

Automated dynamic import resolving using binary emulation

L lopqto.me/posts/automated-dynamic-import-resolving

Sep 8, 2020

Analyzing malwares is often not an easy task because there are lots of tricks and techniques that malwares use to evade detection and classification or to make the post-analysis more difficult. One such trick is to resolve windows API calls dynamically (called “dynamic import resolving”).

In this blog post, we will talk about dynamic import resolving and a pattern to detect it when reversing malwares, how to defeat this trick using binary emulation and Qiling framework (resolve API calls and extract function names), and finally we will integrate our emulation framework with Ghidra.

In the last section, we will talk about a solution to run Python version 3 and Qiling through Ghidra so we can see the result of our script inside the decompiler/disassembler view. It will make post-analysis easier.

As a real-life example, we will analyze Netwalker which used this technique and we will discuss our idea around that sample.

What is dynamic import resolving

Let's talk about dynamic import resolving and indirect function calls. It's a common technique that malwares use to hide their intention, make the static analysis more difficult, bypass some red flags, etc.

In this technique, the malware tries to create an IAT (Import Address Table) during the execution so there is no sign of used API calls in the PE header.

This technique often shows up in a specific pattern; At the beginning of the execution, the program will build an array of function pointers which works like an IAT and the malware can use stored function pointers with indirect calls as shown below:

00EA3A40	51	push ecx	
00EA3A41	57	push edi	
00EA3A42	C705 7412EB00 000000	mov dword ptr ds:[E81274],0	
00EA3A4C	E8 0FA50000	call netwalker.EADF60	
00EA3A51	8BF8	mov edi, eax	
00EA3A53	85FF	test edi, edi	
00EA3A55	0F84 0D010000	je netwalker.EA3868	
00EA3A5B	53	push ebx	
00EA3A5C	E8 9FE9FFFF	call netwalker.EA2400	
00EA3A61	68 39050000	push 539	
00EA3A66	68 697A0000	push 7A69	
00EA3A6B	57	push edi	
00EA3A6E	8B88 40010000	mov ecx, dword ptr ds:[eax+140]	
00EA3A72	FFD1	call ecx	
00EA3A74	8B08	mov ebx, eax	
00EA3A76	85DB	test ebx, ebx	
00EA3A78	0F84 E1000000	je netwalker.EA385F	
00EA3A7E	56	push esi	
00EA3A7F	E8 7CE9FFFF	call netwalker.EA2400	
00EA3A84	53	push ebx	
00EA3A85	57	push edi	
00EA3A86	8B88 44010000	mov ecx, dword ptr ds:[eax+144]	
00EA3A8C	FFD1	call ecx	
00EA3A8E	8BF0	mov esi, eax	esi:EntryPoint
00EA3A90	E8 68E9FFFF	call netwalker.EA2400	esi:EntryPoint
00EA3A95	56	push esi	esi:EntryPoint

ecx=<kerne132.FindResourceA> (7587BF00)

.text:00EA3A72 netwalker.exe:\$3A72 #2E72

It's rather difficult to determine which function would be called by these indirect function calls without actually executing the binary.

To dynamically make a function pointer, the two API calls `LoadLibrary()` and `GetProcAddress()` are often used.

According to the Microsoft docs, `LoadLibrary()` :

Loads the specified module into the address space of the calling process. The specified module may cause other modules to be loaded.

```
HMODULE LoadLibrary(
    LPCSTR lpLibFileName
);
```

And `GetProcAddress()` :

Retrieves the address of an exported function or variable from the specified dynamic-link library (DLL).

```
FARPROC GetProcAddress(
    HMODULE hModule,
    LPCSTR lpProcName
);
```

Look at this pseudo-code as a demonstration:

```
typedef ret_type (__stdcall *f_func)(param_a, param_b);

HINSTANCE hLibrary = LoadLibrary("ntdll.dll");
f_func LocalNtCreateFile = (f_func)GetProcAddress(hLibrary, "NtCreateFile");
```

`LocalNtCreateFile` is a function pointer which points to `NtCreateFile`, which can be stored in an array a.k.a IAT.

To make things more spicy, sometimes malware authors also encrypt the strings passed to `LoadLibrary()` and `GetProcAddress()` like what Netwalker did. It will be near to impossible to analyze malware without solving this problem first.

