerr << "Error: target image has no valid DOS header." << std::endl;
   ExitProcess(3);
}
```

PACKERS

PACKING

```
auto nt_header = reinterpret_cast<const IMAGE_NT_HEADERS *>(target.data() + dos_header->e_lfanew);

// IMAGE_NT_SIGNATURE is 0x4550 (for "PE")
if (nt_header->Signature != IMAGE_NT_SIGNATURE)
{
   std::cerr << "Error: target image has no valid NT header." << std::endl;
   ExitProcess(4);
}
```

PACKERS

USING CMAKE

- For our example, we're only supporting 64-bit executables:

```
// IMAGE_NT_OPTIONAL_HDR64_MAGIC is 0x020B
if (nt_header->OptionalHeader.Magic != IMAGE_NT_OPTIONAL_HDR64_MAGIC)
{
   std::cerr << "Error: only 64-bit executables are supported for this example!" << std::endl;
   ExitProcess(5);
}
```

- The NT headers are rather large and detailed, so we'll just be covering a minimum amount of variables needed for our packer.

PACKERS

PACKING

- After the headers, a PE is organized into what are called *sections*.
- These sections contain program data such as executable code as well as plain old data.
- We’re going to insert an arbitrary section into our stub data containing our compressed executable to later retrieve and load it at runtime.

PACKERS

PACKING

- We compress our data with zlib’s compress function.
- We then align our stub buffer’s size data to the image header’s *file alignment* value and append our compressed data.
  - An address or value is *aligned* if it is evenly divisible by a certain value.

PACKERS

PACKING

```
// next, load the stub and get some initial information
std::vector<std::uint8_t> stub_data = load_resource(MAKEINTRESOURCE(IDB_STUB), "STUB");
auto dos_header = reinterpret_cast<IMAGE_DOS_HEADER *>(stub_data.data());
auto e_lfanew = dos_header->e_lfanew;

// get the nt header and get the alignment information
auto nt_header = reinterpret_cast<IMAGE_NT_HEADERS64 *>(stub_data.data() + e_lfanew);
auto file_alignment = nt_header->OptionalHeader.FileAlignment;
auto section_alignment = nt_header->OptionalHeader.SectionAlignment;

// align the buffer to the file boundary if it isn't already
if (stub_data.size() % file_alignment != 0)
   stub_data.resize(align<std::size_t>(stub_data.size(), file_alignment));
```

PACKERS

PACKING

```
// save the offset to our new section for later for our new PE section
auto raw_offset = static_cast<std::uint32_t>(stub_data.size());

// encode the size of our unpacked data into the stub data
auto unpacked_size = target.size();
stub_data.insert(stub_data.end(),
                 reinterpret_cast<std::uint8_t *>(&unpacked_size),
                 reinterpret_cast<std::uint8_t *>(&unpacked_size)+sizeof(std::size_t));

// add our compressed data.
stub_data.insert(stub_data.end(), packed.begin(), packed.end());
```

PACKERS

PACKING

- Our data is there, but the metadata is not aware of it.
- We start by incrementing the number of sections present in the file header of the NT headers.

```
// increment the number of sections in the file header
auto section_index = nt_header->FileHeader.NumberOfSections;
++nt_header->FileHeader.NumberOfSections;
```

PACKERS

PACKING

- Next, we acquire the section table.
- The section table directly follows the optional header in the NT headers, but don't be fooled-- the section header actually follows it after the SizeOfOptionalHeader offset in the file header structure of the NT headers.

```
// acquire a pointer to the section table
auto size_of_header = nt_header->FileHeader.SizeOfOptionalHeader;
auto section_table = reinterpret_cast<IMAGE_SECTION_HEADER *>(
   reinterpret_cast<std::uint8_t *>(&nt_header->OptionalHeader)+size_of_header
);
```

PACKERS

PACKING

- We are concerned with the following variables in the section header:
  - Name
  - VirtualSize
  - VirtualAddress
  - SizeOfRawData
  - PointerToRawData
  - Characteristics

PACKERS

PACKING

- A Windows executable has two memory states: a *disk* state and a *memory* state.
- These states have entirely different memory layouts.
- VirtualAddress and VirtualSize correspond to those states in memory. VirtualAddress is a *memory* offset, also known as a “relative virtual address,” or RVA.
- PointerToRawData and SizeOfRawData correspond to those states on disk. PointerToRawData is a *disk* offset.
- Characteristics represents various information about the section, such as whether or not it’s executable.

PACKERS

PACKING

- When adding our compressed data, we saved a disk offset to our data for the purpose of adding it to the section table.
- With an alignment function, we can easily calculate our new section’s memory offset in accordance with the image’s memory alignment.
  - The alignment function essentially calculates the difference between the aligned value and the current value, then adds that difference to align the value.
- As for Characteristics, we simply need to mark the section as being readable and initialized.

PACKERS

PACKING

