YARA for config extraction

August 8, 2022

Abdallah Elshinbary

Malware Analysis & Reverse Engineering Adventures

8 minute read

YARA is a tool aimed at helping malware researchers to identify and classify malware samples. It’s considered to be the pattern matching swiss knife for malware researchers.

If you are not familiar with writing YARA rules, the official docs would be a great start.

In this blog I will go through how YARA rules can be used for malware config extraction.

YARA has come a long way since its original release and it now has some awesome modules for writing better and more complex rules.

What is a YARA module

A YARA module is like a plugin for extending YARA features, it allows you to define data structures and functions which can be used in your rules.

To use a YARA module you simply import it using `import "module_name"`, you can refer to the docs to learn about the available functions of each module.

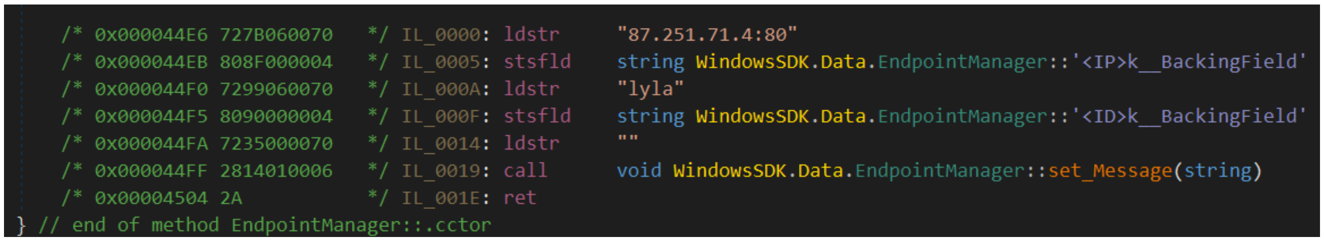
Example:

```
import "pe"

rule test {
    condition:
        pe.number_of_sections == 1
}
```

With that said let's now jump into malware land, I will demonstrate on two variants of RedLine Stealer which is a very popular dotnet stealer.

RedLine Stealer Variant1

The first variant stores the config in plaintext, we are only interested in two fields (C2 and BotnetID).



```
    /* 0x000044E6 727B060070   */ IL_0000: ldstr      "87.251.71.4:80"
    /* 0x000044EB 808F000004   */ IL_0005: stsfld     string WindowsSDK.Data.EndpointManager::'<IP>k__BackingField'
    /* 0x000044F0 7299060070   */ IL_000A: ldstr      "lyla"
    /* 0x000044F5 8090000004   */ IL_000F: stsfld     string WindowsSDK.Data.EndpointManager::'<ID>k__BackingField'
    /* 0x000044FA 7235000070   */ IL_0014: ldstr      ""
    /* 0x000044FF 2814010006   */ IL_0019: call       void WindowsSDK.Data.EndpointManager::set_Message(string)
    /* 0x00004504 2A           */ IL_001E: ret
} // end of method EndpointManager::.cctor
```

To read these fields we need to understand how ldstr instruction works. The instruction's opcode is `0x72` followed by 4 bytes which represent the string token.

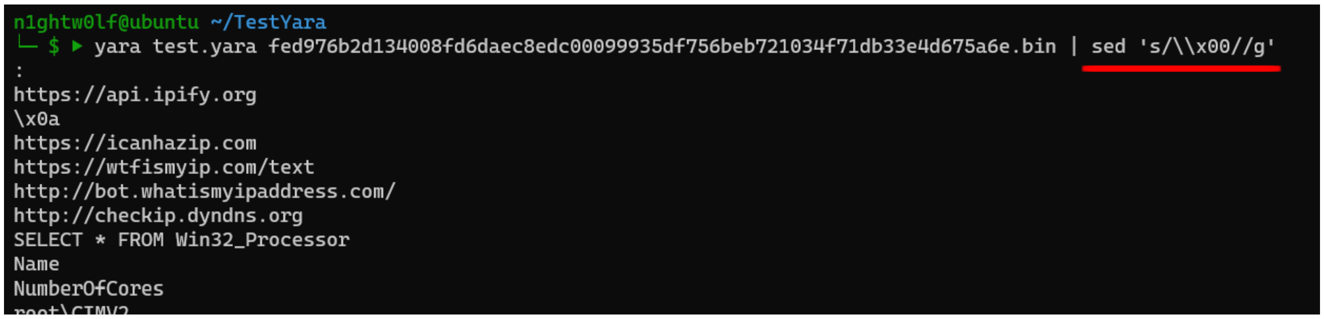
> A token is a DWORD value that represents a table and an index into that table. For example, the EntryPointToken 0x0600002C, references table 0x06 (MethodDef) and its row 0x2C. The table index is 1 byte and the row index is 3 bytes.

