Manalyze Documentation

Release 1.0

Ivan Kwiatkowski

Feb 15, 2020

Contents

Manalyze performs static analysis on PE files, in order to detect signs of malicious behavior. It is a versatile tool with a robust parser and a set of built-in tests, but can also be extended easily. You can use Manalyze to:

- Detect packed executables
- Apply ClamAV and Yara signatures
- Look for suspicious import combinations (i.e. `CreateRemoteThread` + `WriteProcessMemory`)
- Analyze and extract resources
- Identify cryptographic algorithms used
- Submit hashes to VirusTotal
- Verify authenticode signatures
- . . . and more.

Here is a sample report generated by the tool for `643654975b63a9bb6f597502e5cd8f49`, a sample taken from the Siesta campaign:

```
Summary:
--------
Architecture:       IMAGE_FILE_MACHINE_I386
Subsystem:          IMAGE_SUBSYSTEM_WINDOWS_GUI
Compilation Date:   2014-Jan-14 04:38:30
Detected languages: Chinese - PRC

[ MALICIOUS ] Matching ClamAV signature(s):
                Win.Backdoor.Sloth

Matching compiler(s):
                MASM/TASM - sig4 (h)
                Microsoft Visual C++
                Microsoft Visual C++ v6.0

[ SUSPICIOUS ] PEiD Signature:
                Armadillo v1.71

Cryptographic algorithms detected in the binary:
                Uses constants related to DES

The PE contains common functions which appear in legitimate applications.
        [!] The program may be hiding some of its imports:
                GetProcAddress
                LoadLibraryA
        Possibly launches other programs:
                CreateProcessA
                ShellExecuteA
        Can create temporary files:
                CreateFileA
                GetTempPathA

[ MALICIOUS ] The PE is possibly a dropper.
                Resource 108 detected as a PDF document.
                Resource 109 detected as a PE Executable.
                Resources amount for 93.026% of the executable.

[ MALICIOUS ] VirusTotal score: 38/56 (Scanned on 2015-10-26 15:07:59)
                MicroWorld-eScan: Gen:Variant.Zusy.23178
```

(continues on next page)

(continued from previous page)

```
                CAT-QuickHeal: Trojan.Comisproc.r4
                [...]
```

This sample is a dropper of (allegedly) Chinese origin which displays a PDF file upon launch and encrypts its strings with the DES algorithm: all of which could have been guessed from reading the analysis report.

In the first part of this documentation, you will learn how to obtain and use the tool. The second part focuses on Manalyze's plugin system, should you wish to extend its capabilities.

Contents:

CHAPTER 1

User documentation

This section is intended for malware researchers who wish to use Manalyze to assist them in their work. It will cover obtaining the tool (either by downloading binaries or by compiling it yourself), configuration and basic usage.

If you wish to contribute to the project, check out the *Developer Documentation*!

1.1 Obtaining the tool

1.1.1 Binary distributions

Windows users can download the latest binaries here. Unzip the archive somewhere on your filesystem and you're ready to go! All the binaries are signed with a certificate presenting the following fingerprint : `26fc24c12b2d84f77615cf6299e3e4ca4f3878fc`.

Deb packages will hopefully be offered at some point but right now, using Manalyze on other operating systems requires compiling it yourself.

1.1.2 Building Manalyze

Spending hours trying to build someone else's code is one of the most horrendous experiences in software development. A lot of work was put into Manalyze's build system to ensure that anyone would be able to compile it with a minimum of friction. If the following instructions don't work for you, be sure to get in touch with the program's maintainer so the situation (or this documentation) can be improved.

In the general case, you can build this tool in four simple steps:

1. Obtaining the tools and libraries Manalyze depends on:
   - CMake
   - A recent version of Boost.
2. Checkout the program's source code from GitHub.

3. Using CMake to generate system-dependent build files. The CMake script will also check out additional libraries from GitHub.
4. Compile Manalyze. All the binaries are placed in the `bin/` folder.

Here are more specific steps for a few major operating systems:

Linux and BSD

How you take care of step 1 may vary depending on your package manager. On Debian Jessie, use the following command **as root**:

```
apt-get install libboost-regex-dev libboost-program-options-dev libboost-system-dev␣
↪libboost-filesystem-dev libssl-dev build-essential cmake git
```

On FreeBSD 10.2, use this one instead (also **as root**):

```
pkg install boost-libs-1.55.0_8 libressl cmake git
```

Next, get Manalyze's source code and try building it:

```
git clone https://github.com/JusticeRage/Manalyze.git && cd Manalyze
cmake .
make
cd bin && ./manalyze --version
```

If everything went well, the tool's version should be displayed. Otherwise, jump to the *Troubleshooting* section below or look for error messages during the build process and get in touch with the maintainer to request help!

Tip: You can enable debug builds with the following command: `cmake . -DDebug=ON`

Windows

Step 1 requires a bit more work on Windows, because the Boost libraries have to be built manually.

- First, get the latest version on the official website and extract them somewhere (for instance, `C:\code\boost_1_XX_0\`). Open a command prompt and navigate to that folder.
- Run the following command to build the required libraries:

```
./bootstrap.bat && ./b2.exe --build-type=complete --with-regex --with-program_
↪options --with-system --with-filesystem
```

- Set up the `BOOST_ROOT` environment variable to help CMake locate the libraries you just built. In this example, the environment variable should contain: `C:\code\boost_1_XX_0\`.
- Finally, if you haven't done it already, don't forget to install CMake and Git.

That's it for the dependencies. Steps 2 and 3 can be tackled with a single command:

```
git clone https://github.com/JusticeRage/Manalyze.git && cd Manalyze && cmake .
```

Build files should have appeared in Manalyze's folder. Usually, they take the form of a Visual Studio project (i.e. `manalyze.sln`). Double-click it to open it in the IDE, or run the following command inside a Visual Studio command prompt:

```
msbuild manalyze.sln
```

Binaries will appear in the `bin\` folder.

What about MacOS?

I do not own any Apple hardware, so the tool has never been built - let alone tested - on MacOS yet.

1.1.3 Offline builds

If you need to build Manalyze on a machine which doesn't have internet access, a couple of additional steps are required to manually obtain the libraries that the CMake script would normally obtain. Use the following commands to get the tool's source code:

```
git clone https://github.com/JusticeRage/Manalyze.git
cd Manalyze/external
git clone https://github.com/JusticeRage/hash-library.git
git clone https://github.com/JusticeRage/yara.git
```

Now take the whole `Manalyze` folder to the computer on which you intend to build the software (note that this computer still needs the Boost libraries and CMake). Now run the following command to tell the CMake script that it should not try to checkout or update the external libraries:

```
cmake . -DGitHub=OFF
```

. . . and continue as you normally would.

1.1.4 Troubleshooting

This section lists some common compilation errors you may face when building Manalyze, along with their solution.

1. Boost is obsolete

This compilation error is usually encountered on Debian 7 (Wheezy):

```
In file included from ~/Manalyze/manacommons/escape.cpp:18:0:
~/Manalyze/include/manacommons/escape.h:115:97: error: macro "BOOST_STATIC_ASSERT"␣
↪passed 3 arguments, but takes just 1
~/Manalyze/include/manacommons/escape.h:148:66: error: macro "BOOST_STATIC_ASSERT"␣
↪passed 2 arguments, but takes just 1
~/Manalyze/include/manacommons/escape.h: In function 'io::pString io::_do_
↪escape(const string&)':
~/Manalyze/include/manacommons/escape.h:115:2: error: 'BOOST_STATIC_ASSERT' was not␣
↪declared in this scope
~/Manalyze/include/manacommons/escape.h: In function 'io::pString io::escape(const␣
↪string&)':
~/Manalyze/include/manacommons/escape.h:148:2: error: 'BOOST_STATIC_ASSERT' was not␣
↪declared in this scope
~/Manalyze/include/manacommons/escape.h: In instantiation of 'io::pString io::_do_
↪escape(const string&) [with Grammar = io::escaped_string_raw<std::back_insert_
↪iterator<std::basic_string<char> > >; io::pString = boost::shared_ptr<std::basic_
↪string<char> >; std::string = std::basic_string<char>]':
```

(continues on next page)

(continued from previous page)

```
~/Manalyze/manacommons/escape.cpp:24:53:   required from here
~/Manalyze/include/manacommons/escape.h:125:10: error: could not convert 'nullptr'␣
↪from 'std::nullptr_t' to 'io::pString {aka boost::shared_ptr<std::basic_string<char>
↪ >}'
make[2]: *** [CMakeFiles/manacommons.dir/manacommons/escape.cpp.o] Error 1
make[1]: *** [CMakeFiles/manacommons.dir/all] Error 2
make: *** [all] Error 2
```

This issue has been traced to the Boost libraries in Wheezy repositories being too old (1.49.0). You'll need to either upgrade them manually or switch to Debian Jessie.