```
// get a pointer to our new section and the previous section
auto section = &section_table[section_index];
auto prev_section = &section_table[section_index-1];

// calculate the memory offset, memory size and raw aligned size of our packed section
auto virtual_offset = align(prev_section->VirtualAddress + prev_section->Misc.VirtualSize, section_alignment);
auto virtual_size = section_size;
auto raw_size = align<DWORD>(section_size, file_alignment);
```

PACKERS

PACKING

```
// assign the section metadata
std::memcpy(section->Name, ".packed", 8);
section->Misc.VirtualSize = virtual_size;
section->VirtualAddress = virtual_offset;
section->SizeOfRawData = raw_size;
section->PointerToRawData = raw_offset;

// mark our section as initialized, readable data.
section->Characteristics = IMAGE_SCN_MEM_READ | IMAGE_SCN_CNT_INITIALIZED_DATA;
```

PACKERS

PACKING

- Unfortunately, now that we've added this section, we've increased the size of the image.
- Simply adjust the ImageSize variable in the optional header to fix this.

```
// calculate the new size of the image.
nt_header->OptionalHeader.SizeOfImage = align(virtual_offset + virtual_size, section_alignment);
```

- Now save your stub buffer to disk and you have a packed stub!

PACKERS

UNPACKING

- The unpacking process, as an abstract, is simple.
  - Decompress the target binary
  - Simulate the loader as much as you’d like
  - Call the entrypoint of the original image
- Obviously, the second step is the most complex.

PACKERS

UNPACKING

- As a bare minimum, to load our dummy binary, we need to do two things:
  - Resolve DLL imports
  - Relocate execution data to point at our loaded memory
- We can then proceed to get our entrypoint (traditionally called the “original entrypoint”, or OEP) out of the headers and call it.
- In total, this actually produces a very simple main routine.

PACKERS

UNPACKING

```
int main(int argc, char *argv[]) {
   // first, decompress the image from our added section
   auto image = get_image();

   // next, prepare the image to be a virtual image
   auto loaded_image = load_image(image);

   // resolve the imports from the executable
   load_imports(loaded_image);

   // relocate the executable
   relocate(loaded_image);

   // get the headers from our loaded image
   auto nt_headers = get_nt_headers(loaded_image);

   // acquire and call the entrypoint
   auto entrypoint = loaded_image + nt_headers->OptionalHeader.AddressOfEntryPoint;
   reinterpret_cast<void(*)()>(entrypoint)();

   return 0;
}
```

PACKERS

UNPACKING

- We can access our executable headers at runtime with the GetModuleHandleA API function.
- From there, retrieving our compressed binary from the stub is as simple as parsing out the section we added, then doing some pointer arithmetic to retrieve it, and finally decompressing the datastream.

PACKERS

UNPACKING

```
// find our packed section
auto base = reinterpret_cast<const std::uint8_t *>(GetModuleHandleA(NULL));
auto nt_header = get_nt_headers(base);
auto section_table = reinterpret_cast<const IMAGE_SECTION_HEADER *>(
   reinterpret_cast<const std::uint8_t *>(&nt_header->OptionalHeader)+nt_header->FileHeader.SizeOfOptionalHeader
);
const IMAGE_SECTION_HEADER *packed_section = nullptr;

for (std::uint16_t i=0; i<nt_header->FileHeader.NumberOfSections; ++i)
{
   if (std::memcmp(section_table[i].Name, ".packed", 8) == 0)
   {
      packed_section = &section_table[i];
      break;
   }
}

if (packed_section == nullptr) {
   std::cerr << "Error: couldn't find packed section in binary." << std::endl;
   ExitProcess(1);
}
```

PACKERS

UNPACKING

