Unveiling LummaC2 stealer’s novel Anti-Sandbox technique: Leveraging trigonometry for human behavior detection

November 20, 2023

Research & Threat Intel 20 Nov 2023
Written By Alberto Marín Reverse Engineering Team Lead

The Malware-as-a-Service (MaaS) model, and its readily available scheme, remains to be the preferred method for emerging threat actors to carry out complex and lucrative cyberattacks. Information theft is a significant focus within the realm of MaaS, with a specialization in the acquisition and exfiltration of sensitive information from compromised devices, including login credentials, credit card details, and other valuable information. This illicit activity represents a considerable threat that can lead to substantial financial losses for both organizations and individuals.

In this blogpost, we will take a deep dive into a new **Anti-Sandbox** technique LummaC2 v4.0 stealer is using to avoid detonation if no human mouse activity is detected. To be able to reproduce the analysis, we will also assess the **packer** and LummaC2 v4.0 new **Control Flow Flattening** obfuscation (present in all samples by default) to effectively analyze the malware. Analysis of the packer is also relevant, as the threat actor selling LummaC2 v4.0 strongly discourages spreading the malware in its unaltered form.

LummaC2 v4.0 updates

LummaC2 is an information stealer written in C language sold in underground forums since December 2022. KrakenLabs previously published an in-depth analysis of the malware assessing LummaC2’s primary workflow, its different obfuscation techniques, and how to overcome them to effectively analyze the malware with ease. The malware has since gone through different updates and is currently on version **4.0**.

Some of these significant updates include:

Some of these functionalities have been covered in recent publications.

Packer

The malware that is going to be analyzed during these lines comes from the sample *b14ddf64ace0b5f0d7452be28d07355c1c6865710dbed84938e2af48ccaa46cf*. The initial component within the sample is the **Packer**, which serves as the outer layer of LummaC2

v4.0. Its primary function is to **obfuscate** the malicious payload and facilitate its execution during runtime without the need for spawning additional processes in the system (using *CreateThread* instead). The packer's architecture consists of two distinct layers.

We will now delve into the steps the malware takes to execute the payload through these layers, as well as the various techniques it employs to hinder and slow down the analysis process.

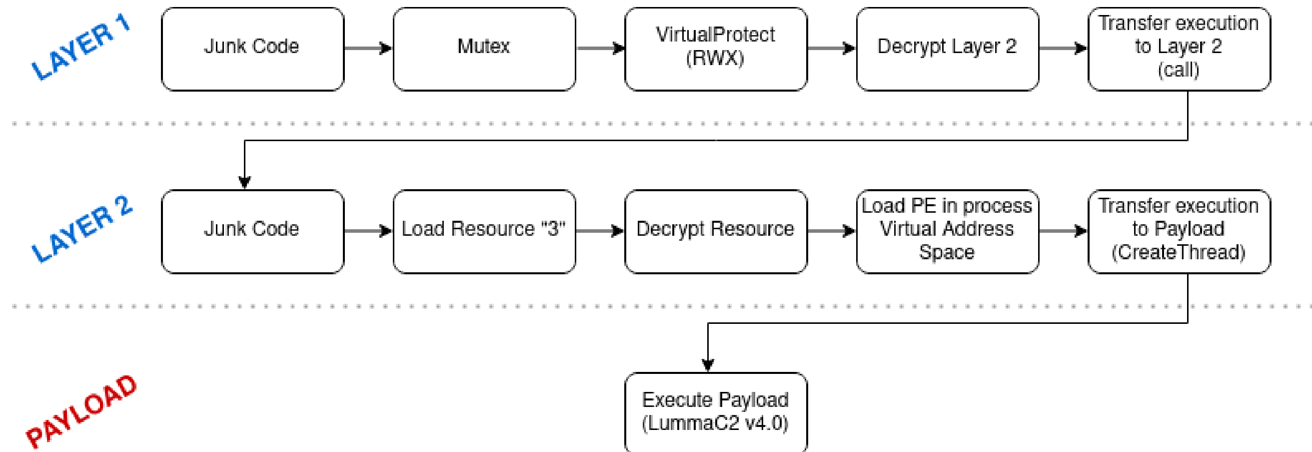

Figure 1. b14ddf64ace0b5f0d7452be28d07355c1c6865710dbed84938e2af48ccaa46cf Packer layers

