

[0001] AmberAmethystDaisy -> QuartzBegonia -> LummaStealer

🌐 0x1c.zip/0001-lummastealer/

0x1c

June 21, 2024

By 0x1c in [LummaStealer](#) — Jun 21, 2024



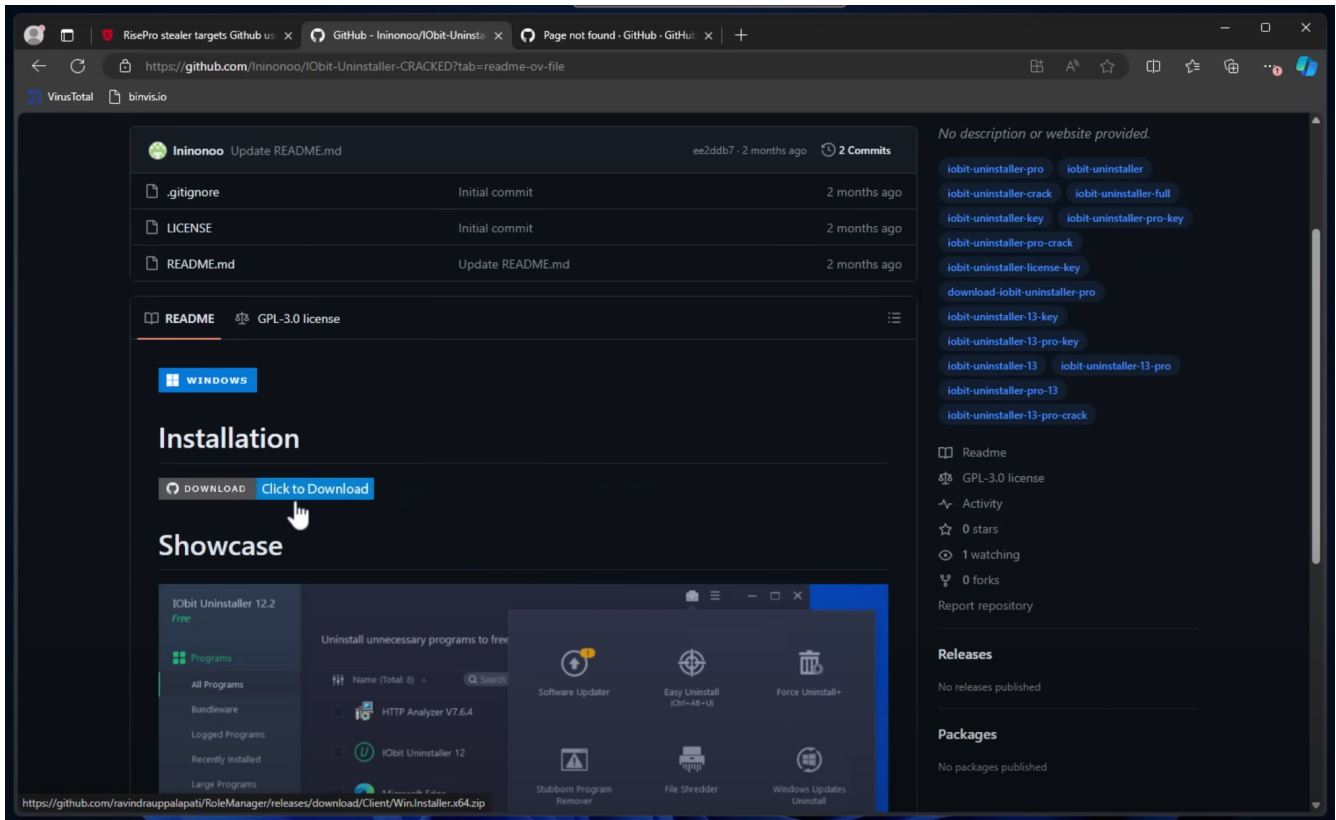
Disclaimer: I have personally noticed a significant difficulty in finding names for many loaders, even if they have been reported on due to the overwhelming focus on the final payload within infection chains. With this in mind, I utilize a custom loader taxonomy system, with the name of the loader in open-source reporting as a secondary identifier. More information on this taxonomy system can be found [here](#). If you happen to know the name of a loader that I report on, please let me know!

Recently, I stumbled across a video on YouTube from "[The PC Security Channel](#)", which noted that there was malware being distributed through fake cracked software on GitHub. Unfortunately, the extent of the analysis performed within the video was to check VirusTotal in order to see if the file is malicious or not.

Video: [How not to Pirate: Malware in cracks on Github \(youtube.com\)](#)

Although this might be good enough for most, my disappointment is *immeasurable*, and my day is nearly ruined. However, we can do the digging ourselves and get to the bottom of this!

Although the original GitHub repo that was shown within the video is now taken down, the actual download URL for the first stage seems to be hosted on another repo, as seen in the hyperlink within the video:



The URL seen in the hyperlink leads us to <https://github.com/ravindrauppapalapati/RoleManager/releases/tag/Client>, which is still up and available for download!

Stage 1 - QuartzDahlia

Also known as: Launch4j

TL;DR:

Initial sample can be executed as a normal executable as well as a JAR

SHA-256

Filename

8ed6a84101dfcafeac6ddb5f5020312b0094576fd3f9106f7df460e1b8a7bd5e1	Win.Installer.x64.zip
94edf5396599aaa9fca9c1a6ca5d706c130ff1105f7bd1acff83aff8ad513164	Win Installer x64.exe

Unpacking the ZIP archive, we can observe the following file structure:

```

| Win Installer x64.exe
|
├── v2024
│   ├── bin
│   │   ├── awt.dll
│   │   ├── glass.dll
│   │   ├── java.dll
│   │   ├── javafx_font.dll
│   │   ├── javafx_iio.dll
│   │   ├── javaw.exe
│   │   ├── msvcp120.dll
│   │   ├── msvcr100.dll
│   │   ├── msvcr120.dll
│   │   ├── net.dll
│   │   ├── nio.dll
│   │   ├── prism_d3d.dll
│   │   ├── sunec.dll
│   │   ├── sunmscapi.dll
│   │   ├── verify.dll
│   │   └── zip.dll
│   └── client
│       └── jvm.dll
├── lib
│   ├── jce.jar
│   ├── jfr.jar
│   ├── jsse.jar
│   ├── resources.jar
│   └── rt.jar
└── ext
    ├── jfxrt.jar
    ├── sunec.jar
    ├── sunjce_provider.jar
    └── sunmscapi.jar

```

Taking a look at the executable, it's unclear at first as to where the malicious code lies. With this in mind, I decided to load it up in x64dbg to do some quick preliminary dynamic analysis.