In the following instruction for example, the string token is `0x7000067B` (little-endian) and the row index is `0x67B`.

```
727B060070   // ldstr "87.251.71.4:80"
```

The `dotnet` module already has the functionality to retrieve all user strings from a dotnet sample.

```
import "dotnet"
import "console"

rule Test {
    condition:
        for all i in (0..dotnet.number_of_user_strings-1): (
            console.log(dotnet.user_strings[i])
        )
}
```

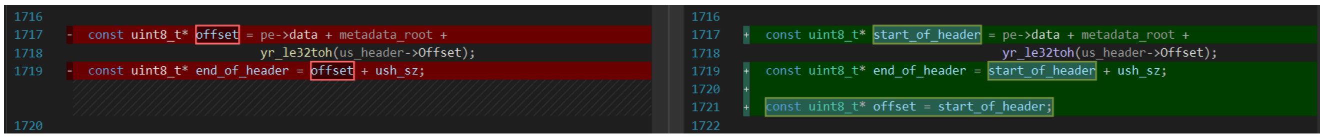


```
n1ghtw0lf@ubuntu ~/TestYara
└─ $ ▶ yara test.yara fed976b2d134008fd6daec8edc00099935df756beb721034f71db33e4d675a6e.bin | sed 's/\\x00//g'
:
https://api.ipify.org
\x0a
https://icanhazip.com
https://wtfismyip.com/text
http://bot.whatismyipaddress.com/
http://checkip.dyndns.org
SELECT * FROM Win32_Processor
Name
NumberOfCores
root\CIMV2
```

> Notice that I used `sed` to remove null characters because dotnet user strings are stored as an array of unicode strings.

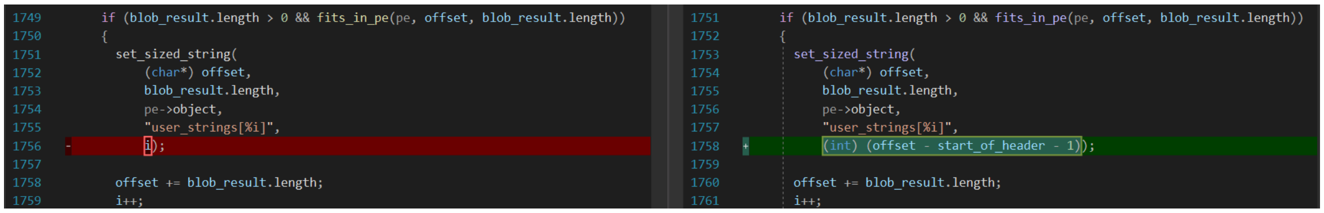
This is cool but we need to get the user strings using the row index from the string token.

To achieve this we need to make a couple of changes to the `dotnet` module source file at `libyara/modules/dotnet/dotnet.c`.



```
- const uint8_t* offset = pe->data + metadata_root +
                         yr_le32toh(us_header->Offset);
- const uint8_t* end_of_header = offset + ush_sz;
```

```
+ const uint8_t* start_of_header = pe->data + metadata_root +
                                   yr_le32toh(us_header->Offset);
+ const uint8_t* end_of_header = start_of_header + ush_sz;
+
+ const uint8_t* offset = start_of_header;
```



```
      if (blob_result.length > 0 && fits_in_pe(pe, offset, blob_result.length))
      {
        set_sized_string(
            (char*) offset,
            blob_result.length,
            pe->object,
            "user_strings[%i]",
-           i);

        offset += blob_result.length;
        i++;
```

```
      if (blob_result.length > 0 && fits_in_pe(pe, offset, blob_result.length))
      {
        set_sized_string(
            (char*) offset,
            blob_result.length,
            pe->object,
            "user_strings[%i]",
+           (int) (offset - start_of_header - 1));

        offset += blob_result.length;
        i++;
```

This will index the user strings array by row index (offset from the start of the strings table).

To compile and install yara you need to run these two scripts for the first time only:

```
$ ./bootstrap.sh
$ ./configure
```

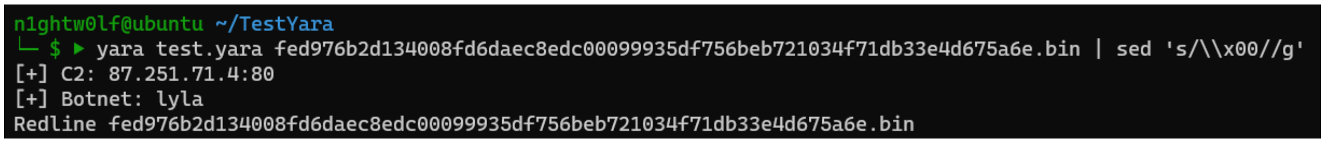
Then you build YARA with your changes:

```
$ make
$ sudo make install
```