Layer 1

The first layer uses a lot of assembly junk instructions, they differ between packed samples but do not execute any relevant action. Then it will use obfuscation techniques like *push+re*t and *jz+jnz* to break disassembly and complicate the analysis, as it can be seen in the following figure:

```
; void sub_40D000()
sub_40D000 proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
call    ds:FreeConsole
mov     [ebp+var_4], offset loc_402DE0
mov     eax, [ebp+var_4]
push    eax
retn
sub_40D000 endp ; sp-analysis faile
```

```
; ---------------------------------------------------------------------------

loc_402DE0:                             ; DATA XREF: sub_40D000+A↓o
                push    ebp
                mov     ebp, esp
                sub     esp, 824h
                push    ebx
                push    esi
                push    edi
                push    0
                lea     eax, [ebp-7E8h]
                push    eax
                call    ds:FindFirstVolumeW
                jz      short near ptr loc_402DFF+3
                jnz     short near ptr loc_402DFF+3

loc_402DFF:                             ; CODE XREF: .text:00402DFB↑j
                                        ; .text:00402DFD↑j
                mov     eax, 0E1B890C3h
                adc     eax, [eax+0]
                dec     eax
                call    eax
```

Figure 2. Example of push+ret and jz+jnz obfuscation techniques present in Layer 1 of the Packer

The malware then executes a lot of junk instructions that do not alter the functionality of the program and proceeds to enter in a big and useless loop; which could be considered as an "**anti-emulation"** technique. It also checks that the value calculated at the end of the loop is the one expected to ensure the loop is actually performed. It then continues by creating a Mutex to ensure no more copies of the malware run at the same time in the infected machine. The **Mutex** name varies between samples.

```
__asm
{
  rcl     esi, 0D5h
  rcr     ebx, 0C2h
  rcl     ebx, 9Ah
  rcr     edi, 49h
}
result = __ROL4__((-_ESI - (((unsigned __int64)(88i64 * -_ESI) >> 32 != 0) + 220)) * -88 * _ESI * -88 * _ESI - 1, 171)
       + 103;
__asm { rcl     edi, 9Bh }
v65 = 0;
for ( i = 0; i < 100000530; ++i )
  ++v65;
if ( v65 == 100000530 )
{
  _EDX = ((unsigned __int64)(result * (__int64)result) >> 32) & 0xFC;
  __asm { rcl     edx, 2 }
  _EAX = 83;
  __asm { rcr     eax, 0FFh }
  v34 = _byteswap_ulong(__ROL4__(__ROL4__(_EDI, 98), 102) + 38) + 1;
  do
  {
    hModule = GetModuleHandleA("kernel32.dll");
    MutexA = CreateMutexA(0, 1, "5x8Rdw2cBe7kmj9DTSXmy8BPsXlTG5FUuDw1wqYTLZdr9HAnNDQ6WCDqXiAi%");
  }
  while ( GetLastError() != 183 );
```

Junk Code

Mutex

Figure 3. Example of useless code present to difficult analysis, and Mutex creation present in Layer 1 of the Packer

After this initial logic, the packer starts by decrypting a 15 characters API from kernel32.dll. The string is encrypted and built in the stack byte per byte. The API being resolved is "*VirtualProtect*" used to give *PAGE_EXECUTE_READWRITE* protections to a fixed address that will contain the **second layer**.

```
ProcName[0] = 18;
ProcName[1] = -57;
ProcNamCHAR ProcName[16]; // [esp+20h] [ebp-810h] BYREF
ProcNam"VirtualProtect"
ProcName[4] = -99;
ProcName[5] = -16;
ProcName[6] = -59;
ProcName[7] = 56;
ProcName[8] = -79;
ProcName[9] = -65;
ProcName[10] = -83;
ProcName[11] = -24;
ProcName[12] = -10;
ProcName[13] = -77;
ProcName[14] = -122;
for ( j = 0; j < 0xF; ++j )
{
  v52 = ~ProcName[j] - j - 104;
  v53 = -((8 * v52) | ((int)v52 >> 5));
  v54 = ~(33 - ((32 * v53) | ((int)v53 >> 3)) + 79);
  v55 = j + ((v54 << 7) | ((int)v54 >> 1));
  v56 = j + j - ((v55 << 7) | ((int)v55 >> 1));
  v57 = j + ((4 * v56) | ((int)v56 >> 6));
  ProcName[j] = -(char)(j + ~((v57 << 6) | ((int)v57 >> 2))) - 97;
}
pVirtualProtect = GetProcAddress(hModule, ProcName);
```

