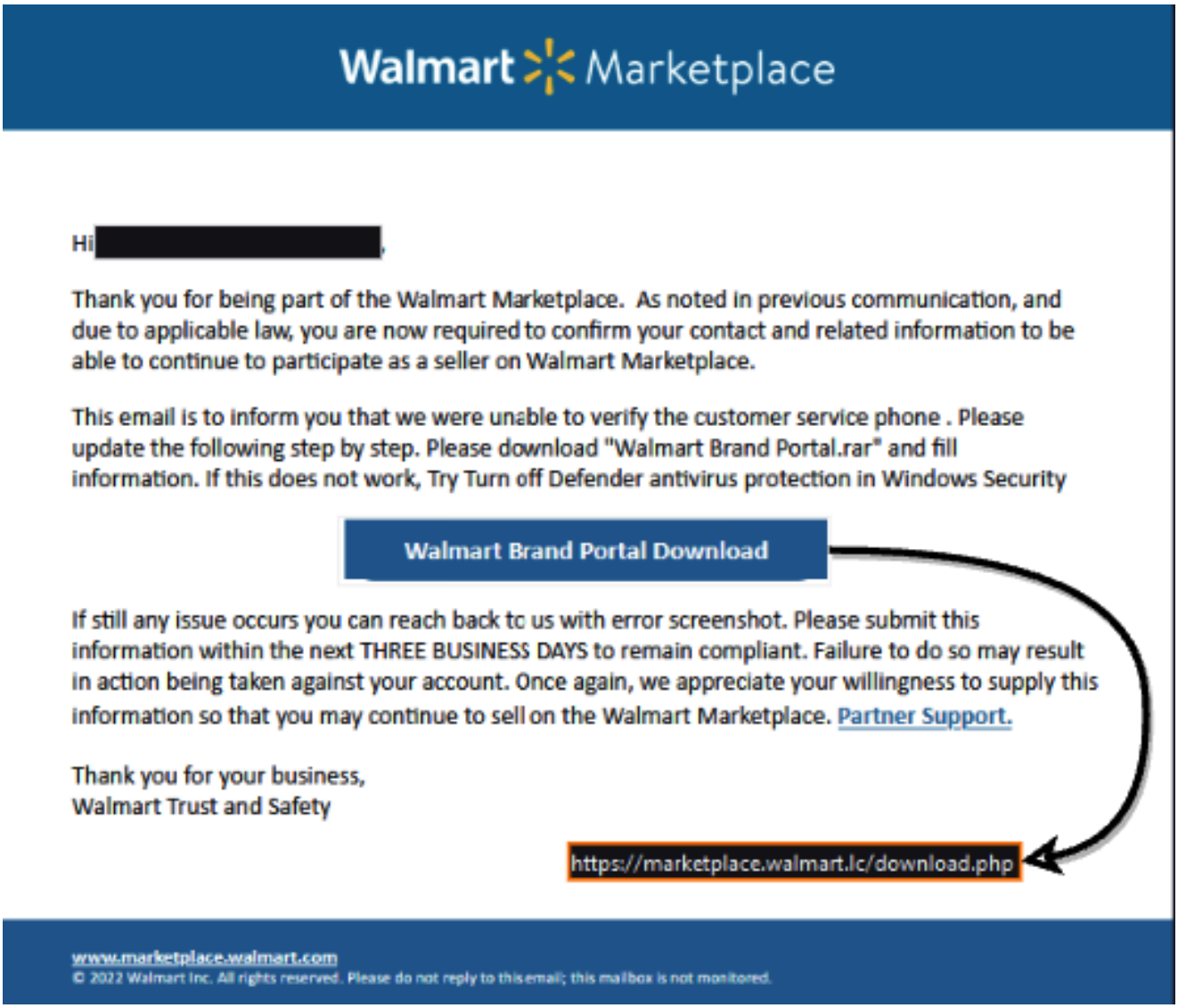
# LummaC2 BreakDown

0xtoxin-labs.gitbook.io/malware-analysis/malware-analysis/lummac2-breakdown

This blog will be a bit different from my usual blogs, it will mainly contain scripts and some research I've spent on finding some of the things you'll read through the blog. I've tried to cover things that weren't covered in previous blogs that can be found on Lumma Stealer Malpedia entry

## The Phish

The phishing email pretends to be from Walmart and targets sellers on the Walmart Marketplace. The email claims that the recipient needs to confirm their contact and related information in order to continue selling on the platform. The email instructs the recipient to download a file called **"Walmart Brand Portal.rar"** to update their information and suggests disabling antivirus protection if the download doesn't work. Clearly it's not a mail sent from Walmart and the archive will contain a malicious executable in it.