We can now write a simple rule to read the config fields.

```
import "dotnet"
import "console"

rule Redline {
    strings:
        $get_conf_v1 = {
            72 ?? ?? ?? 70      // IL_0000: ldstr     "87.251.71.4:80"
            80 ?? ?? ?? 04      // IL_0005: stsfld    <IP>
            72 ?? ?? ?? 70      // IL_000A: ldstr     "lyla"
            80 ?? ?? ?? 04      // IL_000F: stsfld    <ID>
            72 ?? ?? ?? 70      // IL_0014: ldstr     ""
            28 ?? ?? ?? ??      // IL_0019: call      set_Message(string)
            2A                  // IL_001E: ret
        }

    condition:
        $get_conf_v1
        and console.log("[+] C2: ",
            dotnet.user_strings[int32(@get_conf_v1+1) & 0xffffff]
        )
        and console.log("[+] Botnet: ",
            dotnet.user_strings[int32(@get_conf_v1+11) & 0xffffff]
        )
}
```

> - `@get_conf_v1` : address of the first match of `$get_conf_v1`
> - `int32` : reads 4 bytes (string token) from an offset, I used `0xffffff` bit mask to only get the row index.



```
n1ghtw0lf@ubuntu ~/TestYara
└─ $ ▶ yara test.yara fed976b2d134008fd6daec8edc00099935df756beb721034f71db33e4d675a6e.bin | sed 's/\\x00//g'
[+] C2: 87.251.71.4:80
[+] Botnet: lyla
Redline fed976b2d134008fd6daec8edc00099935df756beb721034f71db33e4d675a6e.bin
```

Cool, Let's move to the second variant.

RedLine Stealer Variant2

This variant stores the config in an encrypted form.

```
  /* 0x00003909 7249040070   */ IL_0000: ldstr      "CyYOXysPAwUnB1NQCxtdWioxKUInBC5QCDNUUw=="
  /* 0x0000390E 8002000004   */ IL_0005: stsfld     string Arguments::IP
  /* 0x00003913 729B040070   */ IL_000A: ldstr      "FzcNJDEOEDw7O1Y/FEM/IQ=="
  /* 0x00003918 8003000004   */ IL_000F: stsfld     string Arguments::ID
  /* 0x0000391D 72CD040070   */ IL_0014: ldstr      ""
  /* 0x00003922 8004000004   */ IL_0019: stsfld     string Arguments::Message
  /* 0x00003927 72CF040070   */ IL_001E: ldstr      "Fringe"
  /* 0x0000392C 8005000004   */ IL_0023: stsfld     string Arguments::Key
  /* 0x00003931 2A           */ IL_0028: ret
// end of method Arguments::.cctor
```

The decryption algorithm looks as follows:

```
result = StringDecrypt.FromBase64(StringDecrypt.Xor(StringDecrypt.FromBase64(b64), stringKey));
```

Currently YARA doesn't have a module to do `base64` and `xor` operations in conditions, so why not write our own module :)

Writing our own YARA module

Modules are written in C and built into YARA as part of the compiling process.

I will explain briefly how to write a YARA module, for more details refer to the official docs.

YARA modules reside in `libyara/modules`, it's recommended to use the module name as the file name for the source file. Here I created a new module directory named `malutils` and inside it is the source file named `malutils.c`, now let's go through the source code.

First we need to include the required headers to be able to use YARA's module API.

```
#include <yara/mem.h>
#include <yara/modules.h>

#define MODULE_NAME malutils
```

Next we define the required functions:

> Xor decryption function which takes a buffer and a key and returns the decrypted string buffer.

```
/**
 * xors a buffer with a key
 * @param buffer string
 * @param key string
 * @return xored string
 */
define_function(xord)
{
  YR_OBJECT* module = module();

  SIZED_STRING* buf = sized_string_argument(1);
  SIZED_STRING* key = sized_string_argument(2);
  unsigned char* out = module_alloc_data(module->data, buf->length + 1);

  for (size_t i = 0; i < buf->length; i++)
    out[i] = buf->c_string[i] ^ key->c_string[i % key->length];

  out[buf->length] = 0;
  return_string(out);
}
```

Base64 decoding function which takes a base64 encoded string and returns the decoded value.

```
/**
 * base64 decodes a string
 * @param buffer string
 * @return decoded string
 */
define_function(base64d)
{
  YR_OBJECT* module = module();

  SIZED_STRING* buf = sized_string_argument(1);

  const unsigned char base64_table[65] =
      "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

  unsigned char dtable[256], *out, *pos, block[4], tmp;
  size_t i, count, olen;
  int pad = 0;
```

Helper function to convert dotnet user strings from wide to ascii.