2. CMake does not find OpenSSL

Some versions of CMake (for instance 3.0.2, present in Debian Jessie's repositories) seem to have trouble locating OpenSSL and generate the following error messages:

```
CMake Error at /usr/share/cmake-3.0/Modules/FindOpenSSL.cmake:293 (list):
  list GET given empty list
Call Stack (most recent call first):
  CMakeLists.txt:23 (find_package)

[...]

-- Found OpenSSL: /usr/lib/x86_64-linux-gnu/libssl.so;/usr/lib/x86_64-linux-gnu/
↪libcrypto.so (found version ".0.0`")
```

Upgrading CMake to the latest release (3.5.2 at the time I'm writing this) solves this issue.

3. Incompatibilities between OpenSSL 1.1 and Boost

The following error may be encountered on Debian 9 (Stretch):

```
In function 'bool plugin::vt_api_interact(const string&, const string&, std::__
↪cxx11::string&, plugin::sslsocket&)': ~/Manalyze/plugins/plugin_virustotal/plugin_
↪virustotal.cpp:276:84: error: 'SSL_R_SHORT_READ' was not declared in this scope if␣
↪(error != boost::asio::error::eof && error.value() != ERR_PACK(ERR_LIB_SSL, 0, SSL_
↪R_SHORT_READ))
```

Starting with Stretch, Debian ships with the 1.1 branch of OpenSSL which is not compatible with most versions of Boost. It is unclear from which version the problem has been fixed, but a workaround for this issue is to download one of the latest Boost distributions from upstream and build it instead of using the libraries provided by Debian.

1.2 Initial configuration

You have just downloaded Manalyze, and while it runs on your system, there are just a few more steps to follow before you can use it fully. Some of the plugins bundled with the program need to be configured manually. In most cases, all you have to do is look at `bin/manalyze.conf` and see if there are any values which need editing.

1.2.1 VirusTotal plugin

When you use this plugin for the first time, you're likely to encounter the following error:

```
[*] Warning: The VirusTotal API key was not found in the configuration file.
```

In order to submit hashes to VirusTotal, it is necessary to register on their website and retrieve an API key. If you really can't be bothered, many of these can be found on GitHub.

VirusTotal offers two types of API access: public and private. Right now, Manalyze doesn't support any of the "private" features, but if you're lucky enough to have a such a key, at least you won't be bound by the request rate limit. After you have obtained an API key, edit `bin/manalyze.conf` and add the following line:

```
virustotal.api_key = [your key here]
```

After this, the plugin will be able to retrieve hashes from VirusTotal.

1.2.2 ClamAV plugin

Manalyze can apply ClamAV signatures to detect known malware. Those signature are however **not** distributed with the application because of their size, and the fact that they are constantly updated. This is the reason why running the ClamAV plugin for the first time is likely to print the following error:

```
[!] Error: Could not load yara rules (ERROR_COULD_NOT_OPEN_FILE).
[!] Error: ClamAV rules haven't been generated yet!
[!] Error: Please run yara_rules/update_clamav_signatures.py to create them, and␣
↪refer to the documentation for additional information.
```

You've been promised "additional information": here it is! ClamAV signatures have to be downloaded from the official website. But Manalyze can't read ClamAV signatures out of the box, they first need to be converted to Yara rules. The whole process was a little cumbersome, so a Python script was written to automate the process. Simply run:

```
python yara_rules/update_clamav_signatures.py
```

. . . and the rules will be added to Manalyze. Run the script anytime you want to update the signatures!

Additional considerations

ClamAV signatures are divided into two files, the "main" and the "daily" signatures. The former isn't updated very often, as opposed to the latter. For this reason, the python script will not download the "main" signatures if they have already been retreived: only the daily rules will be regenerated. To perform a full upgrade, call the script with the following parameter:

```
python yara_rules/update_clamav_signatures.py --main
```

1.3 Usage

If you have managed to *obtain* and *configure* Manalyze but want to know more about how to use it, you're in the right place! First, let's have a look at the program's help screen:

```
Usage:
  -h [ --help ]         Displays this message.
  -v [ --version ]      Prints the program's version.
  --pe arg              The PE to analyze. Also accepted as a positional
                        argument. Multiple files may be specified.
```

(continues on next page)

(continued from previous page)

```
  -r [ --recursive ]    Scan all files in a directory (subdirectories will be
                        ignored).
  -o [ --output ] arg   The output format. May be 'raw' (default) or 'json'.
  -d [ --dump ] arg     Dump PE information. Available choices are any
                        combination of: all, summary, dos (dos header), pe (pe
                        header), opt (pe optional header), sections, imports,
                        exports, resources, version, debug, tls, config, delay
  --hashes              Calculate various hashes of the file (may slow down the
                        analysis!)
  -x [ --extract ] arg  Extract the PE resources to the target directory.
  -p [ --plugins ] arg  Analyze the binary with additional plugins. (may slow
                        down the analysis!)

Available plugins:
  - clamav: Scans the binary with ClamAV virus definitions.
  - compilers: Tries to determine which compiler generated the binary.
  - peid: Returns the PEiD signature of the binary.
  - strings: Looks for suspicious strings (anti-VM, process names...).
  - findcrypt: Detects embedded cryptographic constants.
  - packer: Tries to structurally detect packer presence.
  - imports: Looks for suspicious imports.
  - resources: Analyzes the program's resources.
  - mitigation: Displays the enabled exploit mitigation techniques (DEP, ASLR, etc.).
  - authenticode: Checks if the digital signature of the PE is valid.
  - virustotal: Checks existing AV results on VirusTotal.
  - all: Run all the available plugins.

Examples:
  manalyze.exe program.exe
  manalyze.exe -dresources -dexports -x out/ program.exe
  manalyze.exe --dump=imports,sections --hashes program.exe
  manalyze.exe -r malwares/ --plugins=peid,clamav --dump all
```

Most options are self-explanatory, but let's go over them anyway.

1.3.1 Selecting target programs

In order to choose which program(s) should be analyzed, you can use the `--pe` option. Targets are also accepted as positional arguments; this means that listing them on the command line without prefixing them with any particular flag will work. You can specify as many files as you want: they will be studied sequentially. The `-r` (or `--recursive`) option allows you to scan whole directories - even if they contain gigabytes of files (have fun reading the reports though). However, subdirectories will be ignored. For instance, if you have the following folder structure:

```
dir/
   |- malware1.exe
   |- lib1.dll
   `- dropped/
       |- malware2.exe
       `- lib2.dll
```

...then running a recursive analysis on this folder will **not** process malware2.exe and lib2.dll. Use `./manalyze -r dir dir/dropped` to analyze all of them.

1.3.2 Dumping a PE's structure

Since Manalyze implements a PE parser, you can use it to look closely at the structure of target files. the `--dump` (or `-d`) option allows you to control what part of the PE you want to print. For instance, to look at a PE's sections, use `./manalyze [target file] -d sections`. You can of choose to display several categories at once. In terms of syntax, `./manalyze [target file] -d sections -d imports` and `./manalyze [target file] -d sections,imports` are equivalent.

Here is the list of all supported categories:

- **summary**: Contains general information on the input file. It gathers all the metadata which may be relevant to the interests of a malware researcher: possible debug paths present in the binary, a list of detected resource and/or manifest languages, compilation date, etc.
- **dos**, **pe**, **opt**: The DOS, PE, and PE optional headers respectively.
- **sections**; The sections of the PE. Note that if the `--hashes` option has been set, the returned information will also contain the hashes of each section.
- **imports** and **exports**: The imported functions and exported functions of the input file.
- **resources**: Displays information about the resources included in target PE files (size, entropy, filetype if possible, etc.). Cryptographic hashes will also be displayed if the `--hashes` option was activated. You may also be interested in the companion `--extract` (or `-x`) option, which allows you to write the resources inside the folder of your choice. Note that it is of course possible to extract resources without printing information about them, and vice-versa.
- **version**, **debug** and **tls**: These categories respectively show the data contained in the `RT_VERSION` resource, some metadata about embedded debug information and possible TLS callbacks.
- **all**: Dump everything.

If the requested data is not present (for instance, if no TLS callbacks are present in the input file), Manalyze simply won't return anything for the requested category. If no category is requested, the program will display the summary information by default. Finally, in addition to the uses described above, the `--hashes` option will also print the file hashes (MD5, SHA1, SHA256, SHA3, imphash and ssdeep) if given.

1.3.3 Using the plugins

While reading raw PE data may be interesting, Malalyze was designed so that tools could process this information automatically and generate meaningful reports based on them. The basic workflow of the project goes like this:

1. The PE parser gathers as much data as possible on a given input file.
2. The obtained data is provided to plugins so they can study, mine and/or correlate it to give an opinion about whether a program is malicious or not, or simply print out information which would be relevant to someone analyzing the file.

The following plugins are available:

- **clamav**: Applies ClamAV signatures to detect known malware. In order to use this plugin, make sure that you have *downloaded the signatures*!
- **compilers**: Applies PEiD signatures to try to detect the compiler which generated the input file.
- **strings**: Looks for suspicious strings and patterns inside the binary (i.e. references to `cmd.exe`, anti-VM opcodes, etc.).
- **findcrypt**: Detects cryptographic capabilities in a binary by looking at imports and searching for constants used in well-known algorithms.