Stepping through a few functions, I was able to see that the malware attempts to call its own binary with the `-jar` flag using its bundled Java runtime. It turns out that this actually a tool named *Launch4j* which allows for Java applications to be wrapped in an executable.

Address	Hex	ASCII
00617DB0	22 43 3A 5C 55 73 65 72 73 5C 64 79 6E 5C 44 65	"c:\Users\dyn\De
00617DC0	73 68 74 6F 70 5C 4D 61 6C 77 61 72 65 5C 32 30	sktop\Malware\20
00617DD0	32 34 2D 30 35 2D 31 37 20 57 69 6E 2E 49 6E 73	24-05-17 Win.Ins
00617DE0	74 61 6C 6C 65 72 2E 78 36 34 5C 41 6E 61 6C 79	taller.x64\Analy
00617DF0	73 69 73 5C 57 69 6E 2E 49 6E 73 74 61 6C 6C 65	sis\win.Installe
00617E00	72 2E 78 36 34 5C 76 32 30 32 34 5C 62 69 6E 5C	r.x64\v2024\bin\
00617E10	6A 61 76 61 77 2E 65 78 65 22 20 2D 6A 61 72 20	javaw.exe" -jar
00617E20	22 43 3A 5C 55 73 65 72 73 5C 64 79 6E 5C 44 65	"c:\Users\dyn\De
00617E30	73 68 74 6F 70 5C 4D 61 6C 77 61 72 65 5C 32 30	sktop\Malware\20
00617E40	32 34 2D 30 35 2D 31 37 20 57 69 6E 2E 49 6E 73	24-05-17 Win.Ins
00617E50	74 61 6C 6C 65 72 2E 78 36 34 5C 41 6E 61 6C 79	taller.x64\Analy
00617E60	73 69 73 5C 57 69 6E 2E 49 6E 73 74 61 6C 6C 65	sis\win.Installe
00617E70	72 2E 78 36 34 5C 57 69 6E 20 49 6E 73 74 61 6C	r.x64\win Instal
00617E80	6C 65 72 20 78 36 34 2E 65 78 65 22 00 00 00 00	ler x64.exe"....
00617E90	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Since JAR files are able to be unzipped, we can go ahead and extract the contents of this executable with 7-Zip.

Note: Detect-It-Easy also notifies us that this executable contains a ZIP archive, and we could have gone about it that way as well!

```
extract
v2024
Win Installer x64.e
Win.Installer.x64.z

.data
.inc
.packages
.system
.theme
action
app
behaviour
bundle
com
css
facade
font
game
javassist
javax
JPHP-INF
JPHP-INFO
META-INF
org
php
script
timer
translation
tray
App.phb
Async.phb
cURLFile.phb
Dialog.phb
driver_property_info.properties
Files.phb
isc_dpb_types.properties
isc_error_msg.properties
isc_error_sqlstates.properties
isc_tpb_mapping.properties
# jfoenix-custom.fx.css
LICENSE.md
README.md
release-notes.txt
Win Installer x64.exe
```

Stage 2 - AmberAmethystDaisy

Also known as: D3F@ck Loader, NestoLoader

SHA-256

515d025ba2aa1096f65c13569de283b83d86824d08ca48c1fc3bc407d4cf3266 MainForm.phb

Filename

TL;DR:

- Extracted contents of the JAR contains files with the `.phb` extension, indicative of JPHP
- The entry point for JPHP-based applications can be found within `.system/application.conf`
In this case, the entry point resides in `app/forms/MainForm.phb`
- Utilizing [Binary Refinery](#) and [jadx](#), the next stage payload URL is retrieved.

A few of the extracted files have the `.phb` extension, which is indicative of [JPHP](#), an implementation of PHP on the Java VM. For more information on triaging JPHP malware, this same malware family was recently showcased on a [MalwareAnalysisForHedgehogs](#) video.

The entry point for JPHP-based applications can be found within `.system/application.conf`. The content of this file is as follows:

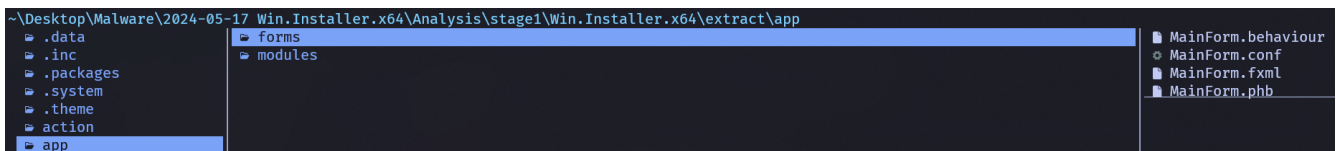
```
# MAIN CONFIGURATION

app.name = DarkLauncher
app.uuid = 6ccf8f8e-fb00-441b-a0f5-f3bc2fa6619b
app.version = 1

# APP
app.namespace = app
app.mainForm = MainForm
app.showMainForm = 1

app.fx.splash.autoHide = 0
```

We now know that the entry point that we are interested in would be located within the `app` folder and should be called `MainForm`. Let's go and take a look! Sure enough, a file titled `MainForm.phb` exists in the `forms` folder located within `app`.



Viewing this file with a hex editor, we can very quickly see what looks to be parts of an embedded configuration. Now we can be fairly sure that this is the file we want to be looking further into.