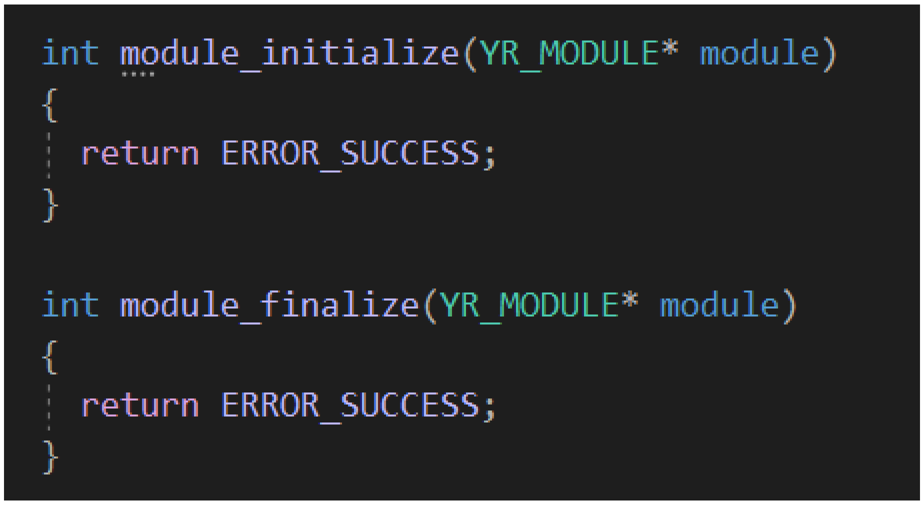
```
/**
* converts wide to ascii by removing null bytes
* @param wide string
* @return ascii string
*/
define_function(wtoa)
{
  YR_OBJECT* module = module();

  SIZED_STRING* buf = sized_string_argument(1);
  unsigned char* out = module_alloc_data(module->data, buf->length / 2 + 1);

  for (size_t i = 0; i < buf->length / 2; i++) out[i] = buf->c_string[i * 2];

  out[buf->length / 2] = 0;
  return_string(out);
}
```

Then comes the declaration section where we declare the functions and data structures that will be available for our YARA rules.

```
begin_declarations
  declare_function("xord", "ss", "s", xord);
  declare_function("base64d", "s", "s", base64d);
  declare_function("wtoa", "s", "s", wtoa);
end_declarations
```

After that we have 2 pairs of functions, the first pair is `module_initialize` & `module_finalize`.

These functions allow you to initialize and finalize any global data structure you may need to use in your module, and both functions are invoked whether or not the module is being imported by some rule.

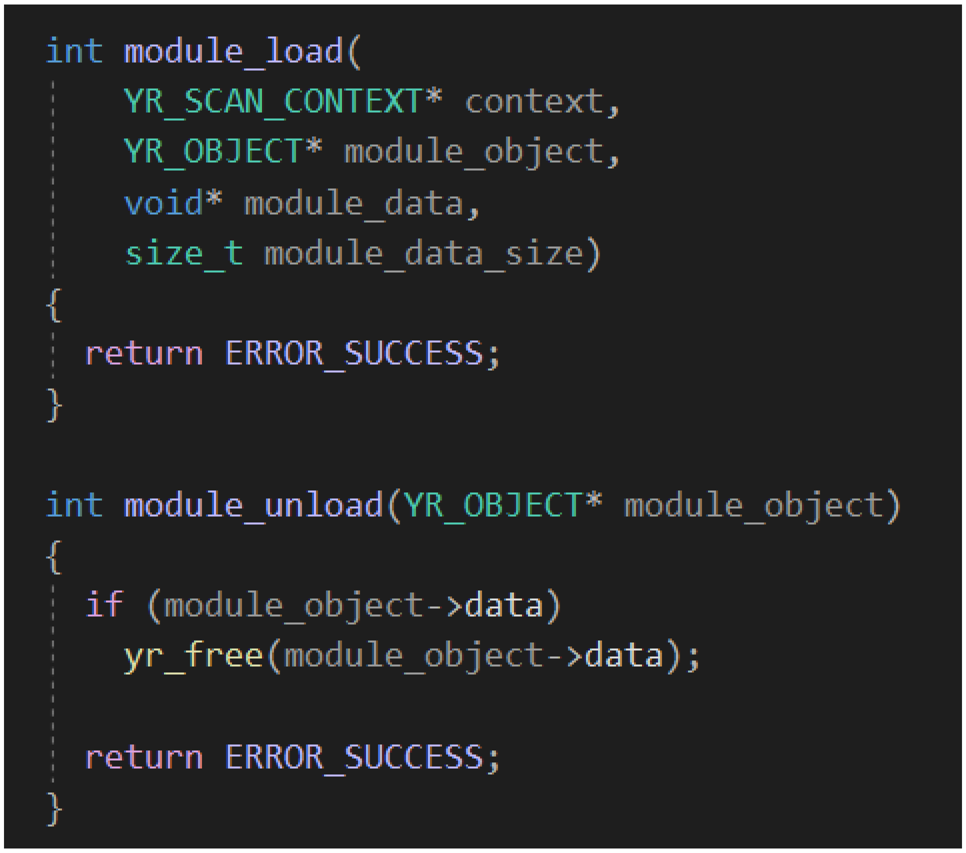


```c
int module_initialize(YR_MODULE* module)
{
  return ERROR_SUCCESS;
}

int module_finalize(YR_MODULE* module)
{
  return ERROR_SUCCESS;
}
```