- **packer**: Applies PEiD signatures to try to detect if the file was packed. Warnings will also be raised based on unusual section names and a low number of imports (which can be set in the configuration file to better suit your needs).
- **imports**: Guesses a PE file's capabilities through its imported functions.
- **resources**: Analyzes a program's resources to see if it contains encrypted files and/or suspicious filetypes. This plugin also contains a couple of heuristic methods to determine if a file might be a dropper.
- **mitigation**: Checks which exploit mitigation techniques (/GS, SafeSEH, ASLR and DEP) are enabled in the binary.
- **authenticode**: Checks the validity of a PE file's signature. At the moment, this plugin is only available on Windows platforms, since it relies heavily on that operating system's API.
- **virustotal**: Submits the hash of the input file to VirusTotal to see if any antivirus engine detects it as malware.
- **all**: Run all plugins.

Installing plugins

I'm not aware of any third-party plugins at the moment, but should anyone develop one, all you have to do to use it is download the `.dll` or `.so` file (depending on your OS) and place it next to Manalyze's binary. It will be detected automatically.

CHAPTER 2

Developer documentation

Welcome to the developer documentation! This section serves as a reference for people willing to contribute to the project. First of all, thank you for wanting to make Manalyze better! During the course of this chapter, we will discuss considerations which should be taken into account before submitting code. Then we'll look at how the code is organized and formatted in the project. Finally, we'll look at the two ways this project can be extended: by writing a plugin to add analysis capabilities and by improving the core of the tool.

In this chapter, it is assumed that you have obtained a copy of the program's source code and know how to build it. If it is not the case, please refer to the *obtaining the tool* page.

2.1 Before contributing

If you're reading this, you're probably eager to start writing code, but please bear with me for a few more minutes: if you don't take into account the instructions contained on this page, your contributions to the project may be rejected regardless of their quality!

2.1.1 About the GPLv3 license

Manalyze is distributed under the terms of the GPLv3 license. If you wish to contribute to the project's, you must agree to its terms and use the same license for code you submit. All source files should start with the following header:

```
/*
    This file is part of Manalyze.

    Manalyze is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    Manalyze is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
```

(continues on next page)

(continued from previous page)

```
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with Manalyze.  If not, see <http://www.gnu.org/licenses/>.
*/
```

If the code you are submitting depends on third-party libraries, make sure that their license is compatible - if it's not, we will not be able to accept your contribution. Refer to the GNU website to check whether a particular license can be used alongside the GPLv3.

Note: If you would like to use any part of Manalyze in a commercial product but the GPL license isn't compatible with it, get in touch with the maintainer: dual-licensing options are available.

2.1.2 Coding style

All code contributions should make every effort to match Manalyze's coding style as closely as possible. Here are the most important things to look for:

- **Naming conventions**
  - Use CamelCase for class names (i.e. `class MyShinyClass {};`).
  - Use lowercase for function names, function members and local variables. Separate words with underscores (i.e. `void process_int(int i);`).
  - Use capital letters for global variables and program-wide contants, and underscores to separate words (i.e. `#define NUMBER_OF_TRIES 10`).
  - Prefix private class method and member names with an underscore (i.e `private: std::string _private_string;`).
  - Choose lowercase, preferably short namespace names (i.e. `namespace plugins { ... }`).
- **Code structure**
  - All your code should reside in a meaningful namespace.
  - Declare class and functions in header (".h") files.
  - Protect header files against multiple inclusions with `#pragma once`.
  - Inclusion of system headers should precede inclusion of user-defined headers. Boost headers are considered system headers.
  - Put function implementations in .cpp files.
  - Function declarations must be documented following the Doxygen convention.
  - Please thoroughly explain non-intuitive code fragments with inline comments.
- **Formatting**
  - Do not put multiple statements on a single line (i.e. `int a = 1; initialize(a);` is not accepted.).
  - For improved readability, please insert the following separator between function declarations:

```
// ------------------------------------------------------------------------
↪-----
```

- Indent your code. You can use either tabluations or blocks of 4 spaces.
- Also indent preprocessor directives:

```
#ifdef BOOST_WINDOWS_API
#       define PLUGIN_API __declspec(dllexport)
#else
#       define PLUGIN_API __attribute__((visibility("default")))
#endif
```

- Do not import whole namespaces in headers (i.e. `using namespace std;` is prohibited).
- Pointer and reference being part of the type, write `char* s1;` or `std::string& s2;` instead of `char *s1;` or `std::string &s2;`.
- Trigraphs and digraphs are banned.
- Never omit brackets in control structures.
- Put brackets on a new line, unless there is only one line of code in the logic block:

```
if (flag) { // Okay
        perform_action();
}

if (flag)
{
        // Okay
        perform_first_action();
        perform_second_action();
}

if (flag) { // Not okay.
    perform_first_action();
    perform_second_action();
}

if (flag)
{
    perform_action(); // Frowned upon.
}
```

- **General recommendations**
  - Use the const keyword wherever applicable.
  - Pass function parameters by constant references when possible (i.e. `void process_string(const std::string& s);`).
  - Avoid global variables.
  - The `goto` keyword may be tolerated if it prevents code duplication and does not overly complicate the program flow. In particular, `goto END;` constructs can be used to go directly to the cleanup of a function before returning.
  - In order to prevent memory leaks, memory should not be managed manually. Use smart pointers.
  - For better encapsulation, prefer non-member non-friend functions when adequate.

2.1.3 Getting Help

Here is how you can request some assistance for problems encountered while trying to contribute to the project:

- If you find a bug, or feel that the current API is not exhaustive enough for a something you're trying to do, create an issue on GitHub.
- If you find this documentation lacking and don't know how where to begin in order to work on a feature you have in mind, contact the project's maintainer directly over e-mail. If the volume gets out of hand, and IRC channel and/or mailing list will be created.

2.2 How the code is organized

This project currently has over 8000 lines of code and it may be hard at first to find what you're looking for. This page contains an overview of how the code is organized and will help you figure out where the implementations you need to find are located.

2.2.1 The root folder

Let's start at the root folder of the project. It only contains one important file: `CMakeLists.txt`, which is CMake's configuration file. If you add new sources in your contribution (as opposed to only modifying existing ones), you will need to edit its contents in order to let the build system know about them. Apart from that, Manalyze's README and its license can be found here.

2.2.2 The `bin` folder

This is where generated binaries are put - but it contains a few files even if you haven't compiled the project yet:

- `manalyze.conf` is a configuration file for the program, pre-filled with default values.
- The `yara_rules` folder contains Yara rules used by the different plugins. You will also find a Python script which generates ClamAV rules by downloading the latest signatures from the official website and converting them to the right format.

2.2.3 The `external` folder

This directory should be mostly empty before you run CMake. It contains the third-party libraries that Manalyze is built against. Those libraries are checked out from GitHub during the compilation process. At the moment, there are two of them:

- Hash Library, a set of hashing algorithm implementations authored by Stephan Brumme.
- Yara, the well known pattern matching tool from plusvic. A few modifications have been made to this project, which justifies maintaining our own fork:
  - The code has been stripped down to a library (the command-line tool has been removed).
  - This version is built with CMake instead of the original Makefile.
  - A C++ wrapper was added to facilitate Yara's manipulation and integration with Manalyze.
  - All modules have been disabled, and a new one was written so Yara can receive information from Manalyze. In particular, their PE module was replaced since Manalyze already contains a powerful PE parser.

2.2.4 Source folders

The code of the project is spread out in the following folders:

- `src`: contains Manalyze's "engine". The entry point of the application is located there (in `main.cpp`), as well as all the functions tasked with validating the arguments, loading the configuration and the plugins, and of course launching the analysis for each target file.
- `manacommons`: a small library which contains functionnality shared between the core and the plugins, for instance what a plugin result or an analysis report should look like, or how to print text in color.
- `manape`: all the code related to PE parsing is located in this library.
- `plugins`: as you might expect, this folder contains Manalyze's plugins. Some of them are fairly simple and fit in a single .cpp file; others are bigger and are subsequently put in a separate folder.
- `include`: in this directory, you will find the headers of all the source files contained in the folders described above. If you want to understand the program's API quickly, it is recommended that you have a look at the files located here: function declarations are thoroughly commented and will give you a good idea of each class' capabilities.

2.2.5 Other folders

- Inside `docs`, you will find the reStructured Text which is used to generate this documentation!
- The `resources` folder contains some useful documentation, like the PE format specification.

2.3 Writing plugins

In this section, we'll learn how to write plugins for this project.

2.3.1 Internal and external plugins

There are two ways plugins can be integrated in Manalyze. You can either:

- Statically bundle them within the executable of the application (internal plugins).
- Build them as a separate library which will be loaded dynamically (external plugins).