0230	73 74 72 61	63 74 46 6F	72 6D 00 00	00 03 00 00	structForm.....
0240	01 00 00 00	00 00 00 00	08 00 08 46	49 4C 45 31FILE1
0250	55 52 4C 00	00 00 00 04	00 3C 68 74	74 70 3A 2F	URL.....<http:/
0260	2F 7B 64 6F	6D 61 69 6E	7D 2F 61 75	74 6F 2F 62	/{domain}/auto/b
0270	30 35 37 33	63 65 66 35	66 62 66 65	66 35 61 31	0573cef5fbfef5a1
0280	35 65 38 61	36 35 32 37	30 38 30 61	64 32 35 2F	5e8a6527080ad25/
0290	39 33 2E 65	78 65 00 00	00 00 00 00	01 00 00 00	93.exe.....
02A0	00 00 00 00	08 00 08 46	49 4C 45 32	55 52 4C 00FILE2URL.
02B0	00 00 00 04	00 00 00 00	00 00 00 00	01 00 00 00
02C0	00 00 00 00	09 00 09 45	56 45 4E 54	5F 53 52 56EVENT_SRV
02D0	00 00 00 00	04 00 0E 31	39 34 2E 31	34 37 2E 33194.147.3
02E0	35 2E 32 35	31 00 00 00	00 00 00 00	00 00 00 00	5.251.....

Although we see a C2 IP address of `194.147.35[.]251` here, this is seemingly not where the next payload is hosted. Let's dig deeper to figure out where the next payload is actually hosted.

Dealing with PHB files

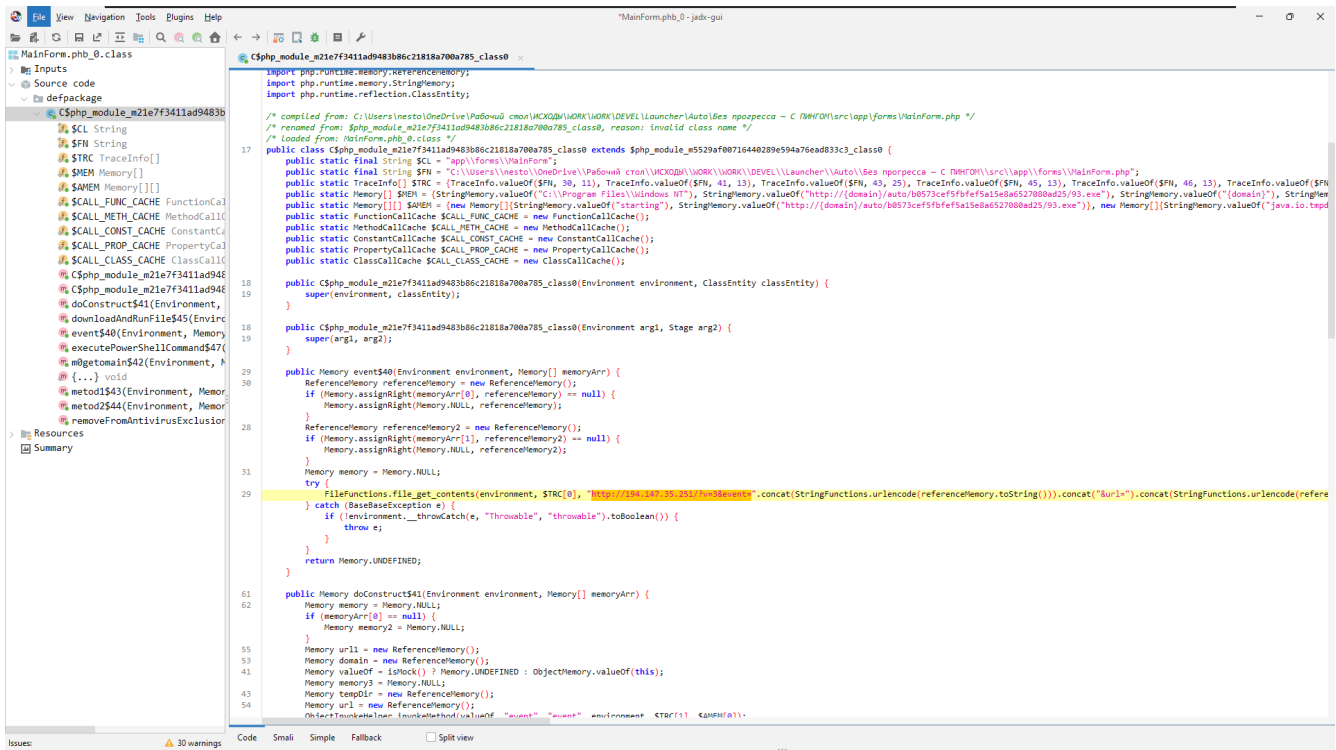
PHB files contain Java class files within them, which are denoted with a magic of `CAFEBABE`. We can utilize these magic bytes as a marker in order to extract the embedded `.class` files.

I set up the following Binary Refinery pipeline to extract the 2 class files from `app/forms/MainForm.phb`:

```
ef MainForm.phb | resplit h:CAFEBABE [ \
  | pop \
  | ccp h:CAFEBABE \
  | dump extracted_class_{index}.class \
]
```

Unit	Name	Definition
<code>ef</code>	Emit File	Places a file into the pipeline
<code>resplit</code>	Regular Expression Split	Splits the data in the pipeline by the supplied regular expression
<code>pop</code>	Pop	Removes a chunk from the frame (and stores it in a meta variable) - Used here to remove the first chunk in the pipeline, which contains data before the first <code>CAFEBABE</code> header
<code>ccp</code>	ConCat Prepend	Concatenates a value to the beginning of each chunk
<code>dump</code>	Dump	Dumps the data stored in each chunk to disk

Using `jadx`, we can decompile the recovered Java class files in order to get a better idea as to what the malicious code does.



Looking through the code, we come across 2 base64 encoded strings which decode to URLs. We can set up the following Binary Refinery pipeline to extract, defang, and print these indicators:

```
ef MainForm.phb | carve b64 [ \
  | b64 \
  | xtp ur1 \
  | defang \
  | cfmt "{ }\n" \
]
https[:]//pastebin[.]com/raw/md5jVrEB
https[:]//t[.]me/+JBdY0q1mUogwZWMY
```

Unit	Name	Definition
ef	Emit File	Places a file into the pipeline
carve	Carve	Extracts pieces of the pipeline that matches a given format - in this case, base64
b64	Base64	Base64 decodes each chunk in the pipeline
xtp	eXtraCT Pattern	Extracts indicators from the data within the pipeline by a given pattern
defang	Defang	Defangs indicators within the pipeline
cfmt	Convert to ForMaT	Transforms each chunk in the pipeline by applying a string format operation

The Pastebin URL holds a paste that contains the IP address **78.47.105[.]28**, which is where the next payload is hosted. We can now reconstruct the true URL of the next-stage payload:
[http://78.47.105\[.\]28/aut0/b0573cef5fbfef5a15e8a6527080ad25/93.exe](http://78.47.105[.]28/aut0/b0573cef5fbfef5a15e8a6527080ad25/93.exe)

Stage 3 - QuartzBegonia

Also known as: N/A

SHA-256

Filename

5b751d8100bbc6e4c106b4ef38f664fb031c86f919c3e2db59a36c70c57f54e0 93.exe

The third-stage payload in this infection chain is a loader written in C++. Loading the sample in Binary Ninja quickly reveals a large amount of non-code data, which is very likely the encrypted payload.