The second pair is `module_load` & `module_unload`.

The `module_load` function is invoked once for each scanned file (only if the module is imported in your rule). It's is where your module can inspect the file being scanned, parse or analyze it in the way preferred, and then populate the data structures defined in the declarations section.

For each call to `module_load` there is a corresponding call to `module_unload`. This function allows your module to free any resource allocated during `module_load`.



```c
int module_load(
    YR_SCAN_CONTEXT* context,
    YR_OBJECT* module_object,
    void* module_data,
    size_t module_data_size)
{
  return ERROR_SUCCESS;
}

int module_unload(YR_OBJECT* module_object)
{
  if (module_object->data)
    yr_free(module_object->data);

  return ERROR_SUCCESS;
}
```

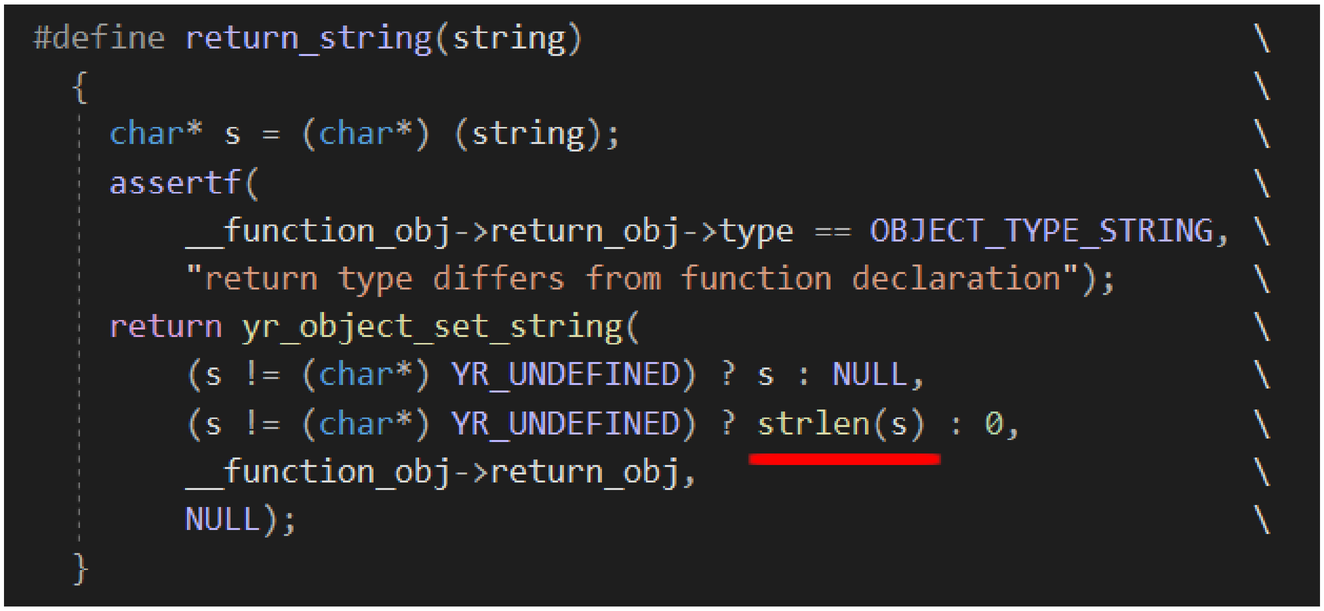
Final Touches

Before we test our module there's a nasty bug we need to take care of.

When writing a YARA module, instead of using the C *return* statement in your declared functions you must use `return_string(x)`, `return_integer(x)` or `return_float(x)` to return from a function.