Which one should you choose? Here are some guidelines to help you decide:

- If you intend to distribute your plugin, you should write an external plugin. It makes more sense to share a `.so` or `.dll` file than a whole new Manalyze binary with added code.
- If your plugin is relatively small and isn't meant to be shared, it is okay to write an internal plugin. Conversely, complex plugins which pull third-party libraries should be compiled in their own module.

In any case, aside from small discreapancies in the way each type of plugin is built, the code you will write will be mostly identical.

2.3.2 A sample plugin

Let's dive right into it and write an Hello World plugin! Let's create `plugins/plugin_helloworld.cpp`.

Internal plugin skeleton:

```
#include "plugin_framework/plugin_interface.h"
#include "plugin_framework/auto_register.h"

namespace plugin
{

class HelloWorldPlugin : public IPlugin
{

};

AutoRegister<HelloWorldPlugin> auto_register_helloworld;

} //!namespace plugin
```

Modifications to `CMakeLists.txt`: add a reference to this new source file with the other internal plugins, on the third line of the following snippet:

```
add_executable(manalyze src/main.cpp src/config_parser.cpp src/output_formatter.cpp␣
↪src/dump.cpp
           src/plugin_framework/dynamic_library.cpp src/plugin_framework/plugin_
↪manager.cpp # Plugin system
           plugins/plugins_yara.cpp plugins/plugin_packer_detection.cpp plugins/
↪plugin_imports.cpp plugins/plugin_resources.cpp plugins/plugin_helloworld.cpp) #␣
↪Bundled plugins
```

External plugin skeleton:

```
#include "plugin_framework/plugin_interface.h"

namespace plugin
{

class HelloWorldPlugin : public IPlugin
{

};

// ----------------------------------------------------------------------------

extern "C"
{
    PLUGIN_API IPlugin* create() { return new HelloWorldPlugin(); }
    PLUGIN_API void destroy(IPlugin* p) { delete p; }
};

} //!namespace plugin
```

Modifications to `CMakeLists.txt`: declare a new library, for instance just under the VirusTotal plugin:

```
# HelloWorld plugin
add_library(plugin_helloworld SHARED plugins/plugin_helloworld.cpp)
target_link_libraries(plugin_helloworld manape hash-library manacommons)
```

There are some parts missing, but it's okay for now. Points of interest are the mandatory included file(s), and the definition of a new class inheriting from `plugin::IPlugin` which defines the interface all plugins must adhere to. Internal plugins contain some additional magic to let the core know about them at startup. If you're building an external plugin, omit the `AutoRegister` instance: Manalyze will find it by scanning its folder for library files. Instead, you have to define the `create` and `destroy` functions so the core can load and unload your plugin.

If you try to build the plugin right now, you'll see that the compiler is very annoyed about some missing functions. Let's go back to our source file and finish our plugin's implementation:

```
class HelloWorldPlugin : public IPlugin
{
    int get_api_version() const override { return 1; }

    pString get_id() const override {
        return boost::make_shared<std::string>("helloworld");
    }

    pString get_description() const override {
        return boost::make_shared<std::string>("A sample plugin.");
    }

    pResult analyze(const mana::PE& pe) override
    {
        pResult res = create_result();
        res->add_information("Hello world from the plugin!");
        return res;
    }
};
```

These functions serve the following purpose:

- `get_api_version`: the version of the API used by this plugin, in case it evolves and breaks retro-compatibility in the future. Just return 1 for now.
- `get_id`: the name of the plugin. This is how it will be referred to in the program's help and on the command-line; make sure to pick something unique!
- `get_description`: a short explanation of what the plugin does. It is only printed when the user calls Manalyze with the `--help` option.
- `analyze`: performs the analysis of the program. We'll get back to this one very soon, for now, it just creates a result object containing a message.

Build the project again, and the plugin will automatically appear in the program's help:

```
$ bin/manalyze --help
Usage:
  -h [ --help ]         Displays this message.
  [...]

Available plugins:
  [...]
  - helloworld: A sample plugin.
  - all: Run all the available plugins.

$ bin/manalyze -p helloworld malware.mal
* Manalyze 1.0 *

------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
malware.mal
-------------------------------------------------------------------------------

Summary:
--------
Architecture:       IMAGE_FILE_MACHINE_I386
Subsystem:          IMAGE_SUBSYSTEM_WINDOWS_GUI
Compilation Date:   2015-Apr-23 16:45:58
Detected languages: English - United States

    Hello world from the plugin!
```

Great, our code has been called! Now let's try doing something useful.

2.3.3 Plugin results

After performing whatever work they do, plugins send back analysis data to the program's core through `plugin::Result` objects. These objects are composed of three things:

- A threat level, which indicates how dangerous the target file is according to the plugin. Keep in mind that plugins are only expected to give an opinion limited to their scope. In other words, it's okay for some plugins to mark known malware as safe: for example, the authenticode plugin would return this threat level for a malware with a valid digital signature. It's the user's job to take all the plugin results into account and determine whether the file is malicious or not.
- A summary describing the plugin's general findings on the PE, or introducing the information which follows.
- Pieces of textual information providing more detailed insight on the target file.

Tip: For instance, the `imports` plugin may return a result containing the following data:

```
Threat Level: MALICIOUS
Summary: The PE contains functions mostly used by malwares.
Information: Uses functions commonly found in keyloggers
             Has Internet access capabilities
             Uses Microsoft's cryptographic API
```

Manalyze takes care of displaying this information to the user when all the plugins have run, and you shouldn't worry about it unless you want to extend the application so it supports a new output format.

Here is how to insert data inside your `Result`:

Threat level

`set_level` and `raise_level` modify a result's threat level. The only difference between the two is that `set_level` will always overwrite the previous value, while `raise_level` will only store it if the previous one was "lower". The following threat levels are available:

- `SAFE`: the plugin has good reason to believe that the input file is not hostile.
- `NO_OPINION`: the plugin cannot decide whether the input file is malicious or not. Use this threat level if you have gathered information worth mentioning, but which doesn't imply that a program could be malware. For instance, using cryptography is something the user probably wants to know, but containing MD5 constants does not make a program malware.

- `SUSPICIOUS`: use this one if the input file has characteristics that most legitimate programs don't have (i.e. not all packed applications are malware, but it's certainly a sign).
- `MALICIOUS`: this threat level should be used when the plugin thinks that the PE file is malware with a high degree of certainty, like when a ClamAV signature matches it.

By default, if no threat level is specified, a value of `NO_OPINION` will be assumed.

Sample usage:

```
pResult res = create_result();
// do some tests
if (bad_things) {
    res->set_level(MALICIOUS);
}
// do more tests
if (other_things) {
    res->raise_level(SUSPICIOUS); // Threat level will not decrease if it was␣
↪MALICIOUS before.
}
// do even more tests
if (actually_ok) {
    res->set_level(SAFE); // If reached, threat level will be set to SAFE regardless␣
↪of the previous value.
}
```

Summary

Use the `set_summary` method to edit the result's summary. There can only be one, so any subsequent calls will overwrite the previous value. Note that the summary is optional and you don't have to set a value if you don't feel the need to.

Sample usage:

```
pResult res = create_result();
res->set_summary("The PE is possibly packed.");
```

Information

Information can (and must) be added to the result through the `add_information` method. If a result contains no information, Manalyze will assume that it has nothing to report and no output will be generated (even if a threat level or a summary has been set). You may add as many pieces of data as you like, but there is no way to remove one that was already inserted. Finally, the order in which the information is pushed will be preserved.

The `add_information` function, or rather set of functions, allow plugin writers to create complex data structures. Let's look at some examples:

```
pResult res = create_result();
res->add_information("Some textual information added to the result.");

res->add_information("key", "value");

std::vector<std::string> data;
data.push_back("One");
data.push_back("Two");
data.push_back("Three");
res->add_information("A list of strings", data);
```

This code generates the following output when using the JSON formatter:

```
"Plugins": {
    "helloworld": {
        "level": 1,
        "plugin_output": {
            "info_0": "Some textual information added to the result.",
            "key": "value"
            "A list of strings": [
                "One",
                "Two",
                "Three"
            ]
        }
    }
}
```

Internally, all the result data is stored as key-value pairs; if you don't provide a key, Manalyze will generate one automatically which will be omitted whenever possible. Here is the same result presented by the default formatter (when printing human-readable results)

```
Some textual information added to the result.
key: value
A list of strings: One
                   Two
                   Three
```

2.3.4 PE objects

Now that we know how to create results, we will look more closely at the `analyze` method. This is where you should write all your plugin's logic. Here is how it's declared:

```
pResult analyze(const mana::PE& pe);
```

It's return type has been covered already, but what about the argument? This `PE` object is all the plugin has to work with, but it contains all the information gathered from the input file's structure. Let's look at some examples:

DOS Header

The DOS header can be retreived through the `get_dos_header()` function:

```
auto pdos = pe.get_dos_header();
if (pdos != nullptr) {
    std::cout << pdos->e_cblp << std::endl;
}
```