Within the main function, we can see that a thread would be created, which would execute a function which I've named `thread_start_addr` (0x401750) with an argument - a pointer to a function I've named `mal::thread_main` (0x41d7b0).

```
0041d9d0 int32_t mal::run_mal_main_thread()
0041d9d0 {
0041d9d3     FreeConsole();
0041d9db     DWORD* thread_main_ptr = operator new(4);
0041d9e5     if (thread_main_ptr == 0)
0041d9e5     {
0041d9ef         thread_main_ptr = nullptr;
0041d9e5     }
0041d9e5     else
0041d9e5     {
0041d9e7         *(uint32_t*)thread_main_ptr = mal::thread_main;
0041d9e5     }
0041da02     int32_t* thread_addr;
0041da02     uintptr_t tid = _beginthreadex(nullptr, 0, thread_start_addr, thread_main_ptr, 0, &thread_addr);
```

When called, the function `thread_start_addr` executes the function at the address that was passed-in as an argument:

```
00401750 uint32_t thread_start_addr(void* main_func_ptr)
00401750 {
00401755     *(uint32_t*)main_func_ptr();
00401759     __Cnd_do_broadcast_at_thread_exit();
0040175e     int32_t var_8 = 4;
00401761     operator new(main_func_ptr);
0040176c     return 0;
00401750 }
```

Diving into the `mal::thread_main` function, we come across an encrypted buffer and its corresponding decryption loop:

```
int32_t mal::thread_main()
0041d8f4 do
0041d8f4 {
0041d88c   enc_buf[counter] = ((enc_buf[counter] ^ 0x73) - 0x15);
0041d89e   if (((int8_t)((char*)val_edx - ptr_unk_1) >> 2) < 0x4ac)
0041d89e   {
0041d8a0       int32_t cur_idx = w_idx;
0041d8a6       if (val_edx == edx)
0041d8a6       {
0041d8bd           sub_4012f0(&ptr_unk, val_edx, &cur_idx);
0041d8c2           edx = var_2c;
0041d8c6           val_edx = val_edx_1;
0041d8ca           ptr_unk_1 = ptr_unk;
0041d8a6       }
0041d8a6       else
0041d8a6       {
0041d8a8           *(uint32_t*)val_edx = w_idx;
0041d8aa           val_edx = &val_edx[1];
0041d8ad           val_edx_1 = val_edx;
0041d8a6       }
0041d89e   }
0041d8d4   w_idx += 2;
0041d8e7   enc_buf[counter] = (((((((enc_buf[counter] - 0x57) ^ 0x74) + 0x4e) ^ 0x70) - 0x65) ^ 0x22) - 0x73) ^ 0x2a);
0041d8ed   counter += 1;
0041d8f4 } while (w_idx < 0x958);
```

Re-implementing this decryption loop in Python, we can recover the content of the encrypted buffer:


```
0x428000 .data {0x428000-0x47093c} Writable data
```

```
.data section started {0x428000-0x47093c}
```

```
00428000 BYTE rc4_key[0x31] =
```

```
00428000 {  
00428000     [0x00] = 0x22  
00428001     [0x01] = 0xa4  
00428002     [0x02] = 0x3b  
00428003     [0x03] = 0x87  
00428004     [0x04] = 0xdf  
00428005     [0x05] = 0x1e  
00428006     [0x06] = 0xde  
00428007     [0x07] = 0xe2  
00428008     [0x08] = 0x94  
00428009     [0x09] = 0xde  
0042800a     [0x0a] = 0xcd  
0042800b     [0x0b] = 0x10  
0042800c     [0x0c] = 0xf5  
0042800d     [0x0d] = 0xe8  
0042800e     [0x0e] = 0x5c  
0042800f     [0x0f] = 0x46  
00428010     [0x10] = 0x8f  
00428011     [0x11] = 0xcc  
00428012     [0x12] = 0xf9  
00428013     [0x13] = 0xfd  
00428014     [0x14] = 0xda  
00428015     [0x15] = 0x2e  
00428016     [0x16] = 0x48  
00428017     [0x17] = 0x84  
00428018     [0x18] = 0x17  
00428019     [0x19] = 0x17  
0042801a     [0x1a] = 0x96  
0042801b     [0x1b] = 0x5a  
0042801c     [0x1c] = 0xbe  
0042801d     [0x1d] = 0xdc  
0042801e     [0x1e] = 0xd6  
0042801f     [0x1f] = 0x1c  
00428020     [0x20] = 0xe4  
00428021     [0x21] = 0xdb  
00428022     [0x22] = 0xe9  
00428023     [0x23] = 0xf3  
00428024     [0x24] = 0xe0  
00428025     [0x25] = 0xc9  
00428026     [0x26] = 0xca  
00428027     [0x27] = 0x66  
00428028     [0x28] = 0xfc  
00428029     [0x29] = 0xea  
0042802a     [0x2a] = 0x73  
0042802b     [0x2b] = 0x76  
0042802c     [0x2c] = 0x2a  
0042802d     [0x2d] = 0x5b  
0042802e     [0x2e] = 0xe  
0042802f     [0x2f] = 0x5c  
00428030     [0x30] = 0x53  
00428031 }
```

```
00428031 00 00 00 00 00 00 00
```

```

00428038 BYTE buf_enc_next_stage[0x46600] =
00428038 {
00428038     [0x00000] = 0x12
00428039     [0x00001] = 0xab
0042803a     [0x00002] = 0xe5
0042803b     [0x00003] = 0xf3
0042803c     [0x00004] = 0xab
0042803d     [0x00005] = 0x2c

```

Key and encrypted content of the final payload, located within the `.data` segment