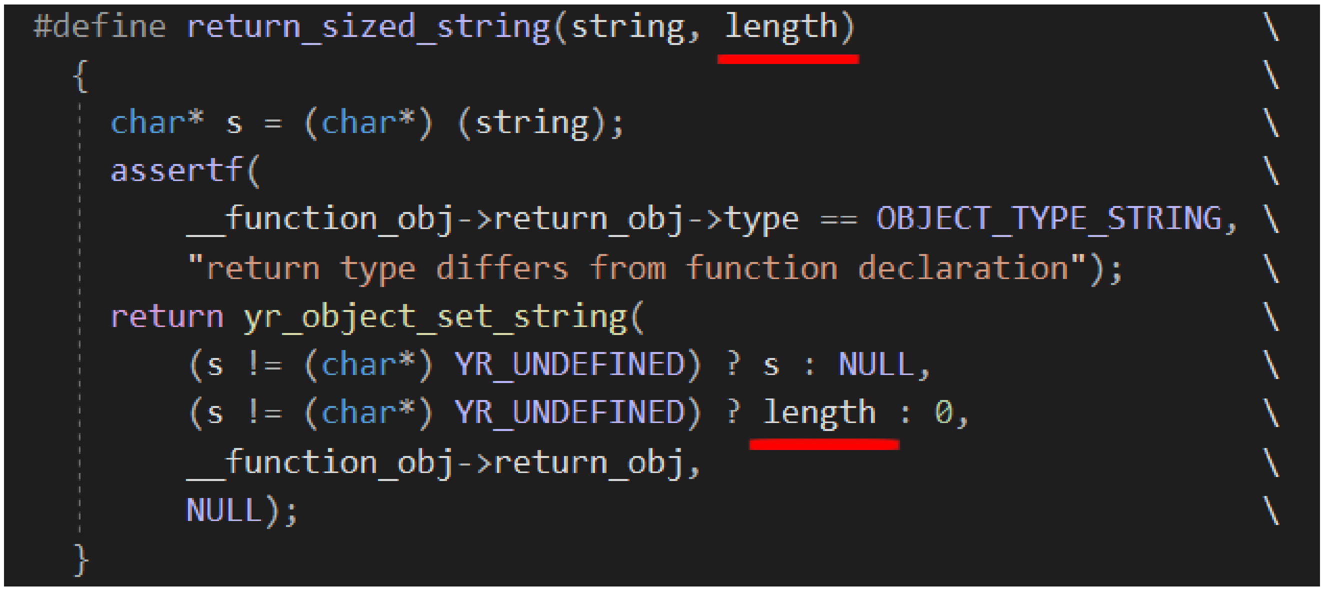
The problem occurs when we return from `base64d` function, the decoded string might contain null bytes so `return_string` won't return the full buffer.

As you can see below, `return_string` uses `strlen` to determine the length of the returned string so it will stop at the first null byte.



```
#define return_string(string)                                       \
  {                                                                 \
    char* s = (char*) (string);                                     \
    assertf(                                                        \
        __function_obj->return_obj->type == OBJECT_TYPE_STRING,     \
        "return type differs from function declaration");           \
    return yr_object_set_string(                                    \
        (s != (char*) YR_UNDEFINED) ? s : NULL,                     \
        (s != (char*) YR_UNDEFINED) ? strlen(s) : 0,                \
        __function_obj->return_obj,                                 \
        NULL);                                                      \
  }
```

As a workaround, I defined a new return macro called `return_sized_string` which enables us to set the length of the returned string rather than relying on `strlen`.



```
#define return_sized_string(string, length)                         \
  {                                                                 \
    char* s = (char*) (string);                                     \
    assertf(                                                        \
        __function_obj->return_obj->type == OBJECT_TYPE_STRING,     \
        "return type differs from function declaration");           \
    return yr_object_set_string(                                    \
        (s != (char*) YR_UNDEFINED) ? s : NULL,                     \
        (s != (char*) YR_UNDEFINED) ? length : 0,                   \
        __function_obj->return_obj,                                 \
        NULL);                                                      \
  }
```

Building our module

To include our module in the compiling process of YARA we must follow two further steps:

- Add our module name to the module_list at `libyara/modules/module_list`

```
MODULE(malutils)
```

- Add our module source file to the must compiled modules at `libyara/Makefile.am`

```
MODULES += modules/malutils/malutils.c
```

