Writing a C++ Yara Agent

 mez0.cc/posts/yaraengine

Table of Contents

Introduction

Whilst writing PreEmpt, one of the requirements was to make use of Yara to be able to identify malware families. In this blog, I wanted to just go over the development process to make use of the C API available for Yara.

Yara?

From https://virustotal.github.io/yara/:

> YARA is a tool aimed at (but not limited to) helping malware researchers to identify and classify malware samples. With YARA you can create descriptions of malware families (or whatever you want to describe) based on textual or binary patterns. Each description, a.k.a rule, consists of a set of strings and a boolean expression which determine its logic.

Essentially, it allows for rules to be ran over memory, processes, and files to identify malware families. Rules are easy to write, here is an example from the Yara homepage:

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

This isn't a blog on writing Yara rules, but essentially `strings` are the "match" criteria, and `condition` are when to flag the rule. In this case, if any of the `strings` match, then the rule flags as a match.

What this blog will do is write a small agent to run these rules against a specified process ID. So, for that, rules are required. All I did was Google: `cobalt strike yara rules github` and made a `.yar` file containing rules from:

1. https://github.com/Neo23x0/signature-base/blob/master/yara/apt_cobaltstrike.yar
2. https://github.com/jas502n/cs_yara
3. https://github.com/Neo23x0/signature-base/blob/master/yara/apt_cobaltstrike_evasive.yar
4. https://github.com/mgreen27/cobaltstrike-1
5. https://github.com/JPCERTCC/MalConfScan/blob/master/yara/rule.yara
6. https://github.com/Te-k/cobaltstrike

All these rules will be ran against the target process. However, the usage of the tool produced in this blog can be used as a Yara rule tester. The commandline arguments will be:

```
YaraAgent.exe <path to rule> <process id>
```

Let's write some code.

Yara Library Setup

Installing Yaralib

To get Yara's library into Visual Studio, I used vcpkg. Once installed:

```
.\vcpkg install yara
```

Then Yara is available as:

```
#include <yara.h>
```

Initializing the Library

The Yara C API documentation is okay, but it lacks when it comes to actually scanning. So, first thing:

> The first thing your program must do when using *libyara* is initializing the library. This is done by calling the `yr_initialize()` function.

The library needs to be initialized, for me, I like to use C++ namespaces and classes for this. This will become useful as Yara requires an initialize, and a finalize. Which are perfect candidates for C++ constructors and destructors.

Lets add that:

```
namespace Yara
{
    class Manager
    {
    public:
        Manager()
        {
            int init = yr_initialize();
            if (init != ERROR_SUCCESS)
            {
                printf("Initialise failed: %s\n", GetErrorMsg(init).c_str());
                return;
            }
        }

        ~Manager()
        {
            int finalise = yr_finalize();
            if (finalise != ERROR_SUCCESS)
            {
                printf("Finalise failed: %s\n", GetErrorMsg(finalise).c_str());
                return;
            }
        }
    }
}
```

Here, yr_initialize() and yr_finalize() are being called which must be done in the main thread. As these are just in the constructor and destructor, they can just be passively handle by creating the object:

```
Yara::Manager yara = Yara::Manager();
```

Creating a Compiler

The next thing that needs to happen is that the Yara Compiler needs to be created, this can also be added into the constructor by calling the following function:

```
BOOL CreateCompiler()
{
    int create = yr_compiler_create(&compiler);

    if (create == ERROR_SUCCESS)
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}
```

Where `compiler` is a `private`:

```
YR_COMPILER* compiler = NULL;
```

The constructor is now:

```
Manager()
{
    int init = yr_initialize();
    if (init != ERROR_SUCCESS)
    {
        printf("Initialise failed: %s\n", GetErrorMsg(init).c_str());
        return;
    }

    if (CreateCompiler())
    {
        success = TRUE;
    }
    else
    {
        success = FALSE;
    }
}
```

Adding rules

Now its ready to receive a rule file. The path was passed in from the commandline and is literally just that, a file path to a yara file.