Taking a look at the function located at `0x41d4d0`, we can see telltale signs of the RC4 encryption algorithm:

```

0041d4d0 int32_t mal::decrypt_payload_rc4(BYTE* enc_data_buf, int32_t sizeof_data, BYTE* rc4_key, int32_t len_rc4_key_0x31)
0041d4d0 {
0041d4db     struct struct_unk* unk_struct_1;
0041d4db     int32_t eax_1 = (__security_cookie ^ &unk_struct_1);
0041d4f5     int32_t var_250 = 0;
0041d509     struct struct_unk* unk_struct = operator new(0x2c);
0041d515     int32_t i = 0;
0041d517     unk_struct->unk_struct_1 = unk_struct;
0041d51a     unk_struct->unk_struct_2 = unk_struct;
0041d51d     unk_struct->unk_struct_3 = unk_struct;
0041d520     unk_struct->val_0x101[0] = 1;
0041d520     unk_struct->val_0x101[1] = 1;
0041d526     unk_struct_1 = unk_struct;
0041d54a     BYTE s_box[0x100];
0041d54a     BYTE rc4_keystream[0x100];
0041d54a     do
0041d54a     {
0041d532         s_box[i] = i;
0041d536         int32_t eax_5;
0041d536         int32_t edx_1;
0041d536         edx_1 = HIGHD(((int64_t)i));
0041d536         eax_5 = LOWD(((int64_t)i));
0041d53c         rc4_keystream[i] = rc4_key[(((int8_t)(COMBINE(edx_1, eax_5) % len_rc4_key_0x31))];
0041d543         i += 1;
0041d54a     } while (i < 0x100);
0041d54c     int32_t i = 0;
0041d54e     int32_t j = 0;
0041d586     do
0041d586     {
0041d550         BYTE cur_sbox_byte = s_box[i];
0041d563         j = (((j + rc4_keystream[i]) + ((uint32_t)cur_sbox_byte)) & 0x800000ff);
0041d569         if (j < 0)
0041d569         {
0041d572             j = (((j - 1) | 0xfffff00) + 1);
0041d569         }
0041d577         s_box[i] = s_box[j];
0041d57b         i += 1;
0041d57c         s_box[j] = cur_sbox_byte;
0041d586     } while (i < 0x100);
0041d588     int32_t edx_2 = 0;
0041d58a     int32_t ebx = 0;
0041d58c     int32_t decrypt_idx = 0;
0041d597     if (sizeof_data > 0)
0041d597     {
0041d5a1         while (true)
0041d5a1         {
0041d5a1             int32_t ebx_2 = ((ebx + 1) & 0x800000ff);
0041d5a7             if (ebx_2 < 0)

```

Tip: Seeing two loops and the number `256 (0x100)` is often indicative of the RC4 encryption algorithm

With this information, I set up a Binary Refinery pipeline to decrypt the final payload:

```

ef 93.exe | \
    vsnip 0x428038:0x46600 | \
    rc4
h:22a43b87df1edee294decd10f5e85c468fccf9fdda2e48841717965abedcd61ce4dbe9f3e0c9ca66fcea73762a5b0e5c53
| \
    dump stage4.bin

```

Unit	Name	Definition
------	------	------------

Unit	Name	Definition
ef	Emit File	Places a file into the pipeline
vsnip	Virtual Snip	Snips (extracts) data from PE/ELF/MACHO files based on virtual offset
rc4	RC4	RC4 decrypts the data in the pipeline, given a key
dump	Dump	Dumps the data stored in each chunk to disk

Stage 4 - LummaStealer

Also known as: LummaC2 Stealer

SHA-256	Filename
0cf55c7e1a19a0631b0248fb0e699bbec1d321240208f2862e37f6c9e75894e7	N/A

Loading the *LummaStealer* sample in Binary Ninja, we see the following function:

```
? 00408ba0 void _start() __noreturn
  ⚠ This function has unresolved stack usage. View graph of stack usage to resolve.

00408bad   if ((sub_432130() & 1) != 0)
00408bba       if ((sub_42d6b0() & 1) != 0)
00408bd0           HANDLE var_104_1 = GetStdHandle(nStdHandle: STD_INPUT_HANDLE)
00408bf5           void var_100
00408bf5           if ((sub_409f50(sub_408c20(&var_100, "eleet or leetspeak, is a system of modified spellings used prima_")) & 1) != 0)
00408bf9               sub_40ff70()
00408c02               sub_4341b0()
00408c12           ExitProcess(uExitCode: 0)
00408c12           noreturn
```

Taking a look at the function `sub_432130`, we immediately come across a problem:

```
00432130 int32_t sub_432130()

? 00432154 | jump(data_440fe8 + (0x8070d626 ^ data_440fec) + 1)
```

Opaque Predicates

Here, we have an example of an obfuscation technique called *Opaque Predicates*. The jumps to the next section of code are obfuscated by making their destination the result of a mathematical operation. Typically, we would deal with these via patching, which is possible (this is not at the same place in code, but is an example of this technique):

```

int32_t __convention("regparm") sub_43438c(void* arg1)

004343aa          90 90 90 90 90 90          .....
004343b0  ff e0                    ..

004343b2  int32_t __fastcall sub_4343b2(int32_t arg1, int32_t arg2, int32_t arg3 @ esi)

004343b2  c744240800000000  mov     dword [esp+0x8 (arg_8)], 0x0
004343ba  31c0              xor     eax, eax (0x0)
004343bc  39d6              cmp     esi, edx
004343be  0f95c0           setne  al
004343c1  8b048560224400  mov     eax, dword [eax*4+0x442268]
004343c8  b9a6ef44ad      mov     ecx, 0xad44efa6
004343cd  330d68224400  xor     ecx, dword [data_442268]
004343d3  01c8              add     eax, ecx
004343d5  40                inc     eax
004343d6  8954240c        mov     dword [esp+0xc (arg_c)], edx
? 004343da  ffe0            jmp     eax

004343dc          90 90 90 90          ....
004343e0  8b 5e 30 31 c0 85 db 0f-95 c0 8b 04 85 9c 22 44  .^01....."D
004343f0  00 b9 b7 44 93 b2 33 0d          ...D..3.

004343f8  void* data_4343f8 = data_4422a4

004343fc          01 c8 40 bf          ..@.
00434400  01 00 00 00 ff e0 8b 4d-08 90 90 90 90 90 90 90  .....M.....
00434410  0f b7 11 0f b7 03 66 85-d2 74 0b 83 c3 02 83 c1  .....f..t.....
00434420  02 66 39 c2 74 ea 31 c9-66 39 c2 0f 94 c1 8b 04  .f9.t.1.f9.....
00434430  8d a8 22 44 00 b9 fd 7e-a1 50 33 0d          .."D...~.P3.

0043443c  void* data_43443c = 0x4422b0