Choosing the approach

To solve these types of techniques and tricks there are a few approaches. For example, we can sometimes decrypt passed strings statically or we can develop an IDA plugin (or any disassembler and decompiler that supports plugins) but that would be a rather time-consuming task. Alternatively, we can use debuggers to execute the malware step by step, and rename variables according to dynamically resolved functions but this is a lot of repetition.

I chose binary emulation because it gives us the best of both worlds, We can have the power of automation *and* the ease of debugging. It's worth mentioning that emulating can be very slow at times, especially when dealing with encryption and decryption algorithms. Personally, I think this is an acceptable trade-off.

For binary emulation we will use Qiling. Read my [previous post](#) to see why.

Analyzing Netwalker

Today's sample is NetWalker [link!](#) . Netwalker used dynamic import resolving technique with encrypted strings so it is a good example for us to demonstrate our idea and approach around that.

As discussed before, most of the time malwares will try to build an IAT at the beginning of the execution - and NetWalker does this.

After disassembling the malware, we can see a function call right after the `entry` .

```
Decompile: entry - (netwalker.exe)
1
2 undefined4 entry(void)
3
4 {
5     int iVar1;
6
7     iVar1 = FUN_00401250();
8     if (iVar1 != 0) {
9         iVar1 = FUN_00403a40();
10        if (iVar1 != 0) {
11            iVar1 = FUN_00402c60();
12            if (iVar1 != 0) {
13                FUN_0040d490();
14            }
15        }
16        iVar1 = FUN_00402400();
17        (**(code **)(iVar1 + 0x120))(1);
18    }
19    Sleep(1);
20    return 0;
21 }
22
```

Jumping to that function, we can see the pattern mentioned above; A function is called multiple times and the return value is stored in an array.

```

Decompile: FUN_00401250 - (netwalker...
21  if (DAT_00411264 == (int *)0x0) {
22      return DAT_00411260;
23  }
24  iVar4 = FUN_00401000(iVar1, -0x5e2ba68c);
25  *DAT_00411264 = iVar4;
26  iVar4 = FUN_00401000(iVar1, -0x50ee43dc);
27  DAT_00411264[1] = iVar4;
28  iVar4 = FUN_00401000(iVar1, -0x468c4724);
29  DAT_00411264[2] = iVar4;
30  iVar4 = FUN_00401000(iVar1, -0x7b9c69f6);
31  DAT_00411264[3] = iVar4;
32  iVar4 = FUN_00401000(iVar1, -0x2ebe502d);
33  DAT_00411264[4] = iVar4;
34  iVar4 = FUN_00401000(iVar1, 0x57f17b6b);
35  DAT_00411264[5] = iVar4;
36  iVar4 = FUN_00401000(iVar1, 0x23398d9a);
37  DAT_00411264[6] = iVar4;
38  iVar4 = FUN_00401000(iVar1, -0x4298ca3d);
39  DAT_00411264[9] = iVar4;
40  iVar4 = FUN_00401000(iVar1, -0x6ff09592);
41  DAT_00411264[10] = iVar4;
42  iVar4 = FUN_00401000(iVar1, -0x57518bee);
43  DAT_00411264[7] = iVar4;
44  iVar4 = FUN_00401000(iVar1, 0x4896a43);
45  DAT_00411264[8] = iVar4;
46  iVar4 = FUN_00401000(iVar1, 0x4c8a5b22);
47  DAT_00411264[0xb] = iVar4;
48  iVar4 = FUN_00401000(iVar1, 0x61e2048f);
49  DAT_00411264[0xc] = iVar4;

```

This pattern is a sign of dynamic import resolving. We can confirm our guess with a debugger like below:

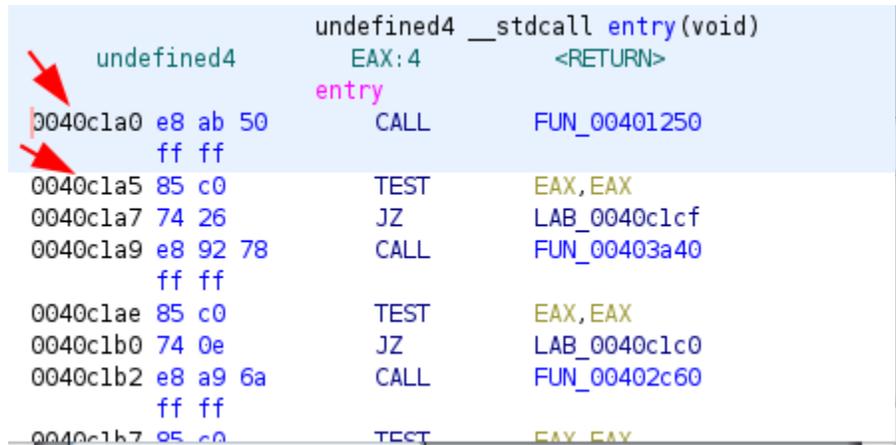
00EA1282	85FF	test edi,edi
00EA1284	0F84 67110000	je netwalker.EA23F1
00EA128A	68 84020000	push 284
00EA128F	6A 08	push 8
00EA1291	E8 2A3E0000	call netwalker.EA50C0
00EA1296	50	push eax
00EA1297	FFD7	call edi
00EA1299	A3 6412EB00	mov dword ptr ds:[EB1264],eax
00EA129E	85C0	test eax,eax
00EA12A0	0F84 4B110000	je netwalker.EA23F1
00EA12A6	68 7459D4A1	push A1D45974
00EA12AB	56	push esi
00EA12AC	E8 4FFDFFFF	call netwalker.EA1000
00EA12B1	8B0D 6412EB00	mov ecx,dword ptr ds:[EB1264]
00EA12B7	68 24BC11AF	push AF11BC24
00EA12BC	56	push esi
00EA12BD	8901	mov dword ptr ds:[ecx],eax
00EA12BF	E8 3CFDFFFF	call netwalker.EA1000
00EA12C4	8B0D 6412EB00	mov ecx,dword ptr ds:[EB1264]
00EA12CA	68 DCB873B9	push B973B8DC
00EA12CF	56	push esi
00EA12D0	8941 04	mov dword ptr ds:[ecx+4],eax
00EA12D3	E8 28FDFFFF	call netwalker.EA1000
00EA12D8	8B0D 6412EB00	mov ecx,dword ptr ds:[EB1264]
00EA12DE	68 0A966384	push 8463960A
00EA12E3	56	push esi

EIP → 00EA12C4
 ecx=3DB80
 dword ptr [netwalker.00EB1264]=<&RtlAllocateHeap>

Let's jump to the code and write a script to extract these function names.

I've discussed the basics of the Qiling like `hook_code()` and `q1.mem.read` in the [previous post](#).

In such scenarios, we don't need to emulate the entire malware, we just need to execute the dynamic import table resolution bit. So we need to find the start and the end of that section. This is rather easy because our target is inside a function, so we only need to emulate that specific function.



```
undefined4      undefined4 __stdcall entry(void)
EAX:4          <RETURN>
entry
0040c1a0 e8 ab 50      CALL     FUN_00401250
              ff ff
0040c1a5 85 c0        TEST    EAX,EAX
0040c1a7 74 26        JZ     LAB_0040c1cf
0040c1a9 e8 92 78      CALL     FUN_00403a40
              ff ff
0040c1ae 85 c0        TEST    EAX,EAX
0040c1b0 74 0e        JZ     LAB_0040c1c0
0040c1b2 e8 a9 6a      CALL     FUN_00402c60
              ff ff
0040c1b7 85 c0        TEST    EAX,EAX
```

```
q1.run(begin=0x0040c1a0, end=0x0040c1a5)
```

In this process of analyzing malwares with binary emulation, you need only be creative. For example, in this sample, there are plenty of approaches that you can use; however I chose the easiest and fastest (specifically development time, this solution performs rather badly).