Phishing Mail

## Dynamic Triage Procedure

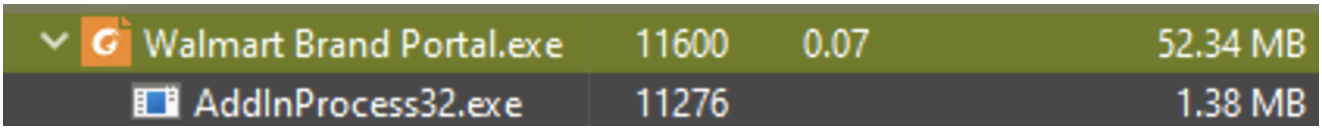
In many cases when I'm analyzing malwares I want to reach to the final payload rather than dealing with the initial loader binary. Every analyst has it's own tricks of how would he find and dump the actual piece of malware that he wants to analyze; And I will share what is my favorite tool when I want to get my hands quickly on the final payload.

### PE-Sieve

PE-Sieve is a great tool created by hasherezade which: ***"Scans a given process. Recognizes and dumps a variety of potentially malicious implants (replaced/injected PEs, shellcodes, hooks, in-memory patches)."***

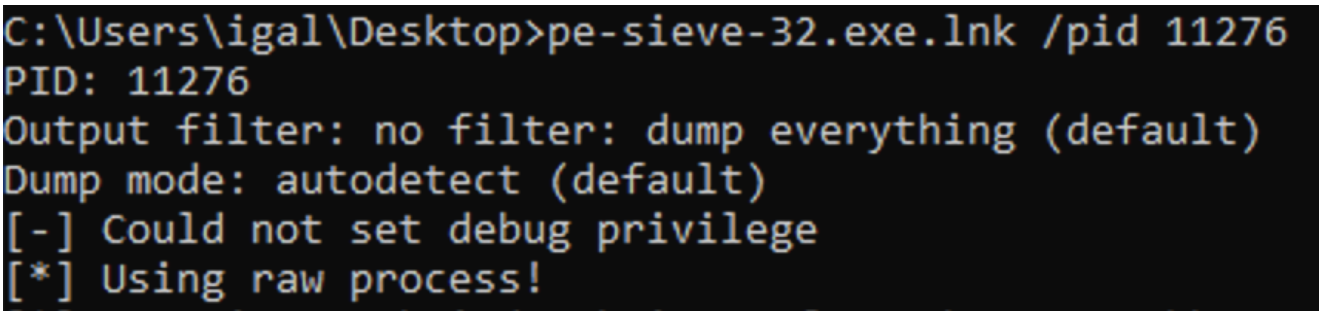
### Dump Lumma Binary

So in order to dump the Lumma binary I will execute the executable which is stored inside of the archive delivered by the phishing email, I will monitor the executable activity in Process hacker and wait for some internal process to be created as part of the injection process. (in this case the injected process will be **AddInProcess32.exe**) **Step 1 - Execute the payload:**



Dynamic Execution

**Step 2 - Use PE-Sieve on the Injected Process:**



PeSieve Execution

## Into The Lumma

Now that we have the dumped payload, I will be going through some of the functionalities and capabilities that the Lumma Stealer has.

### Control Flow Flattening

Lumma's developer added some CFF to the stealer code in order to make some hard time on reversers to find their way through the right execution flow of the malware. There are a lot of blogs talks about this obfuscation technique and how threat actors and malware developers leverages this technique to slow down malware reversers (you can find several by the end of the blog under the **References** section)

```
while ( 1 )
{
  while ( 1 )
  {
    while ( 1 )
    {
      while ( i <= -351597348 )
      {
        if ( i <- -903780410 )
        {
          if ( i > -1788248925 )
          {
            if ( i > -1339127445 )
            {
              if ( i == -1339127444 )
              {
                sub_435930(v62, v63, v64, 1, v76);
                v61 = mw_stringDecrypt(L"Wall576xedets/Ele576xedctrum");
                lpString = mw_stringDecrypt(L"*576xed");
                v59 = (int)mw_stringDecrypt((wchar_t *)"%");
                i = 751477106;
              }
              else
              {
                sub_446338(v4, v5, v70);
                i = -275705997;
              }
            }
            else if ( i == -1788248924 )
            {
              v68 = mw_stringDecrypt(L"Kom576xedeta");
              i = -1359451683;
            }
            else
            {
              v67 = (int)mw_stringDecrypt((wchar_t *)"%");
              i = 1217367503;
            }
```