```

However, I was recently informed by Xusheng from the Binary Ninja / Vector35 team (huge shoutouts to the team!) of a better way to tackle this:

By default, Binary Ninja believes that the value defined at `data_440fe8` and `data_440fec` can be modified by the program. Although this may be true, we know that this is likely not the case. With this in mind, if we convert the types—which are by default `void*`—to `const int32_t`, Binary Ninja can do its magic (dataflow analysis) in order to solve the opaque predicate for us!

```

00432130 int32_t sub_432130()
? 00432154 jump(data_440fe8 + (0x8070d626 ^ data_440fec) + 1)

00432156 int32_t sub_432156(void* arg1 @ esi)
00432161 int32_t var_30
00432161 __builtin_memcpy(dest: &var_30, src: "\x14\x85\x10\x8b\xee\x89\xe4\x8f\xe2\x4d\xde\x
00432183 int32_t s_1
00432183 __builtin_memset(s: &s_1, c: 0, n: 0x17)
004321ad *(arg1 + 8) = 0
004321ba if (*(arg1 + 8) u<= 0x2a)
004321de     do
004321d2         *(&var_30 + *(arg1 + 8)) = (((*(arg1 + 8)).b + 0x15) ^ *(&var_30 + *(arg1 +
004321d5         *(arg1 + 8) += 1
004321de     while (*(arg1 + 8) u< 0x2b)
004321e1 int32_t eax = sub_434300(&var_30)
004321e9 data_443e24 = eax
00432215 int32_t var_34_1 = eax
00432220 int32_t var_60
00432220 int32_t* var_64 = &var_60
00432227 void var_268
00432227 *(arg1 + 0x20) = &var_268
0043222a int32_t* var_26c = &var_60

```

Just like that, we can save our precious reverse-engineering time (and sanity...)! I originally was manually patching a whole bunch of these, and let me tell you—it was *miserable*.

However, going through the code a little more, we hit yet another roadblock:

```

004321f1 int32_t eax = sub_434300(&var_c0)
004321fb data_443e24 = eax
00432339 int32_t eax_1
00432339 eax_1.b = eax == 0
? 00432353 jump((0x5059d3bb ^ data_441004) + (&data_440ffc)[eax_1] + 1)

```

In this case, the value `data_440ffc` holds the address of 2 possible values used in order to calculate the destination. If we take a look at `data_440ffc`, right now, it is only showing up as a `void*`:

```

00440fe4 void* data_440fe4 = 0x4844fa
00440fe8 int32_t const data_440fe8 = 0x4845b9
00440fec int32_t const data_440fec = 0x7f8a0a7d
00440ff0 void* data_440ff0 = 0x4845b9
00440ff4 int32_t const data_440ff4 = 0x4846d9
00440ff8 int32_t const data_440ff8 = -0x5f34365b
00440ffc void* data_440ffc = 0x4846f9
00441000 void* data_441000 = 0x485236
00441004 int32_t data_441004 = -0x505cf020

```

Let's go ahead and change this to a `const int32_t[2]` in order to correctly reflect its type.

```

00440ff8 int32_t const data_440ff8 = 0x01040
00440ffc int32_t const data_440ffc[0x2] =
00440ffc {
00440ffc     [0x0] = 0x004846f9
00441000     [0x1] = 0x00485236
00441004 }

```

Now, if we change the type of `data_441004` to `const int32_t`, we can now see that the variable named `data_440ffc` has automatically been changed to `jump_table_440ffc`:

```

00440ffc int32_t const jump_table_440ffc[0x2] =
00440ffc {
00440ffc     [0x0] = 0x004846f9
00441000     [0x1] = 0x00485236
00441004 }
00441004 uint32_t data_441004 = 0xafa30fe0

```

Going back to our function, we now see that the dataflow analysis has taken care of the opaque predicate! (and left two more of them in its wake...)

```

004321f1 int32_t eax = sub_434300(&var_c0)
004321fb data_443e24 = eax
00432339 int32_t eax_1
00432339 eax_1.b = eax == 0
? 00432368 if (eax_1 == 0)
00432368 |   jump(data_441014 + (0x7855065a ^ data_441018) + 1)
? 00432eb0 if (eax_1 == 1)
00432eb0 |   jump(data_441504 + (0x67d326c7 ^ data_44150c) + 1)

```

We'll have to go and do this a *whole bunch of times*, but it is still much better than calculating the location of the jump and patching it all manually (by a long shot).

After patching up the functions called by the main method, we have a much cleaner look at the binary. Let's move our focus over to the function located at `0x409f50`.

API Hash Resolution

```

uint32_t __convention("regparm") sub_409f50(int32_t arg1)
⚠ This function has unresolved stack usage. View graph of stack usage to resolve.

00409f5e    int32_t var_3c = arg1
00409f5f    int32_t* ebx = &var_3c
00409f61    int32_t var_40 = arg1
00409f62    int32_t* var_30 = &var_40
00409f65    int32_t var_44 = arg1
00409f66    int32_t* esi = &var_44
00409f68    int32_t var_48 = arg1
00409f69    int32_t* var_1c = &var_48
00409f6c    int32_t var_4c = arg1
00409f6d    int32_t* edi = &var_4c
00409f6f    int32_t var_50 = arg1
00409f70    int32_t* var_34 = &var_50
00409f79    void var_150
00409f79    void* var_18 = &var_150
00409f82    void var_350
00409f82    void* var_14 = &var_350
00409fa2    data_44318c = sub_434a60(data_4431bc, 0xcb63c52c)
00409fc2    data_443190 = sub_434a60(data_4431bc, 0x58ae9742)
00409fe2    data_443194 = sub_434a60(data_4431bc, 0x681c6d55)
0040a002    data_443198 = sub_434a60(data_4431bc, 0xd5d0b043)
0040a022    data_44319c = sub_434a60(data_4431bc, 0xe9023384)
0040a046    data_4431a0 = sub_434a60(data_4431bc, 0x12c6785a)
0040a066    data_4431a4 = sub_434a60(data_4431bc, 0xd80a5fbf)
0040a086    data_4431a8 = sub_434a60(data_4431bc, 0x8a9fb04a)
0040a0a6    data_4431ac = sub_434a60(data_4431bc, 0x996d52fb)
0040a0c6    data_4431b0 = sub_434a60(data_4431bc, 0x15efbf48)
0040a0e6    data_4431b4 = sub_434a60(data_4431bc, 0x5062dfe3)
0040a10a    data_4431b8 = sub_434a60(data_4431bc, 0xbd732396)
0040a118    *var_30 = 0x4388c4
0040a120    var_44 = 0x4388c4