Let's talk about the approach. As you can see in the image below, the return value of the (probably) decrypter and resolver function is stored in the `eax` register and then moved to `dword ptr [ecx + int]` . So we just need to hook the code and extract the value of `eax` in the right location.

```

004012c4 0b 0d 04      MOV     ECX,dword ptr [DAT_00411204]
          12 41 00
004012ca 68 dc b8      PUSH   0xb973b8dc
          73 b9
004012cf 56           PUSH   ESI
004012d0 89 41 04      MOV     dword ptr [ECX + 0x4],EAX
004012d3 e8 28 fd      CALL   FUN_00401000
          ff ff
004012d8 8b 0d 64      MOV     ECX,dword ptr [DAT_00411264]
          12 41 00
004012de 68 0a 96      PUSH   0x8463960a
          63 84
004012e3 56           PUSH   ESI
004012e4 89 41 08      MOV     dword ptr [ECX + 0x8],EAX
004012e7 e8 14 fd      CALL   FUN_00401000
          ff ff
004012ec 8b 0d 64      MOV     ECX,dword ptr [DAT_00411264]
          12 41 00
004012f2 68 d3 af      PUSH   0xd141afd3
          41 d1
004012f7 56           PUSH   ESI
004012f8 89 41 0c      MOV     dword ptr [ECX + 0xc],EAX
004012fb e8 00 fd      CALL   FUN_00401000
          ff ff
00401300 8b 0d 64      MOV     ECX,dword ptr [DAT_00411264]
          12 41 00
00401306 68 6b 7b      PUSH   0x57f17b6b
          f1 57

```

We can run the emulator and try to `hook_code()` to catch every instruction that is going to be executed.

```
q1.hook_code(extract_eax)
```

As you may notice, `extract_eax()` is a callback function that is designed to extract the value of `eax`. Qiling will pass the `q1` (sandbox) object, the `address` and the `size` of the instruction to this callback function.

We can extract the instruction inside `extract_eax()` with `mem.read()` as below:

```
buf = q1.mem.read(address, size)
```

`buf` is a Python `bytearray` of our instruction. The next step is detecting the right location to extract `eax`. By looking at the disassembler we can see a pattern. the first part of the opcode is similar.

004012c4	8b 0d 64 12 41 00	MOV	ECX,dword ptr [DAT_00411264]
004012ca	68 dc b8 73 b9	PUSH	0xb973b8dc
004012cf	56	PUSH	ESI
004012d0	89 41 04	MOV	dword ptr [ECX + 0x4],EAX
004012d3	e8 28 fd ff ff	CALL	FUN_00401000
004012d8	8b 0d 64 12 41 00	MOV	ECX,dword ptr [DAT_00411264]
004012de	68 0a 96 63 84	PUSH	0x8463960a
004012e3	56	PUSH	ESI
004012e4	89 41 08	MOV	dword ptr [ECX + 0x8],EAX
004012e7	e8 14 fd ff ff	CALL	FUN_00401000
004012ec	8b 0d 64 12 41 00	MOV	ECX,dword ptr [DAT_00411264]
004012f2	68 d3 af 41 d1	PUSH	0xd141afd3
004012f7	56	PUSH	ESI
004012f8	89 41 0c	MOV	dword ptr [ECX + 0xc],EAX
004012fb	e8 00 fd ff ff	CALL	FUN_00401000
00401300	8b 0d 64 12 41 00	MOV	ECX,dword ptr [DAT_00411264]
00401306	68 6b 7b f1 57	PUSH	0x57f17b6b

Next `if` will detect the right location:

```
if "8941" in buf.hex():
```

to extract `eax` value we need to do this:

```
eax_value = q1.reg.eax
```

`eax_value` is an address that points to an API call. We can search that address inside `import_symbols` to extract the API name.

```
func = q1.loader.import_symbols[eax_value]
func_dll = func["dll"]
func_name = func["name"].decode("ascii")
```

```
print(f"found {func_dll}.{func_name} at {hex(address)}")
```

Full code will be:

```

def extract_eax(q1, address, size):
    buf = q1.mem.read(address, size)

    if "8941" in buf.hex(): # dword ptr [ECX + hex],EAX
        eax_value = q1.reg.eax
        func = q1.loader.import_symbols[eax_value]
        func_dll = func["dll"]
        func_name = func["name"].decode("ascii")

        print(f"found {func_dll}.{func_name} at {hex(address)}")

```

This was easy! right? Next, we need to integrate our script with Ghidra to actually use the information we got here. This will help us to see extracted API names inside Ghidra.