```
// decompress our packed image
auto section_start = base + packed_section->VirtualAddress;
auto section_end = section_start + packed_section->Misc.VirtualSize;
auto unpacked_size = *reinterpret_cast<const std::size_t *>(section_start);
auto packed_data = section_start + sizeof(std::size_t);
auto packed_size = packed_section->Misc.VirtualSize - sizeof(std::size_t);

auto decompressed = std::vector<std::uint8_t>(unpacked_size);
uLong decompressed_size = static_cast<uLong>(unpacked_size);

if (uncompress(decompressed.data(), &decompressed_size, packed_data, packed_size) != Z_OK)
{
   std::cerr << "Error: couldn't decompress image data." << std::endl;
   ExitProcess(2);
}

return decompressed;
```

PACKERS

UNPACKING

- Remember disk and memory addresses? We need to convert our disk image to a memory image.
- We start with a call to VirtualAlloc with the ImageSize value from our headers, noting that the new memory page should be executable. We then map our section data, via their VirtualAddress offsets, onto that new memory section.
  - In our example, we also copy the PE headers. You can choose not to, but it makes things harder.

PACKERS

UNPACKING

```
// get the original image section table
auto nt_header = get_nt_headers(image.data());
auto section_table = reinterpret_cast<const IMAGE_SECTION_HEADER *>(
   reinterpret_cast<const std::uint8_t *>(&nt_header->OptionalHeader)+nt_header->FileHeader.SizeOfOptionalHeader
);

// create a new VirtualAlloc'd buffer with read, write and execute privileges
// that will fit our image
auto image_size = nt_header->OptionalHeader.SizeOfImage;
auto base = reinterpret_cast<std::uint8_t *>(VirtualAlloc(nullptr,
                                                          image_size,
                                                          MEM_COMMIT | MEM_RESERVE,
                                                          PAGE_EXECUTE_READWRITE));

if (base == nullptr) {
   std::cerr << "Error: VirtualAlloc failed: Windows error " << GetLastError() << std::endl;
   ExitProcess(3);
}
```

PACKERS

UNPACKING

```
// copy the headers to our new virtually allocated image
std::memcpy(base, image.data(), nt_header->OptionalHeader.SizeOfHeaders);

// copy our sections to their given addresses in the virtual image
for (std::uint16_t i=0; i<nt_header->FileHeader.NumberOfSections; ++i)
   if (section_table[i].SizeOfRawData > 0)
      std::memcpy(base+section_table[i].VirtualAddress,
                  image.data()+section_table[i].PointerToRawData,
                  section_table[i].SizeOfRawData);

return base;
```

PACKERS

UNPACKING

- Certain PE data structures are stored in what's called the *data directory.* These directories are where we'll parse out imports and relocations.
- We get a directory-specific data structure by parsing at the VirtualAddress of the given data directory.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD   VirtualAddress;
    DWORD   Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

PACKERS

UNPACKING

```
// get the import table directory entry
auto nt_header = get_nt_headers(image);
auto directory_entry = nt_header->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];

// if there are no imports, that's fine-- return because there's nothing to do.
if (directory_entry.VirtualAddress == 0) { return; }

// get a pointer to the import descriptor array
auto import_table = reinterpret_cast<IMAGE_IMPORT_DESCRIPTOR *>(image + directory_entry.VirtualAddress);
```

PACKERS

UNPACKING

- The import directory consists of an array of *import descriptors*.
- These import descriptors contain an address to the string of the DLL to import as well as their API entries.
- In an import descriptor is an *import table* (functions to import) and an *address table* (functions which have had their addresses resolved). Though the names are confusing, these correspond to the OriginalFirstThunk (import table) and the FirstThunk (address table) within the import descriptor.

PACKERS

UNPACKING

- Imports and exports rely on what are called *thunks*. These are addresses which could represent a number of object types.

- For imports, we will only ever encounter an *ordinal* and an *address of data* thunk.

```
typedef struct _IMAGE_THUNK_DATA64 {
    union {
        ULONGLONG ForwarderString;  // PBYTE
        ULONGLONG Function;         // PDWORD
        ULONGLONG Ordinal;
        ULONGLONG AddressOfData;    // PIMAGE_IMPORT_BY_NAME
    } u1;
} IMAGE_THUNK_DATA64;
```

PACKERS

UNPACKING

- An *ordinal* is a value with a flagged most-significant-bit (for differentiation) and an index value into the DLL's export table. This is essentially a lookup by index rather than API name.
- An *address of data* thunk points at this data structure:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;
    CHAR    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

PACKERS

UNPACKING

- The import descriptor array is null-terminated, like a string.
- The import table and address table are also null-terminated. They are also mirrored (i.e., import table thunk 0 goes into address table thunk 0).
- To resolve the imports for the address table, we use the function LoadLibraryA to load the given DLL in the descriptor and GetProcAddress for both the ordinal and the import-by-name structure.

PACKERS

UNPACKING

- When simplified, this is what our import loop looks like:

```
for every import descriptor:
    load the dll
    parse the original and first thunk

    for every thunk:
        if ordinal bit set:
            import by ordinal
        else:
            import by name

        store import in first thunk
```

PACKERS

UNPACKING

```
// when we reach an OriginalFirstThunk value that is zero, that marks the end of our array.
// typically all values in the import descriptor are zero, but we do this
// to be shorter about it.
while (import_table->OriginalFirstThunk != 0)
{
   // get a string pointer to the DLL to load.
   auto dll_name = reinterpret_cast<char *>(image + import_table->Name);

   // load the DLL with our import.
   auto dll_import = LoadLibraryA(dll_name);

   if (dll_import == nullptr) {
      std::cerr << "Error: failed to load DLL from import table: " << dll_name << std::endl;
      ExitProcess(4);
   }

   // load the array which contains our import entries
   auto lookup_table = reinterpret_cast<IMAGE_THUNK_DATA64 *>(image + import_table->OriginalFirstThunk);

   // load the array which will contain our resolved imports
   auto address_table = reinterpret_cast<IMAGE_THUNK_DATA64 *>(image + import_table->FirstThunk);
```

PACKERS

UNPACKING

```
// an import can be one of two things: an "import by name," or an "import ordinal," which is
// an index into the export table of a given DLL.
while (lookup_table->u1.AddressOfData != 0)
{
   FARPROC function = nullptr;
   auto lookup_address = lookup_table->u1.AddressOfData;

   // if the top-most bit is set, this is a function ordinal.
   // otherwise, it's an import by name.
   if (lookup_address & IMAGE_ORDINAL_FLAG64 != 0)
   {
      // get the function ordinal by masking the lower 32-bits of the lookup address.
      function = GetProcAddress(dll_import,
                                reinterpret_cast<LPSTR>(lookup_address & 0xFFFFFFFF));

      if (function == nullptr) {
         std::cerr << "Error: failed ordinal lookup for " << dll_name << ": " << (lookup_address & 0xFFFFFFFF) << std::endl;
         ExitProcess(5);
      }
   }
```

PACKERS

UNPACKING

```
      else {
         // in an import by name, the lookup address is an offset to
         // an IMAGE_IMPORT_BY_NAME structure, which contains our function name
         // to import
         auto import_name = reinterpret_cast<IMAGE_IMPORT_BY_NAME *>(image + lookup_address);
         function = GetProcAddress(dll_import, import_name->Name);

         if (function == nullptr) {
            std::cerr << "Error: failed named lookup: " << dll_name << "!" << import_name->Name << std::endl;
            ExitProcess(6);
         }
      }

      // store either the ordinal function or named function
      // in our address table.
      address_table->u1.Function = reinterpret_cast<std::uint64_t>(function);

      // advance to the next entries in the address table and lookup table
      ++lookup_table;
      ++address_table;
   }

   // advance to the next entry in our import table
   ++import_table;
}
```

PACKERS

UNPACKING

- Addresses for a binary are usually hardcoded to point at the original image base. This is what the *relocation directory* aims to fix when a dynamically allocated base is involved.
  - A dynamic base for a binary is also known as “address space layout randomization,” or ASLR.
- This is an optional characteristic for a binary that may or may not be present. For our example, it is.
- We can check for the presence of a dynamic base by checking the DllCharacteristics variable of the optional header.

PACKERS

UNPACKING

```
// first, check if we can even relocate the image. if the dynamic base flag isn't set,
// then this image probably isn't prepared for relocating.
auto nt_header = get_nt_headers(image);

if (nt_header->OptionalHeader.DllCharacteristics & IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE == 0)
{
   std::cerr << "Error: image cannot be relocated." << std::endl;
   ExitProcess(7);
}

// once we know we can relocate the image, make sure a relocation directory is present
auto directory_entry = nt_header->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];

if (directory_entry.VirtualAddress == 0) {
   std::cerr << "Error: image can be relocated, but contains no relocation directory." << std::endl;
   ExitProcess(8);
}
```

PACKERS

UNPACKING

- While relocations for 32-bit binaries gets complicated, 64-bit is relatively simple:
  - For each relocation, add the base delta.
- The harder part is parsing the relocation data.
- Calculating the base delta is simple: subtract the ImageBase header in the optional header from the allocated image base.