```

Here, we come across a case of *API Hash Resolution*. The function `sub_434a60` is used to take a module (`data_4431bc`, which is a pointer to the base address of `WinHttp.dll`) and a corresponding hash in order to resolve a function for further use.

I won't showcase `sub_434a60` here, as it goes out of scope for this post—but this function essentially goes through the exports of `WinHttp.dll`, hashes all the function names, and returns a pointer to the function matching the provided hash.

I was able to deduce that this copy of *LummaStealer* is utilizing a hashing algorithm, namely `FNV-1a` with a modified offset. I went ahead and [added this modified hashing algorithm](#) to the `hashdb` project.

Now that the modified hashing algorithm has been deployed within `hashdb`, we can go ahead and simply utilize the `hashdb` plugin within Binary Ninja to find the names of the APIs used:

```

00409fa2 WinHttpOpen = mal::resolve_func_by_hash(module_WinHttp, WinHttpOpen);
00409fc2 WinHttpConnect = mal::resolve_func_by_hash(module_WinHttp, WinHttpConnect);
00409fe2 WinHttpOpenRequest = mal::resolve_func_by_hash(module_WinHttp, WinHttpOpenRequest);
0040a002 WinHttpCrackUrl = mal::resolve_func_by_hash(module_WinHttp, WinHttpCrackUrl);
0040a022 WinHttpSetTimeouts = mal::resolve_func_by_hash(module_WinHttp, WinHttpSetTimeouts);
0040a046 WinHttpAddRequestHeaders = mal::resolve_func_by_hash(module_WinHttp, WinHttpAddRequestHeaders);
0040a066 WinHttpSendRequest = mal::resolve_func_by_hash(module_WinHttp, WinHttpSendRequest);
0040a086 WinHttpReceiveResponse = mal::resolve_func_by_hash(module_WinHttp, WinHttpReceiveResponse);
0040a0a6 WinHttpQueryDataAvailable = mal::resolve_func_by_hash(module_WinHttp, WinHttpQueryDataAvailable);
0040a0c6 WinHttpReadData = mal::resolve_func_by_hash(module_WinHttp, WinHttpReadData);
0040a0e6 WinHttpWriteData = mal::resolve_func_by_hash(module_WinHttp, WinHttpWriteData);
0040a10a WinHttpCloseHandle = mal::resolve_func_by_hash(module_WinHttp, WinHttpCloseHandle);

```

Decrypting C2 Addresses

Now that we have both the opaque predicates and API hash resolution out of the way, let's try to find the C2 addresses for *LummaStealer*.

Within the function that resolves the `WinHttp` functions, we see a variable being assigned to a list of pointers. If we investigate this further, we see that the list of pointers contains what looks to be base64 encoded strings. However, if we try to base64 decode the strings, we do not end up with readable text. Let's dig deeper to see how these strings are decrypted!

```

004388c4 char const* enc_c2_addrs[0x9] =
004388c4 {
004388c4     [0x0] = 0x439200 {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLQPqE+a1P0S7px2ZdfH1aN..."}
004388c8     [0x1] = 0x43917f {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLUNrs3dEv0TLZq2YZbGEiI..."}
004388cc     [0x2] = 0x4390fe {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLAMKQ9a0zyU7ptyY9WH0KL..."}
004388d0     [0x3] = 0x43907d {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLR0qQ3b0vpS61z1Y1fBkCU..."}
004388d4     [0x4] = 0x438ffc {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLH0rw3e17oTbt1z41PGVaC..."}
004388d8     [0x5] = 0x438f7b {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLG070gfFnzSrFy1YBdCVeG..."}
004388dc     [0x6] = 0x438efa {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLTMKcgfFziUbZy24hPGUCI..."}
004388e0     [0x7] = 0x438e79 {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLXKro5fFPyUbN114tWE0qB..."}
004388e4     [0x8] = 0x438df8 {"o1/IUhkqhz/fHLzu0mo151GcSbB872oxMzf7zImh0BLCLLs9ekPmS7Zz0oFRD0rJ..."}
004388e8 }
004388e8 data_4388e8:

```

Encrypted *LummaStealer* C2 addresses

In this case, it seems that each string is passed in as the first argument to a function at `0x00409cb0`.

```

0040a160 *(uint32_t*)current_encrypted_c2 = **(uint32_t**)_enc_c2_addrs;
0040a16a char* decrypted_c2 = c2_struct.field_18;
0040a171 mal::w_decrypt(*(uint32_t*)current_encrypted_c2, decrypted_c2);

```

Let's take a further look at that function:

```

00409cb0 int32_t mal::w_decrypt(char* current_c2_addr, uint8_t* output_passed_in)
00409cb0 {
00409cc4     uint32_t len_enc_c2 = strlen(current_c2_addr);
00409cdc     uint32_t len_b64_decoded_expected = len_decoded_buf(current_c2_addr, len_enc_c2);
00409cf8     uint8_t* output = output_passed_in;
00409cfc     int32_t len_b64_decoded = mal::b64decode(current_c2_addr, len_enc_c2, output);
00409d0d     int32_t success;
00409d0d     int32_t result;
00409d0d     if (len_b64_decoded != len_b64_decoded_expected)
00409d0d     {
00409dd4         success = 0;
00409d0d     }

```


At the beginning, we see that the length of the current encrypted C2 address is being calculated, alongside a call to a function at `0x00409e10` which calculates the length of the blob, if you were to base64 decode it. This is followed by a function that actually base64 decodes the data.