The code structure I want for this is:

```
if (yara.AddRuleFromFile(path) == FALSE)
{
    printf("[!] Failed to load %s\n", path.c_str());
    return -1;
}
```

So, lets look at `AddRuleFromFile()`:

```
BOOL AddRuleFromFile(std::string file_name)
{
    FILE* rule_file = NULL;

    int result = fopen_s(&rule_file, file_name.c_str(), "r");
    if (result != ERROR_SUCCESS)
    {
        printf("Failed to open %s: %s\n", file_name.c_str(),
GetErrorMsg(result).c_str());
        return FALSE;
    }

    result = yr_compiler_add_file(compiler, rule_file, NULL, file_name.c_str());
    if (result != ERROR_SUCCESS)
    {
        printf("Failed to add rules from %s: %s\n", file_name.c_str(),
GetErrorMsg(result).c_str());
        return FALSE;
    }

    result = yr_compiler_get_rules(compiler, &rules);

    if (result != ERROR_SUCCESS)
    {
        printf("Failed to get rules from %s: %s\n", file_name.c_str(),
GetErrorMsg(result).c_str());
        return FALSE;
    }

    return TRUE;
}
```

First off, read the file:

```
FILE* rule_file = NULL;

int result = fopen_s(&rule_file, file_name.c_str(), "r");
if (result != ERROR_SUCCESS)
{
    printf("Failed to open %s: %s\n", file_name.c_str(),
GetErrorMsg(result).c_str());
    return FALSE;
}
```

There are three methods for adding rules:

1. yr_compiler_add_file()
2. yr_compiler_add_fd()
3. yr_compiler_add_string()

They are all self-explanatory, but the one used here is `yr_compiler_add_file()`:

```
result = yr_compiler_add_file(compiler, rule_file, NULL, file_name.c_str());
if (result != ERROR_SUCCESS)
{
    printf("Failed to add rules from %s: %s\n", file_name.c_str(),
GetErrorMsg(result).c_str());
    return FALSE;
}
```

The function parameters:

```
int yr_compiler_add_file(
    YR_COMPILER* compiler,
    FILE* file,
    const char* namespace,
    const char* file_name)
```

`namespace` is left to `NULL` because:

> if *namespace* is `NULL` they will be put into the default namespace.

To check this worked, yr_compiler_get_rules() is used. This will either return ERROR_SUCCESS or ERROR_INSUFFICIENT_MEMORY.

At this point, the library is initialised and rules are loaded; time to scan some memory.

Detecting Cobalt Strike

With all that set up, we still don't have anything to scan. But before that, this is the structure I want to get back:

```
std::vector<YaraInfo> matches = yara.ScanProcessMemory(dwPid);
```

A vector of `YaraInfo` where `YaraInfo` is my struct:

```
typedef struct YARAINFO
{
    std::vector<std::string> matched_rules;
    RegionInfo infectedRegion;
} YaraInfo, * PYaraInfo;
```

Which is a vector of rule names, and another struct:

```
typedef struct REGIONINFO
{
    LPVOID pBase;
    LPVOID pAllocation;
    DWORD dwRegion;
    DWORD dwProtect;
    DWORD dwState;
    DWORD dwType;
} RegionInfo, * PRegionInfo;
```

This is another custom struct, and is very similar to MEMORY_BASIC_INFORMATION. However, this code is from PreEmpt which has a much bigger data structure (so I just copied and pasted that and removed some fields).

A few things need to happen before this can be achieved, so lets look at that.

Reading Process Memory Regions

First off, get a `HANDLE`:

```
RAII::Handle hProcess = OpenProcess(PROCESS_READ_FLAGS, FALSE, dwPid);
```

Here, Resource Acquisition Is Initialization is being used to easily handle the, well, `HANDLE`. And `PROCESS_READ_FLAGS` is:

```
#define PROCESS_READ_FLAGS PROCESS_QUERY_INFORMATION | PROCESS_VM_READ
```

For the process, I want to get a vector of every region is a `RegionInfo` struct, like so:

```
std::vector<RegionInfo> regions = GetProcessRegions(hProcess.Get());
```

Here is the function:

```
std::vector<RegionInfo> GetProcessRegions(HANDLE hProcess)
{
    std::vector<RegionInfo> regions;
    MEMORY_BASIC_INFORMATION mbi = {};
    LPVOID offset = 0;

    while (VirtualQueryEx(hProcess, offset, &mbi, sizeof(mbi)))
    {
        offset = (LPVOID)((DWORD_PTR)mbi.BaseAddress + mbi.RegionSize);

        RegionInfo regionInfo;
        regionInfo.pBase = mbi.BaseAddress;
        regionInfo.pAllocation = mbi.AllocationBase;
        regionInfo.dwProtect = mbi.Protect;
        regionInfo.dwRegion = mbi.RegionSize;
        regionInfo.dwState = mbi.State;
        regionInfo.dwType = mbi.Type;
        regions.push_back(regionInfo);
    }
    if (regions.size() == 0)
    {
        ErrorHandler::Show().print_win32error("VirtualQueryEx()");
    }
    return regions;
}
```

Here, VirtualQueryEx() is being used to retrieve information about pages within the virtual address space:

```
SIZE_T VirtualQueryEx(
  [in]           HANDLE                    hProcess,
  [in, optional] LPCVOID                   lpAddress,
  [out]          PMEMORY_BASIC_INFORMATION lpBuffer,
  [in]           SIZE_T                    dwLength
);
```

In order do to that, a `while` loop is used whilst incrementing the offset from 0, by the base address plus the region size. This allows for each page to be incremented over. With each region, we just build out a struct a `push_back` the vector.