CFF in Lumma's binary

I've used SophosLabs emotet_unflatten_poc plugin in order to try and clean the decompiler code abit , it helped by classifying each section as a Label and now it's abit more accessible and not requiring to scroll over the function alot:

```
LABEL_63:
          sub_406A66();
          sub_4043A6();
          sub_404EA7();
          sub_445297(2, v72);
          v34 = mw_stringDecrypt(L"Chr576xedome");
          v35 = mw_stringDecrypt((wchar_t *)"%");
          sub_4212C9(v36, v37, (int)v35, v34, v72);
          sub_446338(v38, v39, (int *)v72);
          sub_445297(2, v75);
LABEL_40:
          v11 = mw_stringDecrypt(L"Chromi576xedum");
          v12 = mw_stringDecrypt((wchar_t *)"%");
          sub_4212C9(v13, v14, (int)v12, v11, v75);
          v15 = mw_stringDecrypt((wchar_t *)"F");
          v16 = mw_stringDecrypt(L"%locala576xedppdata%\\Mic576xedrosoft\\Edge\\Us576xeder Data");
          sub_4212C9(v17, v18, (int)v16, v15, v75);
LABEL_71:
          v68 = mw_stringDecrypt(L"Kom576xedeta");
LABEL_36:
          v67 = (int)mw_stringDecrypt((wchar_t *)"%");
LABEL_78:
          sub_4212C9(v4, v5, v67, v68, v75);
          v48 = mw_stringDecrypt((wchar_t *)"O");
          v49 = mw_stringDecrypt(L"%appd576xedata%\\Ope576xedra Soft576xedware\\Op576xedera Sta576xedble");
          sub_4212C9(v50, v51, (int)v49, v48, v75);
LABEL_76:
          v44 = mw_stringDecrypt(L"Op576xedera G576xedX Stab576xedle");
          v45 = mw_stringDecrypt(L"%appd576xedata%\\Op576xedera Softw576xedare\\Op576xedera GX Sta576xedble");
          sub_4212C9(v46, v47, (int)v45, v44, v75);
```

Clearer Code

## Strings Obfuscation

I took a brief look at the strings presented in the extracted payload and witnessed that alot of them are obfuscated:

| | | | | |
|---|---|---|---|---|
| s | .rdata:0049F35C | 0000002A | C (16... | Me576xedtaMa576xedsk |
| s | .rdata:0049F388 | 00000058 | C (16... | jbalbako576xedplchlghecda576xedlmeeeajnimhm |
| s | .rdata:0049F3E0 | 0000005A | C (16... | nkbihfbeo576xedgaeaoehlef576xednkodbefgpgknn |
| s | .rdata:0049F43C | 00000028 | C (16... | ro576xednLi576xednk |
| s | .rdata:0049F464 | 0000005A | C (16... | ibnejdfjmmk576xedpcnlpebklmnk576xedoeoihofec |
| s | .rdata:0049F4C0 | 00000030 | C (16... | on576xedin Wall576xedet |
| s | .rdata:0049F4F0 | 0000005A | C (16... | fnjhmkhhmkb576xedjkkabndcn576xednogagogbneec |
| s | .rdata:0049F54C | 0000004C | C (16... | in576xedance Cha576xedin Wal576xedlet |
| s | .rdata:0049F598 | 0000005A | C (16... | fhbohimaelboh576xedpjbbldcngcnapn576xeddodjp |
| s | .rdata:0049F5F4 | 00000016 | C (16... | o576xedroi |
| s | .rdata:0049F60C | 00000058 | C (16... | fn576xedbelfdoeiohenk576xedjibnmadjiehjhajb |
| s | .rdata:0049F664 | 00000018 | C (16... | Ni576xedfty |
| s | .rdata:0049F67C | 0000005A | C (16... | jbd576xedaocneiiinmjbj576xedlgalhcelgbejmnid |
| s | .rdata:0049F6D8 | 00000014 | C (16... | a576xedth |
| s | .rdata:0049F6EC | 0000005A | C (16... | afbc576xedbjpbpfadlkmhm576xedclhkeeodmamcflc |
| s | .rdata:0049F748 | 0000001C | C (16... | oinb576xedase |
| s | .rdata:0049F764 | 0000005A | C (16... | hnfanknocfe576xedofbddgcijnm576xedhnfnkdnaad |

Obfuscated Strings

After doing a small research, I found out the Lumma obfuscates the strings by inserting the string `576xed` inside of them. Those strings are being deobfuscated by dedicated function which requires the obfuscated string as an input and afterward it returns the clean string.

```
v15 = mw_stringDecrypt((wchar_t *)"E");
v16 = mw_stringDecrypt(L"%locala576xedppdata%\\Mic576xedrosoft\\Edge\\Us576xeder Data");
```