Finally we build YARA with our module:

```
$ make
$ sudo make install
```

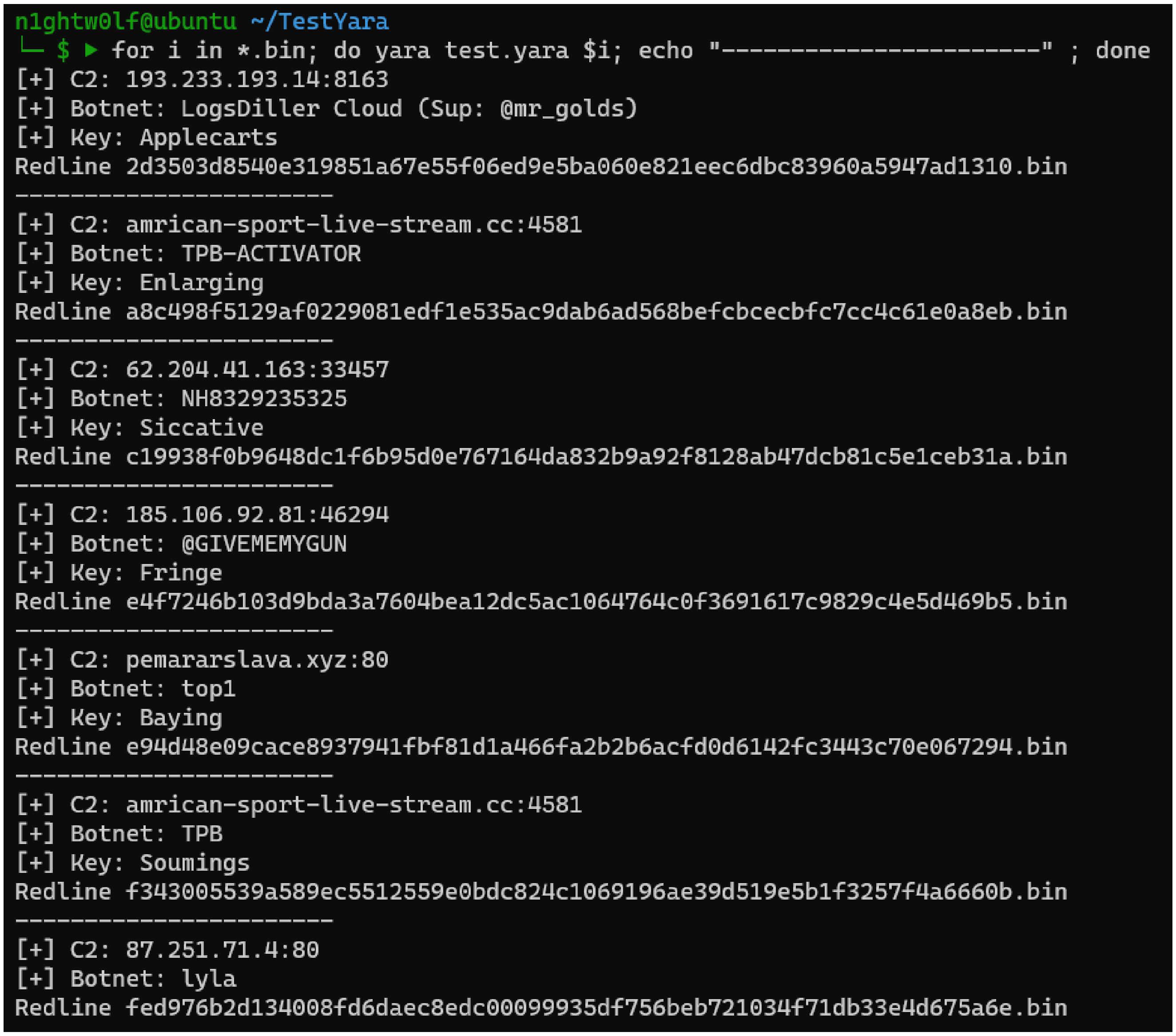
With everything in place, let’s now test our module.