Integrating Qiling with Ghidra

As you probably know Ghidra uses Jython and Jython only supports Python version 2 but Qiling is based on Python version 3. I found an interesting project called [ghidra_bridge link!](#) that helps us solve this problem.

So Ghidra Bridge is an effort to sidestep that problem - instead of being stuck in Jython, set up an RPC proxy for Python objects, so we can call into Ghidra/Jython-land to get the data we need, then bring it back to a more up-to-date Python with all the packages you need to do your work.

After installing [ghidra_bridge](#) you can find an example inside the installation directory called [example_py3_from_ghidra_bridge.py](#) . By opening this file we will have an idea about how to write scripts based on [ghidra_bridge](#) . Let's dissect it.

Most scripts should use this minimal template:

```

def run_script(server_host, server_port):

    import ghidra_bridge
    with ghidra_bridge.GhidraBridge(namespace=globals(), response_timeout=500):
        pass

if __name__ == "__main__":

    in_ghidra = False
    try:
        import ghidra
        # we're in ghidra!
        in_ghidra = True
    except ModuleNotFoundError:
        # not ghidra
        pass

    if in_ghidra:
        import ghidra_bridge_server
        script_file = getSourceFile().getAbsolutePath()
        # spin up a ghidra_bridge_server and spawn the script in external python to
        connect back to it

    ghidra_bridge_server.GhidraBridgeServer.run_script_across_ghidra_bridge(script_file)
    else:
        # we're being run outside ghidra! (almost certainly from spawned by
        run_script_across_ghidra_bridge())

        parser = argparse.ArgumentParser(
            description="py3 script that's expected to be called from ghidra with a
            bridge")
        # the script needs to handle these command-line arguments and use them to
        connect back to the ghidra server that spawned it
        parser.add_argument("--connect_to_host", type=str, required=False,
            default="127.0.0.1", help="IP to connect to the
            ghidra_bridge server")
        parser.add_argument("--connect_to_port", type=int, required=True,
            help="Port to connect to the ghidra_bridge server")

        args = parser.parse_args()

        run_script(server_host=args.connect_to_host,
            server_port=args.connect_to_port)

```

We only need to focus on `run_script()` function. The other part is static and probably there is no need to change. Only inside `run_script()` you are allowed to use Python 3 syntax and only here you are allowed to load Python 3 libraries (like Qiling). As you may notice I added `response_timeout` to the `GhidraBridge` object and sets its value to 500 seconds. Why? because as we discussed earlier emulating is a time-consuming task and emulating decryptor functions is likely more time-consuming because there is so much instruction code that needs to be emulated. So we need to set `response_timeout` to prevent any timeout-related errors.

Leaving aside the base template, we can now write our Qiling code inside `run_script()` .

```
def run_script(server_host, server_port):
    from qiling import Qiling

    import ghidra_bridge
    with ghidra_bridge.GhidraBridge(namespace=globals(), response_timeout=500):

        ql = Qiling(["/home/lopqto/w/automated/samples/netwalker.exe"],
"/home/lopqto/w/automated/rootfs/x86_windows", output = "debug")
        ql.hook_code(extract_eax)
        ql.run(begin=0x0040c1a0, end=0x0040c1a5)
```

Back to the `extract_eax()` function, we need to integrate it with Ghidra and add extracted API names as a comment into Ghidra. To add a comment from a script first of all we need an address (location). We have the `address` value from Qiling but we need to convert this value to Ghidra's `Address` type.

To do this we need `memory.blocks` object from `currentProgram` API. But there is a challenge here. `currentProgram` API only is accessible inside `run_script()` . But we need this API inside `extract_eax()` callback. There is a cool trick to handle this situation. You need to pass things around with `ql` object like below:

```
ql.target_block = currentProgram.memory.blocks[0]
```

Now we can access to `ql.target_block` inside `extract_eax()` . `target_block` (`memory.blocks[0]`) points to the PE entrypoint at `0x00400000` . to convert `address` to `Address` type we need to calculate offset and do something like this:

```
target_address = ql.target_block.getStart()
target_address = target_address.add(address - 0x00400000)
```

Now we have our `target_address` so we need one more step. accessing comment API is similar to above. First we need `getListring()` object:

```
ql.listing = currentProgram.getListring()
```