Deobfuscation Function

I've created python script that will get all the xrefs for the strings deobfuscating function, extract the argument being passed to the function and then will deobfuscate it and write the strings to a file:

```
import idc

import idautils


DECRYPTION_FUNCTION = 0x45DF86 # Change to the relevant function call

STRINGS_FILE_PATH = '' # Output file for the strings

OBFUSCATOR_STRING = '576xed' # Might be changed in future builds


def getArg(ref_addr):

ref_addr = idc.prev_head(ref_addr)

if idc.print_insn_mnem(ref_addr) == 'push':

if idc.get_operand_type(ref_addr, 0) == idc.o_imm:

return(idc.get_operand_value(ref_addr, 0))

else:

return None


stringsList = []


for xref in idautils.XrefsTo(DECRYPTION_FUNCTION):
```

```
argPtr = getArg(xref.frm)

if not argPtr:

continue

data = idc.get_bytes(argPtr, 100)

obfuscatedData = data.split(b'\x00\x00')[0].replace(b'\x00',b'').decode()

stringsList.append(obfuscatedData.replace(OBFUSCATOR_STRING,""))


print(f'[+] {len(stringsList)} Strings were extracted')


out = open(STRINGS_FILE_PATH, 'w')

for string in stringsList:

out.write(f'{string}\n')

out.close()

[+] 135 Strings were extracted


#################

# OUTPUT FILE: #

#################


\Local Extension Settings\

/Extensions/

*

nkddgncdjgjfcddamfgcmfnlhccnimig

NeoLine

cphhlgmgameodnhkjdmkpanlelnlohao
```