For a visual representation of what this struct looks like, Process Hacker has it covered:

Base address	Type	Size	Protect...	Use
0x7ff998eb1000	Image: Commit	576 kB	RX	C:\Windows\System32\user32.dll
0x7ff998de1000	Image: Commit	596 kB	RX	C:\Windows\System32\oleaut32.dll
0x7ff9982e1000	Image: Commit	272 kB	RX	C:\Windows\System32\ws2_32.dll
0x7ff997961000	Image: Commit	2,276 kB	RX	C:\Windows\System32\combase.dll
0x7ff9978d1000	Image: Commit	120 kB	RX	C:\Windows\System32\imm32.dll
0x7ff997821000	Image: Commit	416 kB	RX	C:\Windows\System32\advapi32.dll
0x7ff9976c1000	Image: Commit	468 kB	RX	C:\Windows\System32\SHCore.dll
0x7ff9975d1000	Image: Commit	180 kB	RX	C:\Windows\System32\shlwapi.dll
0x7ff997581000	Image: Commit	60 kB	RX	C:\Windows\System32\gdi32.dll
0x7ff997361000	Image: Commit	900 kB	RX	C:\Windows\System32\rpcrt4.dll
0x7ff9972c1000	Image: Commit	404 kB	RX	C:\Windows\System32\sechost.dll
0x7ff9972b1000	Image: Commit	8 kB	RX	C:\Windows\System32\nsi.dll
0x7ff9971c1000	Image: Commit	336 kB	RX	C:\Windows\System32\msvcp_win.dll
0x7ff9970b1000	Image: Commit	624 kB	RX	C:\Windows\System32\gdi32full.dll

The only thing we don't care about here is the `Use`.

Okay, so thats the memory regions of the process mapped out; lets read them as a buffer.

Reading regions to buffer

The usage here is:

```
std::vector<RegionInfo> regions = GetProcessRegions(hProcess.Get());

for (RegionInfo& regionInfo : regions)
{
    std::vector<std::byte> region = ReadRegionToBuffer(regionInfo, hProcess.Get());
    if (region.empty()) continue;
}
```

For every region, read the region into a vector of bytes. Lets look at `ReadRegionToBuffer()`:

```
std::vector<std::byte> ReadRegionToBuffer(RegionInfo regionInfo, HANDLE hProcess)
{
    if (regionInfo.dwProtect == PAGE_NOACCESS) return std::vector<std::byte>{};

    std::vector<std::byte> buffer(regionInfo.dwRegion);

    BOOL bRead = ReadProcessMemory(hProcess, (LPVOID)regionInfo.pBase, buffer.data(),
regionInfo.dwRegion, NULL);
    if (bRead == FALSE)
    {
        ErrorHandler::Show().print_win32error("ReadProcessMemory()");
    }

    return buffer;
}
```

First off, if the protection is `PAGE_NOACCESS`, leave. Otherwise, create a vector with the size of the region. Then ReadProcessMemory() is used:

```
BOOL ReadProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPCVOID lpBaseAddress,
  [out] LPVOID  lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesRead
);
```

This is quite simple, take in:

1. A `HANDLE` to the process
2. The base address to read from
3. Where to put the read bytes
4. How much **TO** read
5. How much **WAS** read

If it returns `TRUE`, then the bytes were read. When this returns, we check if it actually did:

```
if (region.empty()) continue;
```

Then we cast to `unsigned char` with another check for empties:

```
const unsigned char* buffer = (const unsigned char*)region.data();
int buffer_size = region.size();

if (strlen((char*)buffer) == 0) continue;
```

Scanning Memory Regions

Now that the regions are identified and read into buffers, a `YaraInfo` struct is prepared:

```
YaraInfo yaraInfo;
```

And finally another Yara Library call:

```
int result = yr_rules_scan_mem(rules, buffer, buffer_size, SCAN_FLAGS_PROCESS_MEMORY,
capture_matches, &yaraInfo, 0);
```

This is where the actual magic happens. yr_rules_scan_mem() will scan a memory buffer and will return one of the following:

1. ERROR_SUCCESS
2. ERROR_INSUFFICIENT_MEMORY
3. ERROR_TOO_MANY_SCAN_THREADS
4. ERROR_SCAN_TIMEOUT
5. ERROR_CALLBACK_ERROR
6. ERROR_TOO_MANY_MATCHES