The return value is a pointer to an instance of the following structure, which matches the Windows standard:

```
typedef struct dos_header_t
{
    boost::uint8_t  e_magic[2];
    boost::uint16_t e_cblp;
    boost::uint16_t e_cp;
    boost::uint16_t e_crlc;
    boost::uint16_t e_cparhdr;
```

(continues on next page)

(continued from previous page)

```
    boost::uint16_t e_minalloc;
    boost::uint16_t e_maxalloc;
    boost::uint16_t e_ss;
    boost::uint16_t e_sp;
    boost::uint16_t e_csum;
    boost::uint16_t e_ip;
    boost::uint16_t e_cs;
    boost::uint16_t e_lfarlc;
    boost::uint16_t e_ovno;
    boost::uint16_t e_res[4];
    boost::uint16_t e_oemid;
    boost::uint16_t e_oeminfo;
    boost::uint16_t e_res2[10];
    boost::uint32_t e_lfanew;
} dos_header;
```

PE Header

The `get_pe_header()` function can be used to query the PE header:

```
auto ppe_header = pe.get_pe_header();
if (ppe_header != nullptr && ppe_header->NumberofSections > 4) {
    // ...
}
```

The return value is a pointer to an instance of the following structure, which matches the Windows standard:

```
typedef struct pe_header_t
{
    boost::uint8_t  Signature[4];
    boost::uint16_t Machine;
    boost::uint16_t NumberofSections;
    boost::uint32_t TimeDateStamp;
    boost::uint32_t PointerToSymbolTable;
    boost::uint32_t NumberOfSymbols;
    boost::uint16_t SizeOfOptionalHeader;
    boost::uint16_t Characteristics;
} pe_header;
```

Optional Header

If you need to access data contained in the PE optional header, `get_image_optional_header()` is the function you should use:

```
auto popt = pe.get_image_optional_header();
if (popt != nullptr && popt->Magic ==0x10b) {
    // ...
}
```

The return value is a pointer to an instance of the following structure, which matches the Windows standard:

```
typedef struct image_optional_header_t
{
```

(continues on next page)

(continued from previous page)

```
    boost::uint16_t Magic;
    boost::uint8_t  MajorLinkerVersion;
    boost::uint8_t  MinorLinkerVersion;
    boost::uint32_t SizeOfCode;
    boost::uint32_t SizeOfInitializedData;
    boost::uint32_t SizeOfUninitializedData;
    boost::uint32_t AddressOfEntryPoint;
    boost::uint32_t BaseOfCode;
    boost::uint32_t BaseOfData;
    boost::uint64_t ImageBase;
    boost::uint32_t SectionAlignment;
    boost::uint32_t FileAlignment;
    boost::uint16_t MajorOperatingSystemVersion;
    boost::uint16_t MinorOperatingSystemVersion;
    boost::uint16_t MajorImageVersion;
    boost::uint16_t MinorImageVersion;
    boost::uint16_t MajorSubsystemVersion;
    boost::uint16_t MinorSubsystemVersion;
    boost::uint32_t Win32VersionValue;
    boost::uint32_t SizeOfImage;
    boost::uint32_t SizeOfHeaders;
    boost::uint32_t Checksum;
    boost::uint16_t Subsystem;
    boost::uint16_t DllCharacteristics;
    boost::uint64_t SizeofStackReserve;
    boost::uint64_t SizeofStackCommit;
    boost::uint64_t SizeofHeapReserve;
    boost::uint64_t SizeofHeapCommit;
    boost::uint32_t LoaderFlags;
    boost::uint32_t NumberOfRvaAndSizes;
    image_data_directory directories[0x10];
} image_optional_header;
```

Sections

You can iterate on the input file's sections using the `get_sections()` function:

```
auto psections = pe.get_sections();
if (psections != nullptr)
{
    for (auto it = psections->begin() ; it != psections->end() ; ++it) {
        // ...
    }
}
```

The return value is a shared vector of Section objetcts, which are described later on this page.

Imports

In Manalyze, looking up imports is a two-step process. You usually query the list of DLLs imported by the PE first, then look up particular functions imported in a given DLL. Here is how you would list all the imported DLLs for a PE, using `get_imported_dlls` and `get_imported_functions`:

```
auto dlls = pe.get_imported_dlls();
if (dlls == nullptr) {
    return;
}
for (auto dll = dlls->begin() ; dll != dll->end() ; ++dll)
{
    auto functions = pe.get_imported_functions(dll);
    if (functions == nullptr) {
        continue;
    }
    std::cout << dll << ":" << std::endl;
    for (auto f = functions->begin() ; f != functions->end() ; ++f) {
        std::cout << "\t" << f << std::endl;
    }
}
```

You can also use the `find_imports` and `find_imported_dlls` function if you're looking for something specific. For instance:

```
auto dlls = pe.find_imports("WS2_32.dll", false);
```

... will return all shared libraries imported by the PE matching the regular expression given as the first argument. The second argument controls whether the regular expression is case sensitive and defaults to false when omitted.

auto functions = pe.find_imports(".*basic_ostream.*", "MSVCP\d{3}.dll|KERNEL32.dll", false);

... where the first argument is a regular expression matching the functions to look for, the second one is a regular expression matching the DLLs to search, and the third one is whether the regular expression is case sensitive. You can omit the latter two to look for the requested functions in any DLL with a case insensitive expression:

```
auto functions = pe.find_imports(".*bAsIc_OsTrEaM.*"); // Will search in any DLL,␣
↪case insensitive
```

Finally, if you're interested in looking into the underlying structures, `pe.get_imports` returns `ImportedLibrary` objects which give direct access to the `IMAGE_IMPORT_DESCRIPTOR` and `IMPORT_LOOKUP_TABLE`.

Exports

You can sift through exported functions with `get_exports`:

```
auto pexports = pe.get_exports();
if (pexports != nullptr)
{
    for (auto export = pexports->begin() ; export != pexports->end() ; ++export) {
        std::cout << export->Name << " at ordinal " << export->Ordinal << std::endl;
    }
}
```

The function returns a shared vector containing pointers to instances of the following structure:

```
typedef struct exported_function_t
{
    boost::uint32_t Ordinal;
    boost::uint32_t Address;
    std::string        Name;
```

(continues on next page)

(continued from previous page)

```
    std::string        ForwardName;
} exported_function;
```

Resources

It s possible to iterate through the input file's resources with the `get_resources()` function:

```
auto resources = pe.get_resources();
if (resources != nullptr)
{
    for (auto r = resources->begin() ; r != resources->end() ; ++r) {
        // ...
    }
}
```

The return value is a shared vector of Resource objetcts, which are described later on this page.

Debug Information

If debug information is present in the binary, you can access it through the `get_debug_info` function:

```
auto pdebug = pe.get_debug_info();
if (pdebug != nullptr)
{
    for (auto d = pdebug->begin() ; d != pdebug->end() ; ++d) {
        // do something with d->TimeDateStamp
    }
}
```

The function returns a shared vector containing pointers to instances of the following structure:

```
typedef struct debug_directory_entry_t
{
    boost::uint32_t    Characteristics;
    boost::uint32_t    TimeDateStamp;
    boost::uint16_t    MajorVersion;
    boost::uint16_t    MinorVersion;
    boost::uint32_t    Type;
    boost::uint32_t    SizeofData;
    boost::uint32_t    AddressOfRawData;
    boost::uint32_t    PointerToRawData;
    std::string        Filename;
} debug_directory_entry;
```

Thread Local Storage

If TLS callbacks are defined in the binary, you can look them up with `get_tls`:

```
auto ptls = pe.get_tls();
if (tls == nullptr) {
        return; // No TLS callbacks or failed to parse them.
}
```

(continues on next page)

(continued from previous page)

```
for (auto it = tls->Callbacks.begin() ; it != tls->Callbacks.end() ; ++it) {
        std::cout << "Callback address: 0x" << std::hex << *it);
}
```

The object returned by this function is a pointer to an instance of the following structure:

```
typedef struct image_tls_directory_t
{
        boost::uint64_t                                 StartAddressOfRawData;
        boost::uint64_t                                 EndAddressOfRawData;
        boost::uint64_t                                 AddressOfIndex;
        boost::uint64_t                                 AddressOfCallbacks;
        boost::uint32_t                                 SizeOfZeroFill;
        boost::uint32_t                                 Characteristics;
        std::vector<boost::uint64_t>                    Callbacks; // Non-standard!
} image_tls_directory;
```

It closely resembles the original IMAGE_TLS_DIRECTORY structure, but with a list of all the callback addresses already parsed and stored in the `Callbacks` vector for your convinience.