Clover

nhnkbkgjikgcigadomkphalanndcapjk

Liquality

kpfopkelmapcoipemfendmdcghnegimn

Terra Station

fhmfendgdocmcbmfikdcogofphimnkno

Auro

cnmamaachppnkjgnildpdmkaakejnhae

aeachknmefphepccionboohckonoeemg

Authenticator

bhghoamapcdpbohphigoooaddinpkbai

Cyano

dkdedlpgdmmkkfjabffeganieamfklkm

Byone

Login Data For Account

OneKey

Nifty

jbdaocneiiinmjbjlgalhcelgbejmnid

Math

iWlt

kncchdigobghenbbaddojjnnaogfppfj

EnKrypt

kkpllkodjeloidieedojogacfhpaihoh

Wombat

amkmjjmmfldd ogmhpjloimipbofnfjih

MEW CX

nlbmnnijcnlegkjjpcfjclmcfggfefdm

Guild

nanjmdknhkinifnkgdcggcfnhdaammmj

Coin98

infeboajgfhgbjpjbeppbkgnabfdkdaf

Leaf

cihmoadaighcejopammfbmddcmdekcje

Authy

ejbalbakoplchlghecdalmeeeajnimhm

nkbihfbeogaeaoehlefnkodbefgpgknn

TronLink

ibnejdfjmmkpcnlpebklmnkoeoihofec

Ronin Wallet

fnjhmkhhmkbjkkabndcnnogagogbneec

Binance Chain Wallet

fhbohimaelbohpjbbldcngcnapndodjp

Yoroi

afbcbjpbpfadlkmhmclhkeeodmamcflc

gaedmjdfmmahhbjefcbgaolhhanlaolb

Saturn

bcopgchhojmggmffilplmbdicgaihlkp

ZilPay

klnaejjgbibmhlephnhpmaofohgkpgkd

Phantom

bfnaelmomeimhlpmgjnjophhpkkoljpa

hcflpincpppdclinealmandijcmnkbgn

Temple

ookjlbkiijinhpmnjffcofjonbfbgaoc

TezBox

mnfifefkajgofkcjkemidiaecocnkjeh

DAppPlay

lodccjjbdhfakaekdiahmedfbieldgik

BitClip

ijmpgkjfkbfhoebgogflfebnmejmfbml

Steem Keychain

History

Jaxx Liberty

cjelfplplebdjjenllpjcblmjkfcffne

BitApp

fihkakfobkmkjojpchpfgcmhfjnmnfpi

Network\Cookies

History

Polymesh

jojhfeoedkpkglbfimdfabpdfjaoolaf

ICONex

flpiciilemghbmfalicajoolhkkenfel

Nabox

ffnbelfdoeiohenkjibnmadjiehjhajb

Web Data

Login Data

aiifbnbfobpmeekipheeijimdpnlpgpp

Keplr

dmkamcknogkgcdfhhbddcghachkejeap

Sollet

nlgbhdfgdhgbiamfdfmbikcdghidoadd

Coinbase

hnfanknocfeofbddgcijnmhnfnkdnaad

Guarda

hpglfhgfnhbgpjdenjgmdgoeiappafln

EQUAL

blnieiiffboillknjnepogjhkgnoapac

lkcjlnjfpbikmcmbachjpdbijejflpcm

Nash Extension

onofpnbbkehpmmoabgpcpmigafmmnjhl

Hycon Lite Client

Trezor Password Manager

imloifkgjagghnncjkhggdhalmcnfklk

EOS Authenticator

oeljdldpnmdbchonielidgobddfffflal

GAuth Authenticator

ilgcnhelpchnceeipipijaljkblbcobl

nknhiehlklippafakaeklbeglecifhad

KHC

\Local State

*.txt

%userprofile%

Wallets/Ethereum

keystore

%appdata%\Ethereum

%localappdata%\Kometa\User

Chromium

%localappdata%\Chromium\User Data

Edge

%localappdata%\Microsoft\Edge\Us

%appdata%\Opera Software\Op576xe

Chrome

%localappdata%\Google\Chro

Mozilla Firefox

Wallets/Binance

app-store.json

%appdata%\Binance

Kometa

Important Files/Profile

Opera GX Stable

%appdata%\Opera Software\Op576xe

Opera Neon

Opera Stable

%appdata%\Opera Software\Op576xe

Wallets/Electrum

*

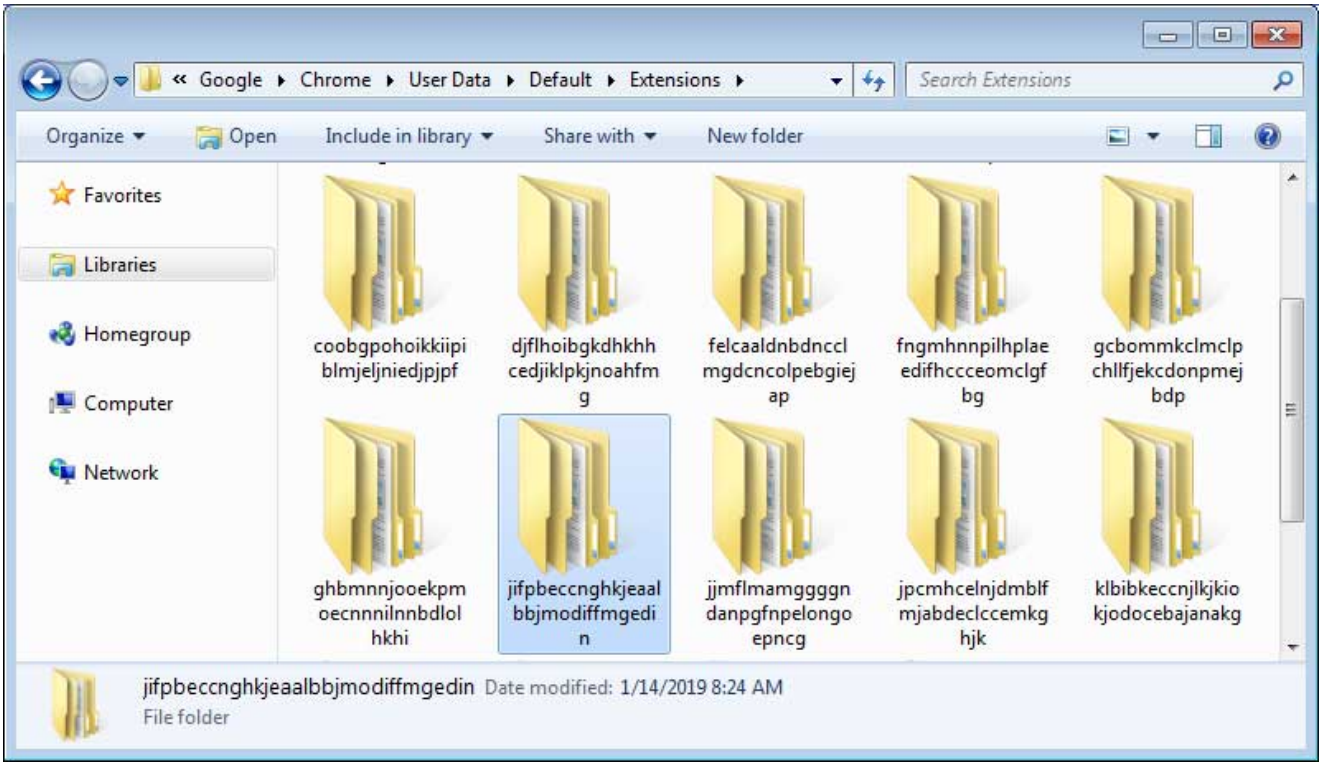
%appdata%\Electrum\wallets

%appdata%\Mozilla\Firefox\Prof57

System.txt

## Chrome Extensions (CRX)

Looking at the strings there is a lot of extensions names that Lumma targets, but the thing that I was curious about were the 32 length lower case strings (for example: `ilgcnhelpchnceeipipijaljkblbcobl`) those strings are actually CRX IDs (Chrome Extension ID) which by navigating to `C:\Users\User\AppData\Local\Google\Chrome\User Data\Default\Extensions` and looking for the CRX ID the stealer will know whether or not the extension exists on the victims computer and if it does it will continue to exfiltrate the extension sensitive data.