And to add a comment we can do:

```
codeUnit = ql.listing.getCodeUnitAt(target_address)
comment_message = "{}.{}".format(func_dll, func_name)
codeUnit.setComment(codeUnit.PRE_COMMENT, comment_message)
```

Full source code for `extract_eax()` will be this:

```

def extract_eax(q1, address, size):
    buf = q1.mem.read(address, size)
    if "8941" in buf.hex(): # dword ptr [ECX + hex],EAX

        eax_value = q1.reg.eax
        func = q1.loader.import_symbols[eax_value]
        func_dll = func["dll"]
        func_name = func["name"].decode("ascii")
        target_address = q1.target_block.getStart()
        target_address = target_address.add(address - 0x00400000)
        codeUnit = q1.listing.getCodeUnitAt(target_address)
        comment = "{}.{}".format(func_dll, func_name)
        codeUnit.setComment(codeUnit.PRE_COMMENT, comment)

```

Now we have a Ghidra script that will use Python3 to run samples trough Qiling and extract dynamic resolved function names and comment them into Ghidra. See the final result:



```

25  *DAT_00411264 = iVar4;
26  iVar4 = FUN_00401000(iVar1,-0x50ee43dc);
27                                     /* ntdll.RtlFreeHeap */
28  DAT_00411264[1] = iVar4;
29  iVar4 = FUN_00401000(iVar1,-0x468c4724);
30                                     /* ntdll.RtlReAllocateHeap */
31  DAT_00411264[2] = iVar4;
32  iVar4 = FUN_00401000(iVar1,-0x7b9c69f6);
33                                     /* ntdll.memset */
34  DAT_00411264[3] = iVar4;
35  iVar4 = FUN_00401000(iVar1,-0x2ebe502d);
36                                     /* ntdll.memcpy */
37  DAT_00411264[4] = iVar4;
38  iVar4 = FUN_00401000(iVar1,0x57f17b6b);
39                                     /* ntdll.memcmp */
40  DAT_00411264[5] = iVar4;
41  iVar4 = FUN_00401000(iVar1,0x23398d9a);
42                                     /* ntdll.sprintf */
43  DAT_00411264[6] = iVar4;
44  iVar4 = FUN_00401000(iVar1,-0x4298ca3d);
45                                     /* ntdll.strcpy */
46  DAT_00411264[9] = iVar4;
47  iVar4 = FUN_00401000(iVar1,-0x6ff09592);
48                                     /* ntdll.strcat */
49  DAT_00411264[10] = iVar4;
50  iVar4 = FUN_00401000(iVar1,-0x57518bee);
51                                     /* ntdll.st|chr */
52  DAT_00411264[7] = iVar4;
53  iVar4 = FUN_00401000(iVar1,0x4896a43);

```

And we are done. :)

Tips and tricks

Two tricks helped me to make this script. First of all, tracing the binary and printing assembly instructions can help a lot while debugging source!:

```
md = Cs(CS_ARCH_X86, CS_MODE_64)

def print_asm(ql, address, size):
    buf = ql.mem.read(address, size)
    for i in md.disasm(buf, address):
        print(":: 0x%x:\t%s\t%s" %(i.address, i.mnemonic, i.op_str))

ql.hook_code(print_asm)
```

You can compare emulation result with your disassembler to debug your program.

The second tip is when you try to run a time-consuming script and write something back to Ghidra (like adding a comment) you may face with an error like this:

```
ERROR (BackgroundCommandTask) Command Failure: An unexpected error occurred while
processing the command: Auto Analysis java.lang.RuntimeException: Timed-out waiting
to run a Swing task--potential deadlock!
```

It's because java closed the file and to solve this problem you need to increase timeout. Open the file in `ghidra/support/launch.properties` and add this line:

```
VMARGS=-Dghidra.util.Swing.timeout.seconds=3600
```

Conclusion

The idea described in this article can be extended and used to analyze any other malware families that dynamically resolve imports. It's not an ultimate general solution and you need to change things a little bit to match it against your target binary. I tried to explain my mindset behind the scene as much as possible to help you in this process. Hope this post was helpful.

Don't hesitate to ping me if there is something wrong or if you want to discuss about the post. I dropped the final script and the malware sample [here!](#).

Read more