Load Configuration

You can query the load configuration of the PE with the following function:

```
auto pconfig = pe.get_config();
if (pconfig != nullptr && config->SecurityCookie == 0) {
        std::cout << "/GS seems to be disabled!" << std::endl;
}
```

The structure returned by this function mirrors the one defined in the MSDN:

```
typedef struct image_load_config_directory_t
{
        boost::uint32_t Size;
        boost::uint32_t TimeDateStamp;
        boost::uint16_t MajorVersion;
        boost::uint16_t MinorVersion;
        boost::uint32_t GlobalFlagsClear;
        boost::uint32_t GlobalFlagsSet;
        boost::uint32_t CriticalSectionDefaultTimeout;
        boost::uint64_t DeCommitFreeBlockThreshold;
        boost::uint64_t DeCommitTotalFreeThreshold;
        boost::uint64_t LockPrefixTable;
        boost::uint64_t MaximumAllocationSize;
        boost::uint64_t VirtualMemoryThreshold;
        boost::uint64_t ProcessAffinityMask;
        boost::uint32_t ProcessHeapFlags;
        boost::uint16_t CSDVersion;
        boost::uint16_t Reserved1;
        boost::uint64_t EditList;
        boost::uint64_t SecurityCookie;
        boost::uint64_t SEHandlerTable;
        boost::uint64_t SEHandlerCount;
} image_load_config_directory;
```

Delay Load Table

For PE files which have delayed imports, the `DELAY_LOAD_DIRECTORY_TABLE` can be retreived through `get_delay_load_table`:

```
auto dldt = pe.get_delay_load_table();
if (dldt == nullptr) {
        return; // No delayed imports.
}
std::cout << dldt->NameStr << " is delay-loaded!" << std::endl;
```

The function returns a pointer to the following structure:

```
typedef struct delay_load_directory_table_t
{
        boost::uint32_t Attributes;
        boost::uint32_t Name;
        boost::uint32_t ModuleHandle;
        boost::uint32_t DelayImportAddressTable;
        boost::uint32_t DelayImportNameTable;
        boost::uint32_t BoundDelayImportTable;
        boost::uint32_t UnloadDelayImportTable;
        boost::uint32_t TimeStamp;
        std::string             NameStr; // Non-standard!
} delay_load_directory_table;
```

RICH Header

The RICH header can be can be obtained with the `get_rich_header` function:

```
auto rich = pe.get_rich_header();
if (rich == nullptr) {
        return; // No RICH header.
}
std::cout << "XOR key: " << rich->xor_key << std::endl;
std::cout << "File offset: " << rich->file_offset << std::endl;
for (auto v : rich->values)     {
        std::cout << "Type: " << std::get<0>(v) << " - Prodid: " << std::get<1>(v) <<
↪" - Count: " << std::get<2>(v) << std::endl;
}
```

As there is no official documentation for this structure, it is defined like this in Manalyze:

```
typedef struct rich_header_t
{
        boost::uint32_t xor_key;
        boost::uint32_t file_offset;
        // Structure : id, product_id, count
        std::vector<std::tuple<boost::uint16_t, boost::uint16_t, boost::uint32_t> >␣
↪values;
} rich_header;
```

The *file_offset* field is the absolute position in bytes of the structure in the file (usually `0x80`). For more information regarding the origin of this structure and what information is contained in it, you can consult this article.

Miscellaneous

`pe.get_filesize()` returns the size of the input file in bytes.

`pe.get_architecture()` returns either `PE::x86` or `PE::x64` depending on the program's target architecture.

`pe.rva_to_offset(boost::uint64_t rva)` translates a relative virtual address into a file offset.

`pe.get_raw_bytes(size_t size)` returns the `size` first raw bytes of the file. If `size` is omitted, every byte from the file is returned:

```
auto bytes = pe.get_raw_bytes(1000);
for (auto it = bytes->begin() ; it != bytes->end() ; ++it) {
        // Iterate on the bytes...
}
// Or access them directly:
if ((*bytes)[0] == 'M' && &(*bytes)[1] == 'Z') { ... }
```

`pe.get_overlay_bytes(size_t size)` returns the `size` first bytes of the overlay data of the PE. If `size` is omitted, every byte from the overlay data is returned; and if the file contains no such data, `nullptr` is returned.

`nt::translate_to_flag` and `nt::translate_to_flags` are two functions that come in handy when you need to expand flags (i.e. the `Characteristics` field of many structures). Use the first function for values which translate into a single flag, and the second one for values which are composed of multiple ones:

```
auto pType = nt::translate_to_flag(ppe_header->Machine, nt::MACHINE_TYPES);
if (pType != nullptr) {
    std::cout << "Machine type: " << *pType << std::endl;
}
```

The first argument is of course the value to translate, while the second is a map describing all the flags. Dictionaries relevant to PE structures can be found in `manape/nt_values.cpp` in the `nt` namespace. You can also define your own like this:

```
import "manape/pe.h"

const nt::flag_dict MY_DICT = boost::assign::map_list_of("Value 1", 0x0)
                                                         ("Value 2", 0x1)
                                                         // ...
                                                         ("Value F", 0xF);
```

Results are returned as a shared string or a shared vector of strings respectively.

2.3.5 Section objects

Section objects represent sections of a PE executable. They are very close to the structures defined in the norm, but have been enriched with a couple of utility functions.

`get_name`, `get_virtual_size`, `get_virtual_address`, `get_size_of_raw_data`, `get_pointer_to_raw_data`, `get_pointer_to_relocations`, `get_pointer_to_line_numbers`, `get_number_of_relocations`, `get_number_of_line_numbers` and `get_characteristics` are simple accessors to all the standard information describing a section.

In addition, a `get_entropy` function was added to determine the entropy of a section.

Finding a section

A `find_section` function is available to locate a section based on a relative virtual address (RVA):

```
mana::image_optional_header ioh = *pe.get_image_optional_header();
mana::pSection sec = mana::find_section(ioh.AddressOfEntryPoint, *pe.get_sections());
if (sec != nullptr) {
        std::cout << "Found section: " << *sec->get_name() << std::endl;
}
```

The first argument is a RVA which is contained in the section to locate, while de second one is a list of candidate sections (it will almost always be the return value of `pe.get_sections`).

Conversely, you can check whether an address belongs to a specific section with the following function:

```
bool is_address_in_section(boost::uint64_t rva, mana::pSection section, bool check_
↪raw_size = false);
```

The last argument can be used to perform the check by taking the raw size of the section into account instead of the virtual size, which may be useful for some malformed PEs.

Accessing the raw bytes

If you need to perform some processing on the section's bytes, use `get_raw_data()`. Be aware that the whole section will be loaded in memory, so you may encounter problems when processing very big files:

```
mana::pSection sec = mana::find_section(ioh.AddressOfEntryPoint, *pe.get_sections());
if (sec != nullptr)
{
        mana::shared_bytes bytes = sec->get_raw_data();
        for (auto it = bytes->begin() ; it != bytes->end() ; ++it)
        {
                // ...
        }
}
```

2.3.6 Resource objects

In Manalyze, PE resources are represented as Resource objects that can be manipulated similarly to Sections. `get_name`, `get_type`, `get_language`, `get_codepage`, `get_size`, `get_id` and `get_offset` provide access to basic information and `get_entropy` calculates the entropy of the resource.

Accessing the underlying resource

Most of the time, you'll want to look at the actual resource bytes. A `get_raw_bytes` function is provided and can be used just like the one described above in the context of sections. Some PE resources however have a well-known structure and can be converted into C++ objects to be reused immediately. This is where the function `template <class T> T interpret_as();` comes in. Depending on the template parameter, here are the resource types you can handle:

- `pString` for `RT_MANIFEST` resources. The contents of the PE manifest are returned as a shared string:

```
if (*resource->get_type() == "RT_MANIFEST")
{
        pString rt_manifest = r->interpret_as<pString>();
        if (rt_manifest != nullptr) {
                std::cout << "Manifest contents: " << *rt_manifest << std::endl;
```

(continues on next page)

(continued from previous page)

```
        }
}
```

- `const_shared_string` for `RT_STRING`: the strings contained in the resource are returned as a shared vector.

```
        if (*resource->get_type() != "RT_STRING") {
                return;
        }
        auto string_table = resource->interpret_as<const_shared_strings>();
        std::wcout << L"Dumping a RT_STRING resource:" << std::endl;
        for (auto it = string_table->begin() ; it != string_table->end() ; ++it) {
                std::wcout << *it << std::endl;
        }

The strings returned are UTF-8 encoded.
```

- `pgroup_icon_directory` for `RT_GROUP_ICON` and `RT_GROUP_CURSOR`. Because of the way icons and cursors are stored in resources, an additional function `reconstruct_icon` was added to recreate a valid ICO file. Here is how you'd do it:

```
if (*res->get_type() == "RT_GROUP_ICON" || *res->get_type() == "RT_GROUP_CURSOR")
↪{
        ico_file = reconstruct_icon((*it)->interpret_as<pgroup_icon_directory>(),␣
↪*pe.get_resources());
}
FILE* f = fopen("icon.ico", "wb");
if (f == nullptr) {
        return;
}
fwrite(&ico_file[0], 1, ico_file->size(), f);
fclose(f);
```

- `pbitmap` for a `RT_BITMAP`.
- `pversion_info` for `RT_VERSIONINFO` resources.
- And finally `shared_bytes` for any resource type, which will behave exactly like `get_raw_data`.