Chrome Extensions Local Path

I've created a python script that will lookup for those unique ID's in the previously extracted strings file and fetch the name of the extension from google chrome webstore:

import re, requests

```
LUMMA_STRINGS = '/Users/igal/malwares/Lumma/29.03.2023/LummaStrings.txt'

REGEX_PATTERN = '^[a-z]{32}$'

strings = open(LUMMA_STRINGS,'r').read()

crxExtensionList = re.findall(REGEX_PATTERN,strings,re.MULTILINE)



for crxId in crxExtensionList:

url = f'https://chrome.google.com/webstore/detail/{crxId}'

responseText = requests.get(url).text

try:

startIndex = responseText.index('itemprop="name" content="') + len('itemprop="name" content="')

endIndex = responseText.index('"', startIndex)

extensionName = responseText[startIndex:endIndex]

print(f'[+] Extension name:{extensionName} , CRX ID:{crxId}')

except ValueError:

continue
```

```
[+] Extension name:NeoLine , CRX ID:cphhlgmgameodnhkjdmkpanlelnlohao

[+] Extension name:CLV Wallet , CRX ID:nhnkbkgjikgcigadomkphalanndcapjk

[+] Extension name:Liquality Wallet , CRX ID:kpfopkelmapcoipemfendmdcghnegimn

[+] Extension name:Auro Wallet , CRX ID:cnmamaachppnkjgnildpdmkaakejnhae

[+] Extension name:Coin98 Wallet , CRX ID:aeachknmefphepccionboohckonoeemg

[+] Extension name:authenticator , CRX ID:bhghoamapcdpbohphigoooaddinpkbai

[+] Extension name:Cyano Wallet , CRX ID:dkdedlpgdmmkkfjabffeganieamfklkm

[+] Extension name:iWallet , CRX ID:kncchdigobghenbbaddojjnnaogfppfj
```

[+] Extension name:Enkrypt: Ethereum, Polkadot & Canto Wallet , CRX ID:kkpllkodjeloidieedojogacfhpaihoh

[+] Extension name:Wombat - Gaming Wallet for Ethereum & EOS , CRX ID:amkmjjmmflddogmhpjloimipbofnfjih

[+] Extension name:MEW CX - is now Enkrypt , CRX ID:nlbmnnijcnlegkjjpcfjclmcfggfefdm

[+] Extension name:LeafWallet - Easy to use EOS wallet , CRX ID:cihmoadaighcejopammfbmddcmdekcje

[+] Extension name:MetaMask , CRX ID:nkbihfbeogaeaoehlefnkodbefgpgknn

[+] Extension name:TronLink , CRX ID:ibnejdfjmmkpcnlpebklmnkoeoihofec

[+] Extension name:Ronin Wallet , CRX ID:fnjhmkhhmkbjkkabndcnnogagogbneec

[+] Extension name:Binance Wallet , CRX ID:fhbohimaelbohpjbbldcngcnapndodjp

[+] Extension name:Math Wallet , CRX ID:afbcbjpbpfadlkmhmclhkeeodmamcflc

[+] Extension name:Authy , CRX ID:gaedmjdfmmahhbjefcbgaolhhanlaolb

[+] Extension name:Hycon Lite Client , CRX ID:bcopgchhojmggmffilplmbdicgaihlkp

[+] Extension name:ZilPay , CRX ID:klnaejjgbibmhlephnhpmaofohgkpgkd

[+] Extension name:Phantom , CRX ID:bfnaelmomeimhlpmgjnjophhpkkoljpa

[+] Extension name:KHC , CRX ID:hcflpincpppdclinealmandijcmnkbgn

[+] Extension name:Temple - Tezos Wallet , CRX ID:ookjlbkiijinhpmnjffcofjonbfbgaoc

[+] Extension name:TezBox - Tezos Wallet , CRX ID:mnfifefkajgofkcjkemidiaecocnkjeh

[+] Extension name:DAppPlay , CRX ID:lodccjjbdhfakaekdiahmedfbieldgik

[+] Extension name:Polymesh Wallet , CRX ID:jojhfeoedkpkglbfimdfabpdfjaoolaf

[+] Extension name:ICONex , CRX ID:flpiciilemghbmfalicajoolhkkenfel

[+] Extension name:Yoroi , CRX ID:ffnbelfdoeiohenkjibnmadjiehjhajb

[+] Extension name:Station Wallet , CRX ID:aiifbnbfobpmeekipheeijimdpnlpgpp

[+] Extension name:Keplr , CRX ID:dmkamcknogkgcdfhhbddcghachkejeap

[+] Extension name:Byone , CRX ID:nlgbhdfgdhgbiamfdfmbikcdghidoadd

[+] Extension name:Coinbase Wallet extension , CRX ID:hnfanknocfeofbddgcijnmhnfnkdnaad