Testing our module

Below is the final YARA rule that handles both RedLine variants.

```
import "dotnet"
import "console"
import "malutils"

rule Redline {
    meta:
        date = "2022-08-08"
        author = "Abdallah 'n1ghtw0lf' Elshinbary"
        description = "Extracts Redline config (educational)"

    strings:
        $get_conf_v1 = {
            72 ?? ?? ?? 70      // IL_0000: ldstr     "87.251.71.4:80"
            80 ?? ?? ?? 04      // IL_0005: stsfld    <IP>
            72 ?? ?? ?? 70      // IL_000A: ldstr     "lyla"
            80 ?? ?? ?? 04      // IL_000F: stsfld    <ID>
            72 ?? ?? ?? 70      // IL_0014: ldstr     ""
            28 ?? ?? ?? ??      // IL_0019: call      set_Message(string)
            2A                  // IL_001E: ret
        }
        $get_conf_v2 = {
            72 ?? ?? ?? 70      // IL_0000: ldstr
"CyYOXysPAwUnB1NQCxtdWioxKUInBC5QCDNUUw=="
            80 ?? ?? ?? 04      // IL_0005: stsfld    <IP>
            72 ?? ?? ?? 70      // IL_000A: ldstr     "FzcNJDEOEDw7O1Y/FEM/IQ=="
            80 ?? ?? ?? 04      // IL_000F: stsfld    <ID>
            72 ?? ?? ?? 70      // IL_0014: ldstr     ""
            80 ?? ?? ?? 04      // IL_0019: stsfld    <Message>
            72 ?? ?? ?? 70      // IL_001E: ldstr     "Baying"
            80 ?? ?? ?? 04      // IL_0023: stsfld    <Key>
        }

    condition:
        dotnet.is_dotnet and
        (
          (
            $get_conf_v1
            and console.log("[+] C2: ",
              malutils.wtoa(dotnet.user_strings[int32(@get_conf_v1+1) & 0xffffff])
            )
            and console.log("[+] Botnet: ",
              malutils.wtoa(dotnet.user_strings[int32(@get_conf_v1+11) & 0xffffff])
            )
          )
          or
          (
            $get_conf_v2
            and console.log("[+] C2: ",
              malutils.base64d(
                malutils.xord(
                  malutils.base64d(
                    malutils.wtoa(dotnet.user_strings[int32(@get_conf_v2+1) &
0xffffff])    // enc c2
                  ), malutils.wtoa(dotnet.user_strings[int32(@get_conf_v2+31) &
0xffffff])  // xor key
                )
              )
            )
            and console.log("[+] Botnet: ",
              malutils.base64d(
                malutils.xord(
                  malutils.base64d(
                    malutils.wtoa(dotnet.user_strings[int32(@get_conf_v2+11) &
0xffffff])   // enc botnet
                  ), malutils.wtoa(dotnet.user_strings[int32(@get_conf_v2+31) &
0xffffff])  // xor key
                )
              )
            )
            and console.log("[+] Key: ",
              malutils.wtoa(dotnet.user_strings[int32(@get_conf_v2+31) & 0xffffff])
            )
          )
        )
}
```

Running the rule on a list of samples produces the following output:



```
n1ghtw0lf@ubuntu ~/TestYara
└─ $ ▶ for i in *.bin; do yara test.yara $i; echo "-----------------------" ; done
[+] C2: 193.233.193.14:8163
[+] Botnet: LogsDiller Cloud (Sup: @mr_golds)
[+] Key: Applecarts
Redline 2d3503d8540e319851a67e55f06ed9e5ba060e821eec6dbc83960a5947ad1310.bin
-----------------------
[+] C2: amrican-sport-live-stream.cc:4581
[+] Botnet: TPB-ACTIVATOR
[+] Key: Enlarging
Redline a8c498f5129af0229081edf1e535ac9dab6ad568befcbcecbfc7cc4c61e0a8eb.bin
-----------------------
[+] C2: 62.204.41.163:33457
[+] Botnet: NH8329235325
[+] Key: Siccative
Redline c19938f0b9648dc1f6b95d0e767164da832b9a92f8128ab47dcb81c5e1ceb31a.bin
-----------------------
[+] C2: 185.106.92.81:46294
[+] Botnet: @GIVEMEMYGUN
[+] Key: Fringe
Redline e4f7246b103d9bda3a7604bea12dc5ac1064764c0f3691617c9829c4e5d469b5.bin
-----------------------
[+] C2: pemararslava.xyz:80
[+] Botnet: top1
[+] Key: Baying
Redline e94d48e09cace8937941fbf81d1a466fa2b2b6acfd0d6142fc3443c70e067294.bin
-----------------------
[+] C2: amrican-sport-live-stream.cc:4581
[+] Botnet: TPB
[+] Key: Soumings
Redline f343005539a589ec5512559e0bdc824c1069196ae39d519e5b1f3257f4a6660b.bin
-----------------------
[+] C2: 87.251.71.4:80
[+] Botnet: lyla
Redline fed976b2d134008fd6daec8edc00099935df756beb721034f71db33e4d675a6e.bin
```

Beautiful right!

You can pull the code and try it yourself at https://github.com/N1ght-W0lf/yara/tree/malutils.

It was just for learning purposes so not the best code :)

Samples

fed976b2d134008fd6daec8edc00099935df756beb721034f71db33e4d675a6e
e4f7246b103d9bda3a7604bea12dc5ac1064764c0f3691617c9829c4e5d469b5
2d3503d8540e319851a67e55f06ed9e5ba060e821eec6dbc83960a5947ad1310
a8c498f5129af0229081edf1e535ac9dab6ad568befcbcecbfc7cc4c61e0a8eb
c19938f0b9648dc1f6b95d0e767164da832b9a92f8128ab47dcb81c5e1ceb31a
e94d48e09cace8937941fbf81d1a466fa2b2b6acfd0d6142fc3443c70e067294
f343005539a589ec5512559e0bdc824c1069196ae39d519e5b1f3257f4a6660b

References

https://yara.readthedocs.io/en/stable/index.html

https://www.ntcore.com/files/dotnetformat.htm