```
// calculate the difference between the image base in the compiled image
// and the current virtually allocated image. this will be added to our
// relocations later.
std::uintptr_t delta = reinterpret_cast<std::uintptr_t>(image) - nt_header->OptionalHeader.ImageBase;
```

PACKERS

UNPACKING

```
typedef struct _IMAGE_BASE_RELOCATION {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
//  WORD    TypeOffset[1];
} IMAGE_BASE_RELOCATION;
```

This is a relocation directory header. This is another null-terminated array.

- VirtualAddress points to the base address to relocate.
- SizeOfBlock is the size of the entire relocation entry, header and all.
- TypeOffset is an array of encoded offsets from the VirtualAddress base to relocate. While it's commented out, it does in fact reside directly after SizeOfBlock. We'll need to do some pointer math to get to it.

PACKERS

UNPACKING

- TypeOffset encodes 16-bit values in the following way:
  - The top 4 bits (mask 0xF000) contain the relocation type.
  - The lower 12 bits (mask 0x0FFF) contain the offset from the VirtualAddress to relocate.
- We are only concerned with a DIR64 relocation type, which is just adding the base delta to the pointer created.

PACKERS

UNPACKING

- Explaining how to parse relocations obfuscates how simple the operation is. A pointer to a relocation looks like this:

```
auto ptr = reinterpret_cast<std::uintptr_t *>(image + relocation_table->VirtualAddress + offset);
```

- And performing the relocation looks like this:

```
*ptr += delta;
```

- Onto the code!

PACKERS

UNPACKING

```
// get the relocation table.
auto relocation_table = reinterpret_cast<IMAGE_BASE_RELOCATION *>(image + directory_entry.VirtualAddress);

// when the virtual address for our relocation header is null,
// we've reached the end of the relocation table.
while (relocation_table->VirtualAddress != 0)
{
   // since the SizeOfBlock value also contains the size of the relocation table header,
   // we can calculate the size of the relocation array by subtracting the size of
   // the header from the SizeOfBlock value and dividing it by its base type: a 16-bit integer.
   std::size_t relocations = (relocation_table->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) / sizeof(std::uint16_t);

   // additionally, the relocation array for this table entry is directly after
   // the relocation header
   auto relocation_data = reinterpret_cast<std::uint16_t *>(&relocation_table[1]);
```

PACKERS

UNPACKING

```
for (std::size_t i=0; i<relocations; ++i)
{
   // a relocation is an encoded 16-bit value:
   //   * the upper 4 bits are its relocation type
   //     (https://learn.microsoft.com/en-us/windows/win32/debug/pe-format see "base relocation types")
   //   * the lower 12 bits contain the offset into the relocation entry's address base into the image
   //
   auto relocation = relocation_data[i];
   std::uint16_t type = relocation >> 12;
   std::uint16_t offset = relocation & 0xFFF;
   auto ptr = reinterpret_cast<std::uintptr_t *>(image + relocation_table->VirtualAddress + offset);

   // there are typically only two types of relocations for a 64-bit binary:
   //   * IMAGE_REL_BASED_DIR64: a 64-bit delta calculation
   //   * IMAGE_REL_BASED_ABSOLUTE: a no-op
   //
   if (type == IMAGE_REL_BASED_DIR64)
      *ptr += delta;
}
```

PACKERS

UNPACKING

```
// the next relocation entry is at SizeOfBlock bytes after the current entry
relocation_table = reinterpret_cast<IMAGE_BASE_RELOCATION *>(
   reinterpret_cast<std::uint8_t *>(relocation_table) + relocation_table->SizeOfBlock
);
```

PACKERS

UNPACKING

- Congratulations! We've just done the following:
  - Decompressed and remapped our binary for execution
  - Parsed and loaded the imports
  - Relocated the image to point at the right addresses
- Now we need to transfer execution to our unpacked image.

PACKERS

UNPACKING

- In the NT optional headers is AddressOfEntryPoint, an RVA to the entrypoint of the given binary.
- We simply use a function pointer at the entrypoint to call our loaded image, which looks like this:

```
return_type (*variable_name)(int arg1, int arg2, ...)
```

PACKERS

UNPACKING

```
// get the headers from our loaded image
auto nt_headers = get_nt_headers(loaded_image);

// acquire and call the entrypoint
auto entrypoint = loaded_image + nt_headers->OptionalHeader.AddressOfEntryPoint;
reinterpret_cast<void(*)()>(entrypoint)();
```

PACKERS

ALL DONE!

- With the entrypoint transferred, you’re done! You’ve successfully packed and unpacked a Windows executable.
- Further exercises to get a good handle on packing:
  - Support 32-bit executables
  - Support other data directories, such as TLS and resources
  - Learn various anti-analysis techniques to harden your packer