[+] Extension name:Guarda , CRX ID:hpglfhgfnhbgpjdenjgmdgoeiappafln

[+] Extension name:Trezor Password Manager , CRX ID:imloifkgjagghnncjkhggdhalmcnfklk

[+] Extension name:EOS Authenticator , CRX ID:oeljdldpnmdbchonielidgobddfffflal

[+] Extension name:GAuth Authenticator , CRX ID:ilgcnhelpchnceeipipijaljkblbcobl

[+] Extension name:Nabox Wallet , CRX ID:nknhiehlklippafakaeklbeglecifhad

## Dynamic API Resolve

Lumma hides some of its APIs by hashing then using the **MurmurHash2 hashing algorithm** , it can be identified by the const: `0x5bd1e995`:

| Address | Function | Instruction |
|---|---|---|
| .text:00472D51 | mw_MurmurHash2 | imul eax, [ebp+var_254], 5BD1E995h |
| .text:00472D5E | mw_MurmurHash2 | imul eax, [ebp+var_34], 5BD1E995h |
| .text:00472F45 | mw_MurmurHash2 | imul eax, [ebp+var_28], 5BD1E995h |
| .text:004736BF | mw_MurmurHash2 | imul eax, [ebp+var_A0], 5BD1E995h |
| .text:00473707 | mw_MurmurHash2 | imul eax, [ebp+var_34], 5BD1E995h |
| .text:004758CF | mw_MurmurHash2 | imul eax, [ebp+var_E4], 5BD1E995h |
| .text:00476437 | mw_MurmurHash2 | imul eax, [ebp+var_2DC], 5BD1E995h |
| .text:0047716C | mw_MurmurHash2 | imul eax, [ebp+var_58], 5BD1E995h |
| .text:004778C9 | mw_MurmurHash2 | imul eax, [ebp+var_34], 5BD1E995h |
| .text:00478594 | mw_MurmurHash2 | imul eax, ecx, 5BD1E995h |
| .text:00478A0A | mw_MurmurHash2 | imul ecx, [ebp+var_250], 5BD1E995h |

Hashing Const References

Lumma will pass two arguments to the API resolving function:

1. 1.
   The hash of the wanted API
2. 2.
   The DLL which contains the API

```
int __cdecl mw_apiResolve(int v_APIhash, int v_APIDLL)
{
  int v4; // esi
  int v5; // eax
  int v6; // edx
  int v7; // ecx
  int (__stdcall *loadLibrary)(int); // eax
  int v9; // edx
  int v10; // ecx
  int v11; // eax
  int v13; // [esp+Eh] [ebp-18h]

  v5 = mw_PEBwalk();
  loadLibrary = (int (__stdcall *)(int))mw_MurmurWrapper(v6, v7, v5, 0xAB87776C);
  v13 = loadLibrary(v_APIDLL);
  v11 = 0xE84852C6;
  if ( !v_APIhash )
    v11 = 0x79F33DD2;
  v4 = 0;
  if ( v11 == 0xE84852C6 )
    return mw_MurmurWrapper(v9, v10, v13, v_APIhash);
  return v4;
}
```

API Resolving Function

I have two scripts that I wrote for this part, the first script is IDA script that will get all the xrefs to the API resolving function, extract the hash and the API hash, and will save the output to a text file:

```
import idc

import idautils


API_FUNCTION = 0x471958 # Change to the relevant function call

APIS_FILE_PATH = '' # Output file for the strings


apiDict = {}


def getDLLRef(hash_addr):

ref_addr = idc.prev_head(hash_addr)

if idc.print_insn_mnem(ref_addr) == 'push':
```

```
if idc.get_operand_type(ref_addr, 0) == idc.o_imm:
dll_addr = idc.get_operand_value(ref_addr, 0)
return idc.get_bytes(dll_addr, 50).split(b'\x00\x00')[0].replace(b'\x00',b'').decode()
return None

def getHashDict(ref_addr):
ref_addr = idc.prev_head(ref_addr)
if idc.print_insn_mnem(ref_addr) == 'push':
if idc.get_operand_type(ref_addr, 0) == idc.o_imm:
hashVal = hex(idc.get_operand_value(ref_addr, 0))
if hashVal not in apiDict or apiDict[hashVal] == None:
dllVal = getDLLRef(ref_addr)
apiDict[hashVal] = dllVal

for xref in idautils.XrefsTo(API_FUNCTION):
getHashDict(xref.frm)

print(f'[+] {len(apiDict)} API hashes were extracted')

out = open(APIS_FILE_PATH, 'w')
for k, v in apiDict.items():
out.write(f'{k} - {v}\n')
out.close()
```

```
[+] 18 API hashes were extracted
```