Lets look at the structure of this call:

```
int yr_rules_scan_mem(
    YR_RULES* rules,
    uint8_t* buffer,
    size_t buffer_size,
    int flags,
    YR_CALLBACK_FUNC callback,
    void* user_data,
    int timeout)
```

The first three parameters are clear; the rules we created earlier, a buffer, and a buffer size. The `flags` has the following options:

```
#define SCAN_FLAGS_FAST_MODE                1
#define SCAN_FLAGS_PROCESS_MEMORY           2
#define SCAN_FLAGS_NO_TRYCATCH              4
#define SCAN_FLAGS_REPORT_RULES_MATCHING    8
#define SCAN_FLAGS_REPORT_RULES_NOT_MATCHING 16
```

I'm not completely sure what behavioral differences these flags have, so I stuck with `SCAN_FLAGS_PROCESS_MEMORY`.

The `callback` and `user_data` where the most complicated and took some Googling, shout out to Radare2 for making use of this API call and helping me solve this part.

So, `callback` is the function to run on scans; the code I went with:

```
static int capture_matches(YR_SCAN_CONTEXT* context, int message, void* message_data,
void* user_data)
{
    PYaraInfo yaraInfo = static_cast<PYaraInfo>(user_data);

    if (message == CALLBACK_MSG_RULE_MATCHING)
    {
        YR_RULE* rule = (YR_RULE*)message_data;
        YR_STRING* string;

        yr_rule_strings_foreach(rule, string)
        {
            std::string rule_name = rule->identifier;
            if (VectorContainsStringA(yaraInfo->matched_rules, rule_name) == FALSE)
            {
                yaraInfo->matched_rules.push_back(rule_name);
            }
        }
    }

    return CALLBACK_CONTINUE;
}
```

Note the `user_data` again. What this is, is the structure that you want back. So, when we called the function, `&yaraInfo` was passed. This can then be retrieved inside the callback, allowing a custom struct to be filled:

```
PYaraInfo yaraInfo = static_cast<PYaraInfo>(user_data);
```

In the above, if a rule matches (`CALLBACK_MSG_RULE_MATCHING`), loop over all the rules matched and add them into the `YaraInfo` struct:

```
yaraInfo->matched_rules.push_back(rule_name);
```

The `VectorContainsStringA` is a helper function I use a lot:

```
inline std::string ConvToLowerA(std::string a)
{
    std::transform(a.begin(), a.end(), a.begin(), ::tolower);
    return a;
}

inline BOOL VectorContainsStringA(std::vector<std::string> haystack, std::string
needle)
{
    for (std::string& hay : haystack)
    {
        if (ConvToLowerA(hay) == ConvToLowerA(needle))
        {
            return TRUE;
        }
    }
    return FALSE;
}
```

Once the `yr_rules_scan_mem()` function finishes, and the callback is executed, the `YaraInfo` struct is checked:

```
if (yaraInfo.matched_rules.size() > 0)
{
    yaraInfo.infectedRegion = regionInfo;
    allYaraInfo.push_back(yaraInfo);
}
```

Then the `vector` of `YaraInfo` is returned:

```
return allYaraInfo;
```

ALOT has happened in this one function call:

```
std::vector<YaraInfo> matches = yara.ScanProcessMemory(dwPid);

if (matches.size() == 0)
{
    printf("[!] No Yara matches!\n");
    return -1;
}
```

To recap:

1. All the process memory regions were obtained
2. Read to a buffer
3. Passed to the Yara Library
4. Callback function checked each region for rules and updated the `YaraInfo` struct with matching rules
5. A vector of `YaraInfo` was returned.

Demo

Now that malware is identifed, it can be displayed:

```
int idx = 1;
for (YaraInfo& match : matches)
{
    printf("\\_ Match: %d/%I64u\n", idx, matches.size());
    printf("  | Base Address: 0x%p\n", match.infectedRegion.pBase);
    printf("  | Allocation Address: 0x%p\n", match.infectedRegion.pAllocation);
    printf("  | Page Protection: %ld\n", match.infectedRegion.dwProtect);
    printf("  | Page State: %ld\n", match.infectedRegion.dwState);
    printf("  | Page Type: %ld\n", match.infectedRegion.dwType);
    printf("  | Rules:\n");
    for (std::string& rule : match.matched_rules)
    {
        printf("   - %s\n", rule.c_str());
    }
    idx++;
    printf("\n");
}
```

Conclusion

This was a *very* code heavy blog post, but I just wanted to demonstrate how to use the Yara Library as I couldn't really find too much online about it. As mentioned, this is a feature in PreEmpt and writing this blog helped me get a better understanding of how this works.

The code for this agent is available on GitHub.