All these functions will return null pointers if for some reason the resource cannot be interpreted as the requested type.

Extracting resources

Utility functions are provided to extract resources into a file. Use *extract* or *icon_extract* depending on the resource type - the reason why two separate functions were written is that icon data may be spread over multiple resources, therefore a list of all the resources of the PE must be provided to reconstruct them properly. Here is how you can extract resources in the general case:

```
auto resources = pe.get_resources();
for (auto it = resources->begin() ; it != resources->end() ; ++it)
{
        bool res;
        if (*(*it)->get_type() == "RT_GROUP_ICON" || *(*it)->get_type() == "RT_GROUP_
↪CURSOR") {
                res = (*it)->icon_extract(*(*it)->get_name() + ".ico", resources);
        }
```

(continues on next page)

(continued from previous page)

```
        else {
                res = (*it)->extract(*(*it)->get_name() + ".bin");
        }

        if (!res) {
                std::cerr << "Resource extraction failed :(" << std::endl;
        }
}
```

Note: What's up with all these pointers?

Manalyze is built statically on Windows for a number of reasons which go beyond the scope of this documentation. This causes some issues when functions are called across DLLs, issues which can only be resolved through smart pointers ensuring that a module which allocated an object will be the one to free it. This ends up making all the interfaces a little more complex by having pointers everywhere.

2.3.7 Using the configuration file

If you want to give users additional control on the plugin's behavior, you can let them pass arguments through the configuration file, `manalyze.conf`. For instance, this mechanism is used to provide a VirusTotal API key without having to hard-code it into the software. Each plugin has access to a protected `_config` variable through its parent class. It is a simple map between strings. Here is an example of how it is used, taken from the packer detection plugin:

```
unsigned int min_imports;
// Check that a value was set in the configuration, otherwise use a default one.
if (_config == nullptr || !_config->count("min_imports")) {
        min_imports = 10;
}
else
{
        try {
                min_imports = std::stoi(_config->at("min_imports"));
        }
        catch (std::invalid_argument) // In case someone writes "packer.min_imports =␣
↪ABC"
        {
                PRINT_WARNING << "Could not parse packer.min_imports in the␣
↪configuration file." << std::endl;
                min_imports = 10;
        }
}
```

Using variables from the configuration doesn't require any additional work: lines added to the configuration files are automatically parsed and provided to the target plugin. Let's assume you add the following lines to the configuration file:

```
helloworld.msg = Hello World!
# Lines starting with # are comments, the parser ignores them.
helloworld.val = 0
```

... then the `_config` variable of a plugin whose name (as reported by the `get_id` function) is `helloworld` would contain the following map:

```
{
        "msg":  "Hello World!",
        "val":  "0" // Careful! That's still a string!
}
```

Note: The configuration is only initialized before calling the `analyze` method. This means that you won't be able to reference your plugin's configuration from its constructor.

2.3.8 Anything missing?

If you are trying to do something but still can't figure out how to do it, be sure to get in touch with the project's maintainer via GitHub. Possible problems might include:

- API not providing access to data you need.
- Some part of the PE being parsed incorrectly or insufficiently.
- The documentation not being clear enough on a particular topic.

2.4 Writing Yara rules

This section is dedicated to the intricacies of writing Yara rules which can be used by Manalyze.

2.4.1 Introduction

Because Manalyze already includes an (hopefully) efficient PE parser, it was deemed unnecessary to rely on the one that is provided with Yara. The Yara engine provided with Manalyze was essentially stripped down to the library code and contains none of the plugins provided with the original distribution. Custom C++ wrappers were also added to the project. All the modifications to the code may be found on GitHub.

For this reason, Yara rules relying on the original PE module will not work with Manalyze ; they need to be modified so they rely on the one provided to Yara by the tool.

Note: The functionnalities provided by this module are added on a need basis. If you need additional data exposed, please create an issue on GitHub!

2.4.2 Supported commands

All scripts relying on Manalyze's PE module must start by importing it with the `import "manape"` directive.

- The entry point of the executable is designated by `manape.ep`.
- The number of sections is exposed through `manape.num_sections`.
- For each section, you can access the start address and the size with `manape.section[i].start` and `manape.section[i].size`, `i` being the zero-based index of the section.
- You can scan the `VERSION_INFO` resource with `manape.version_info.start` and `manape.version_info.size`.

- The authenticode signature of the binary can be located through `manape.authenticode.start` and `manape.authenticode.size`.

2.4.3 Sample rule

```
import "manape"

rule D_Win_dot_Trojan_dot_Patched_dash_300
{
        meta:
                signature = "Win.Trojan.Patched-300"
        strings:
                $a0 = { 837c24080175 }
                $a1 = { 726f6341c745e064647265 }
                $a2 = {␣
↪43006f006d00700061006e0079004e0061006d006500000000004d006900630072006f0073006f006600740020004300
↪}
        condition:
                $a0 at manape.ep and $a1 and $a2 in (manape.version_info.start ..␣
↪manape.version_info.start + manape.version_info.size)
}
```

2.5 Reusing the PE parser

2.5.1 Embedding the code

This section will explain how you can take the PE parser (ManaPE) out of Manalyze and re-use it in another project.

Let's start by writing some sample code that would read a PE file using Manalyze's parser:

```
#include <iostream>
#include "manape/pe.h"

int main(int argc, char** argv)
{
        mana::PE pe("file.exe");
        if (pe.is_valid()) { // Always check this.
                std::cout << "File parsed successfully: " << *pe.get_path() <<␣
↪std::endl;
        }
        else
        {
                std::cout << "The file is invalid!" << std::endl;
                return 1;
        }

        // Do stuff with the PE
        auto sections = pe.get_sections();
        for (auto it = sections->begin() ; it != sections->end() ; ++it) {
                std::cout << *(*it)->get_name() << std::endl;
        }
        // ...
```

(continues on next page)

(continued from previous page)

```
        return 0;
}
```

For this to compile, you'll have to grab ManaPE's code and put it inside your project. you need both the `manape` and `include/manape` folders.

```
~/code/project$ mkdir include
~/code/project$ cp -r [...]/Manalyze/manape/ . && cp -r [...]/Manalyze/include/manape/
↪ include/
```

You don't have to follow the same folder structure, it's only given as an example. Then, assuming you copied the previous code in `main.cpp`, the only thing left to do is to compile everything:

```
~/code/project$ g++ main.cpp manape/*.cpp -lboost_system -lboost_regex -Iinclude -
↪std=c++11
~/code/project$ ./a.out
File parsed successfully: file.exe
.text
.rdata
.data
.rsrc
```

Obviously, you'll want to write a Makefile or use CMake, but this should be enough to get you started. If you need detailed information on available methods that you can use from here, please see this section on *PE objects*.

2.5.2 Reusing binaries

On Linux

Depending on your use-case, you may alternatively re-use the shared libraries which are distributed and/or generated with Manalyze and its build system.

In that case, you still have to include the header files in your project as described above (except you only need the `[...]/Manalyze/include/manape/` directory). You also need to copy the shared objects:

```
~/code/project$ mkdir include lib
~/code/project$ cp -r [...]/Manalyze/include/manape/ include/
~/code/project$ cp [...]/Manalyze/bin/*.so lib/
```

Subsequently, add `-Llib` and `-lmanape -lmanacommons` to your compilation flags to indicate that the compiler should link against those libraries.

On Windows

Linking against DLLs requires a little more work on Windows. First, copy Manalyze's header files in your project directory as described above. Also put Manalyze's DLLs somewhere in the PATH of your project (likely the folder where your executable will be generated). You'll need `manape.dll`, `manacommons.dll`, `hash-library.dll` and `yara.dll`.

Sadly, Visual Studio is *only* capable of linking against `.lib` files, even if the code will *in fine* be found in a DLL. Those files are generated when Manalyze is built but are not distributed with the program - this means that you have to checkout Manalyze's source code from GitHub and build it manually. Hopefully, this should be as simple as this:

```
$ git clone https://github.com/JusticeRage/Manalyze.git
$ cd Manalyze
$ cmake .
```

...Then use Visual Studio to build everything. Following that, you will find a few `.lib` files in `[...]\Manalyze\Debug\` or `[...]\Manalyze\Release\` (use whichever matches your build profile). Copy `*.lib` to a `lib` folder in your project directory and configure VS so that they will be taken into account. This involves:

- Adding the `lib` folder to `Library Directories` under `VC++ Directories`.
- Specifying `manape.lib` and `manacommons.lib` in `Linker > Input > Additional Dependencies`

From there, you should be able to write code relying on the PE parser!