################

# OUTPUT FILE: #

################

0xe8ff1073 - crypt32.dll

0x864087d1 - crypt32.dll

0x7328f505 - kernel32.dll

0xc40f97d4 - advapi32.dll

0x507048c2 - winhttp.dll

0x406457c2 - winhttp.dll

0x7aa0edcc - winhttp.dll

0xb72f0de - winhttp.dll

0x59886bc0 - winhttp.dll

0x76b029a - winhttp.dll

0xf9f57cf0 - winhttp.dll

0xe268a0c1 - winhttp.dll

0xab3372e8 - winhttp.dll

0x5658bf2e - KernelBase.dll

0x23fef64a - advapi32.dll

0x5f086d32 - kernel32.dll

0xa2f80070 - kernel32.dll

0x2f9959e0 - kernel32.dll

The second script will extract the hashes and the DLLs names from the text file, and based on the DLL name it will be loaded using PEfile, and iterate through the DLL exports , hash them using murmurhash2 and compare it to the given hash: (**NOTE:** The seed in this case was `0x20` but it might be possible changed in future builds)

```
import pefile

from murmurhash2 import murmurhash2


DLLS_PATH = '/Users/igal/malwares/Lumma/29.03.2023/dlls/' # can be replaced with system32 folder

LUMMA_API_HASHES = '/Users/igal/malwares/Lumma/29.03.2023/LummaApiHashes.txt'


SEED = 0x20 # might be changed in upcoming builds


def hashDllAPI(dllName, apiHash):

pe = pefile.PE(dllName)

for export in pe.DIRECTORY_ENTRY_EXPORT.symbols:

try:

expName = export.name

hashValue = murmurhash2(expName, SEED)

if hex(hashValue) == apiHash:

return expName.decode()

except AttributeError:

continue


apiHashesFile = open(LUMMA_API_HASHES,'r').read()

lines = apiHashesFile.split('\n')
```

```
for line in lines:

args = line.split(' - ')

dllName = hashDllAPI(f'{DLLS_PATH}{args[1]}', args[0])

print(f'[+] {args[0]} - {dllName}')
```

```
[+] 0xe8ff1073 - CryptStringToBinaryA

[+] 0x864087d1 - CryptUnprotectData

[+] 0x7328f505 - ExpandEnvironmentStringsW

[+] 0xc40f97d4 - GetCurrentHwProfileA

[+] 0x507048c2 - WinHttpOpenRequest

[+] 0x406457c2 - WinHttpConnect

[+] 0x7aa0edcc - WinHttpCloseHandle

[+] 0xb72f0de - WinHttpSendRequest

[+] 0x59886bc0 - WinHttpWriteData

[+] 0x76b029a - WinHttpReceiveResponse

[+] 0xf9f57cf0 - WinHttpOpen

[+] 0xe268a0c1 - WinHttpSetTimeouts

[+] 0xab3372e8 - WinHttpAddRequestHeaders

[+] 0x5658bf2e - IsWow64Process2

[+] 0x23fef64a - GetUserNameA

[+] 0x5f086d32 - GetPhysicallyInstalledSystemMemory

[+] 0xa2f80070 - GetComputerNameA

[+] 0x2f9959e0 - GetSystemDefaultLocaleName
```

## Yara Rule

```
rule Win_LummaC2 {

meta:

author = "0xToxin"

description = "LummaC2 Strings"

strings:

$obfuscatorString = "576xed" ascii wide

$s1 = "dp.txt" ascii wide

$s2 = "c2sock" ascii wide

$s3 = "TeslaBrowser" ascii wide

$s4 = "Software.txt" ascii wide

condition:

uint16(0) == 0x5a4d and all of ($s*) and #obfuscatorString > 10 and filesize < 1500KB

}
```

Yara Hunt of the rule

## Summary

In this blog I went over some techniques used by the Lumma stealer including the hashing algorithim used in the API hashing procedure, the strings obfuscation and some Google Chrome extensions research. If you want to learn more about how Lumma exfiltrates it data, check the blogs in the references below.

## IOCs

URLs:

https://marketplace.walmart[.]lc/download.php

Files:

Walmart Brand Portal.rar - d69520637a73226a61c09298295145923fc60a06584528cb1f05a530479a7a36
Walmart Brand Portal.exe - 9b9388c1b9e9417df5ca4e883ef595455932dfce24ca1dad9897d506aecdac2a
Lumma Binary - 19fefb958bd9c9280d07754ab903022a3dc9fc380a6964733a1dcc016aba8150

C2:

82.117.255[.]80/c2sock

User Agent:

TeslaBrowser/5.5

## References