Continuing through the function, we see the following code:

```

00409d0d     else
00409d0d     {
00409d3c         void var_28;
00409d3c         __builtin_memcpy(&var_28, output_passed_in, 0);
00409d4c         int32_t __saved_esi;
00409d4c         __builtin_memcpy(&__saved_esi, &output_passed_in[0x20], 0);
00409d51         int32_t len_c2_result = 0;
00409d61         while (true)
00409d61         {
00409d61             int32_t finished;
00409d61             finished = len_c2_result < (len_b64_decoded - 0x20);
00409d66             if ((finished & 1) == 0)
00409d66             {
00409d66                 break;
00409d66             }
00409d74             uint32_t current_char = ((uint32_t)output_passed_in[(0x20 + len_c2_result)]);
00409d7f             uint32_t ecx_5 = ((uint32_t)*(uint8_t*)&var_28 + (len_c2_result & 0x1f));
00409d9e             output_passed_in[len_c2_result] = (current_char - (((int8_t)(ecx_5 & ((current_char ^ ecx_5) ^ 0xffffffff) << 1)) - ecx_5));
00409db0             len_c2_result = (0 - ((0 - len_c2_result) - 1));
00409d61         }
00409dbe         output_passed_in[(len_b64_decoded - 0x20)] = 0;
00409dc6         result = (len_b64_decoded - 0x20);
00409dca         success = 1;
00409d0d     }
00409de9     if (success == 0)
00409de9     {
00409deb         result = 0;
00409df3         int32_t var_3c = 1;
00409de9     }
00409e0a     return result;
00409cb0 }

```

This code takes the first **32 (0x20)** bytes of the decoded blob as a key and XORs the rest of the data with it. The resulting output is a C2 address for *LummaStealer!*

With this in mind, I set up the following Binary Refinery pipeline in order to decrypt the *LummaStealer* C2 addresses:

```

ef stage4.bin \
| vsnip 0x438df8:0x451 \
| carve b64 -n 5 [ \
  | b64 \
  | push [ \
    | snip :32 \
    | pop key \
  ] \
  | snip 32: \
  | xor var:key \
  | defang \
  | cfmt "{}\n" \
]

```

```

associationokeo[.]shop
turkeyunlikelyofw[.]shop
pooreveningfuseor[.]pw
edurestunningcrackyow[.]fun
detectordiscusser[.]shop
relevantvoicelesskw[.]shop
colorfulequalugliess[.]shop
wisemassiveharmonious[.]shop
sailsystemeyesusjw[.]shop

```

Unit	Name	Definition
------	------	------------

Unit	Name	Definition
ef	Emit File	Places a file into the pipeline
vsnip	Virtual Snip	Snips (extracts) data from PE/ELF/MACHO files based on virtual offset
carve	Carve	Extracts pieces of the pipeline that matches a given format—in this case, base64 with a minimum length of 5 characters
b64	Base64	Base64 decodes each chunk in the pipeline
push	Push	Temporarily sets aside the current chunk of data and replaces it with a new chunk. This is useful if you want to perform operations on a piece of data while keeping the original data intact for later use. Think of this as a way to create a copy of the data in order to do some work on the data, before restoring the original data.
snip	Snip	On the copy of the data, retrieves (snips) the first 32 bytes, which is the XOR key
pop	Pop	Places the modified copy of the data into a meta-variable. Meta-variables can be later utilized with the <code>var</code> keyword
snip	Snip	On the original data, retrieves (snips) everything after the first 32 bytes, which is the encrypted C2 address
xor	XOR	Performs an exclusive-or operation on the data within the chunk with the popped key
defang	Defang	Defangs indicators within the pipeline
cfmt	Convert to ForMaT	Transforms each chunk in the pipeline by applying a string format operation

And now, we can happily say that we *actually* know what this infection chain is, how it works, and we've successfully retrieved the final payload and its C2 addresses. Thanks for reading! ❤️

Indicators of Compromise:

IoC	Description
https://github.com/ravindrauppalapati/RoleManager/releases/tag/Client	Sample Download URL
8ed6a84101dfcafeac6ddb5020312b0094576fd3f9106f7df460e1b8a7bd5e1	Sample ZIP
94edf5396599aaa9fca9c1a6ca5d706c130ff1105f7bd1acff83aff8ad513164	<i>QuartzDahlia</i> EXE
515d025ba2aa1096f65c13569de283b83d86824d08ca48c1fc3bc407d4cf3266	<i>AmberAmethystDaisy</i> PHB
194.147.35[.]251	<i>AmberAmethystDaisy</i> Event Server
https://pastebin.com/raw/md5jVrEB	<i>AmberAmethystDaisy</i> Dead-Drop

IoC	Description
https://t.me/+JBdY0q1mUogwZWMy	<i>AmberAmethystDaisy</i> Telegram
http://78.47.105.28/auto/b0573cef5fbfef5a15e8a6527080ad25/93.exe	<i>QuartzBegonia</i> Download URL
5b751d8100bbc6e4c106b4ef38f664fb031c86f919c3e2db59a36c70c57f54e0	<i>QuartzBegonia</i> EXE
0cf55c7e1a19a0631b0248fb0e699bbec1d321240208f2862e37f6c9e75894e7	<i>DiamondDaffodil</i> Shellcode
d6a40534d8a76509605e67ead55ef3506050c7df86701db13443d091c7a4bce2	<i>LummaStealer</i> EXE
associationokeo[.]shop	<i>LummaStealer</i> C2
turkeyunlikelyofw[.]shop	<i>LummaStealer</i> C2
pooreveningfuseor[.]pw	<i>LummaStealer</i> C2
edurestunningcrackyow[.]fun	<i>LummaStealer</i> C2
detectordiscusser[.]shop	<i>LummaStealer</i> C2
relevantvoicelesskw[.]shop	<i>LummaStealer</i> C2
colorfulequalugliess[.]shop	<i>LummaStealer</i> C2
wisemassiveharmonious[.]shop	<i>LummaStealer</i> C2
sailsystemeyusjw[.]shop	<i>LummaStealer</i> C2

P.S - Huge thanks to my friend [donaldduck8](#) for proofreading this post, be sure to check out his blog at <https://sinkhole.dev>