CHAPTER 3

Interfacing with Manalyze

3.1 Endpoints

If you're working on a tool that could benefit from integrating with Manalyze, there are a few ways you can obtain results from the tool. The most straightforward one is to parse the output directly:

```
manalyze [sample] --dump=... --plugins=... --output=json
```

If you are not willing or able to use Manalyze on your local machine, the web portal can provide the same results. You will, however, need to contact the project's maintainer to obtain an API key.

```
import requests

# Submit a file
f = {'file': open("sample.exe", "rb")}
data = {'api_key': "[Your API key]"}
r = requests.post("https://manalyzer.org/api/submit", files=f, data=data)
print r.text

# Get a report for an existing file (no API key required)
r = requests.get("https://manalyzer.org/json/1804821148ae7c305d0e5d3463bcbd67")
print r.text
```

3.2 JSON structure

In both cases, you'll obtain a JSON document which represents the report produced by Manalyze. Their high-level structure is as follows:

```
user@machine:~/samples$ manalyze -ojson file1 /tmp/file2
{
    "/home/user/samples/file1": {
```

(continues on next page)

(continued from previous page)

```
        // Report for file1
    }
    "/tmp/file2": {
        // Report for file2
    }
}
```

At the root of the document, you'll find an entry for each file analyzed. If the analysis could not complete successfully, no object will be added to the document root. In the rest of the documentation, only reports for a single file will be used, as they all have the exact same structure.

3.2.1 Dump of the PE

The reports can be viewed as the sum of two parts. First, all the information pertaining to the file format that Manalyze would print through the `--dump` option. Here is what that part of the document may look like:

```
{
        "ab35c68e263bb4dca6c11e16cd7fb9d8": {
            "Summary": {
                "Compilation Date": "2017-Nov-16 22:05:22",
                "Detected languages": [
                    "English - United States"
                ],
                "CompanyName": "Sysinternals - www.sysinternals.com"
                // ...
            },
            "DOS Header": {
                "e_magic": "MZ",
                "e_cblp": 144
                // ...
            },
            "Sections": {
                ".text": {
                    "MD5": "c151016c0929a571e7a3882e3c292524",
                    "NumberOfRelocations": 0,
                    "Characteristics": [
                        "IMAGE_SCN_CNT_CODE",
                        "IMAGE_SCN_MEM_EXECUTE",
                        "IMAGE_SCN_MEM_READ"
                    ],
                    "Entropy": 6.60464
                    // ...
            },
            "Imports": {
                "WINTRUST.dll": [
                    "CryptCATEnumerateMember",
                    "CryptCATEnumerateCatAttr"
                    // ...
                ],
                "VERSION.dll": [
                    "GetFileVersionInfoSizeW",
                    "VerQueryValueW",
                    "GetFileVersionInfoW"
                ]
                // ...
```

(continues on next page)

(continued from previous page)

```
            },
            "Resources": {
                "1": {
                    "Type": "RT_VERSION",
                    "Language": "English - United States",
                    "SHA1": "48cf205c2a63018aa56267f95490b0da0156aa6d"
                    // ...
                }
                // ...
            },
            "Hashes": {
                "MD5": "ab35c68e263bb4dca6c11e16cd7fb9d8"
                // ...
            }
            // ...
}
```

This document has been trimmed down a for readability purposes, but links to complete reports are provided below. Here is the list of possible keys you can encounter:

- `Summary` (example 1)
- `DOS Header` (example 1)
- `PE Header` (example 1)
- `Image Optional Header` (example 1)
- `Sections` (sample with unprintable section names)
- `Imports` (example 2, sample with no imports, imports with name mangling)
- `Delayed Imports` (example 3)
- `Exports` (example 4)
- `Resources` (example 2, sample with no resources)
- `Version Info` (example 5)
- `Debug Info` (example 6)
- `TLS Callbacks` (example 5)
- `Load Configuration` (example 6)
- `StringTable` (example 7)
- `RICH Header` (example 1)
- `Hashes` (example 1)
- `Plugins` (see below)

You can expect at least the `Summary`, `DOS Header` and `DOS Header` to be present in any valid report.

You'll notice that JSON documents from the web service may contain an additional `Error` section that contains any message that Manalyze has printed on `stderr`. This will not be done automatically with Manalyze's JSON output, so you should capture `stderr` manually if you're interested in errors and warnings.

3.2.2 Plugins

The reports also contain a whole section dedicated to the output of any plugin called by Manalyze. As plugins are more dynamic by nature (users may have downloaded some from third-parties or developed their own), it is not possible to provide an exhaustive list of possible sections. However, all plugin results adhere to the same structure:

```
"plugin name": {
        "level": 3,
        "plugin_output": {
            "key 1": [
                "value 1",
                "value 2"
                // ...
            ],
            "key 2": "value 3"
            // ...
        },
        "summary": "A single string"
}
```

The `level` is an integer value which describes the threat level reported by the plugin. Four values are possible:

- 0: The plugin indicates that the file is harmless (`SAFE`).
- 1: The information gathered is interesting but does not indicate that the file is either goodware or malware (`NO_OPINION`).
- 2: The file contains elements that can be indicative of malicious behavior (`SUSPICIOUS`).
- 3: The sample exhibits characteristics that are generally found in malicious programs only. (`MALICIOUS`).

Keep in mind that each plugin has a very narrow scope and that it's not unexpected to have conflicting plugin verdicts (for instance, a PE file which is both packed and signed would be flagged as safe by the authenticode plugin and malicious by the packer plugin).

Then, the `plugin_output` is an optional series of key-value pairs that can be freely filled by the plugin. Note that the value can be of any type (string, integer, or even lists of strings!). You'll also notice that some keys have a names such as `info_0`. Those names are generated automatically by Manalyze when the plugin doesn't specify one and can be safely ignored for any display purposes. Finally, the `summary` is a high-level description of the plugin's verdict.

Here is a sample plugin output for `WannaCry`:

```
"Plugins": {
        "compilers": {
            "level": 1,
            "plugin_output": {
                "info_0": "Microsoft Visual C++ 6.0 - 8.0",
                "info_1": "Microsoft Visual C++",
                "info_2": "Microsoft Visual C++ v6.0",
                "info_3": "Microsoft Visual C++ v5.0/v6.0 (MFC)"
            },
            "summary": "Matching compiler(s):"
        },
        "strings": {
            "level": 2,
            "plugin_output": {
                "Miscellaneous malware strings": [
                    "cmd.exe"
                ]
            },
```

(continues on next page)

(continued from previous page)

```
            "summary": "Strings found in the binary may indicate undesirable behavior:
↪"
        },
        "findcrypt": {
            "level": 1,
            "plugin_output": {
                "info_0": "Uses constants related to CRC32",
                "info_1": "Uses constants related to AES",
                "info_2": "Microsoft's Cryptography API"
            },
            "summary": "Cryptographic algorithms detected in the binary:"
        },
        "btcaddress": {
            "level": 3,
            "plugin_output": {
                "Contains a valid Bitcoin address": [
                    "115p7UMMngoj1pMvkpHijcRdfJNXj6LrLn",
                    "12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw",
                    "13AM4VW2dhxYgXeQepoHkHSQuy6NgaEb94"
                ]
            },
            "summary": "This program may be a ransomware."
        },
        "imports": {
            "level": 2,
            "plugin_output": {
                "Possibly launches other programs": [
                    "CreateProcessA"
                ],
                "Uses Microsoft's cryptographic API": [
                    "CryptReleaseContext"
                ],
                "Interacts with services": [
                    "CreateServiceA",
                    "OpenServiceA",
                    "OpenSCManagerA"
                ]
                // ...
            },
            "summary": "The PE contains functions most legitimate programs don't use."
        },
        "resources": {
            "level": 2,
            "plugin_output": {
                "info_0": "Resources amount for 98.1255% of the executable."
            },
            "summary": "The PE is possibly a dropper."
        },
        "mitigation": {
            "level": 1,
            "plugin_output": {
                "Stack Canary": "disabled",
                "SafeSEH": "disabled",
                "ASLR": "disabled",
                "DEP": "disabled"
            },
            "summary": "The following exploit mitigation techniques have been detected
↪"
```

(continues on next page)

(continued from previous page)

```
        },
        "virustotal": {
            "level": 3,
            "plugin_output": {
                "Bkav": "W32.WanaCryptBTTc.Worm",
                "MicroWorld-eScan": "Trojan.Ransom.WannaCryptor.A",
                "nProtect": "Ransom/W32.WannaCry.Zen",
                "Paloalto": "generic.ml",
                "ClamAV": "Win.Trojan.Agent-6312832-0",
                "Kaspersky": "Trojan-Ransom.Win32.Wanna.zbu",
                "BitDefender": "Trojan.Ransom.WannaCryptor.A",
                // ...
            },
            "summary": "VirusTotal score: 58/62 (Scanned on 2017-07-08 14:55:28)"
        }
}
```

Source

3.2.3 Additional JSON samples:

If you need additional JSON documents to test your Manalyze integration, head to Manalyzer and find a report that interests you. Just change the URL from:

```
https://manalyzer.org/report/[md5]
```

...to...

```
https://manalyzer.org/json/[md5]
```

...and you'll be presented with the source JSON document.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search