Figure 4. Decryption of "VirtualProtect" to resolve Windows API without exposing clear string

After *VirtualProtect* is successfully executed, it proceeds to decrypt the second stage with fixed hardcoded size of **6556 bytes**. Once the layer has been decrypted, execution is transferred via *call* instruction with the fixed address of the **second layer**. The following picture shows the decryption algorithm for the **next** layer:

```
do
{
  *ArgInputBuffer = ~*ArgInputBuffer;
  *ArgInputBuffer += 0x78;
  *ArgInputBuffer += 0x49;
  result = 0;
  ++ArgInputBuffer;
  --size;
}
while ( size );
return result;
```

Figure 5. Decryption algorithm for the second Layer of the Packer

Layer 2

This layer is very similar to the first one as it uses the same obfuscation techniques to break disassembly and slow down analysis. However, this second stage will eventually extract, decrypt and execute LummaC2 v4.0. This is accomplished by loading a resource using *LoadResource* and *LockResource*. The **Resource Name** is hardcoded as “**3**” and contains the final stage (encrypted).

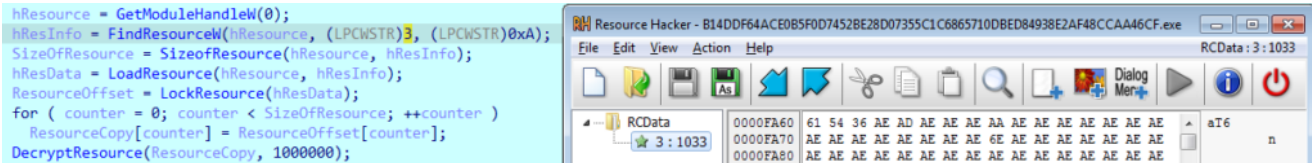

Figure 6. Last Layer is contained in the packed PE resources

The resource is decrypted using a very similar algorithm from the first layer, constants change but the **algorithm remains the same**.

```
do
{
  *ArgInputBuffer = ~*ArgInputBuffer;
  *ArgInputBuffer += 0x47;
  *ArgInputBuffer += 0x68;
  result = 0;
  ++ArgInputBuffer;
  --size;
}
while ( size );
return result;
```

Figure 7. Decryption algorithm for last Layer (LummaC2 v4.0)

Once the resource has been decrypted, the packer will check if this resource is a PE file, (it checks if “PE” magic is present at offset 0x3C from the base address of the decrypted resource). If the check fails, the program finishes abruptly. This shows how the packer is only expecting a PE in its resources.

A copy of the decrypted resource is then made into an allocated buffer. At this point, we can safely **dump** the unpacked memory just after it has been written decrypted in the allocated buffer (and before relocations are made). We will have now the unpacked LummaC2 v4.0 stealer, which we can dump for further analysis.

The packer finally loads this new PE in its process virtual address space by copying the payload and applying relocations. Finally, it will execute its **Original EntryPoint** via *CreateThread* using *NTHeaders->OptionalHeader.AddressOfEntryPoint* as the *ThreadRoutine* parameter.

```
hHandle = CreateThread(
            0,
            0,
            (LPTHREAD_START_ROUTINE)((char *)ResourceCopyDecrypted_copy + NTHeaders->OptionalHeader.AddressOfEntryPoint),
            0,
            0,
            0);
```

Figure 8. Entry Point of LummaC2 v4.0 is executed via CreateThread

This will transfer execution to the **unpacked** LummaC2 v4.0. The following figure shows the **main** routine of the malware:

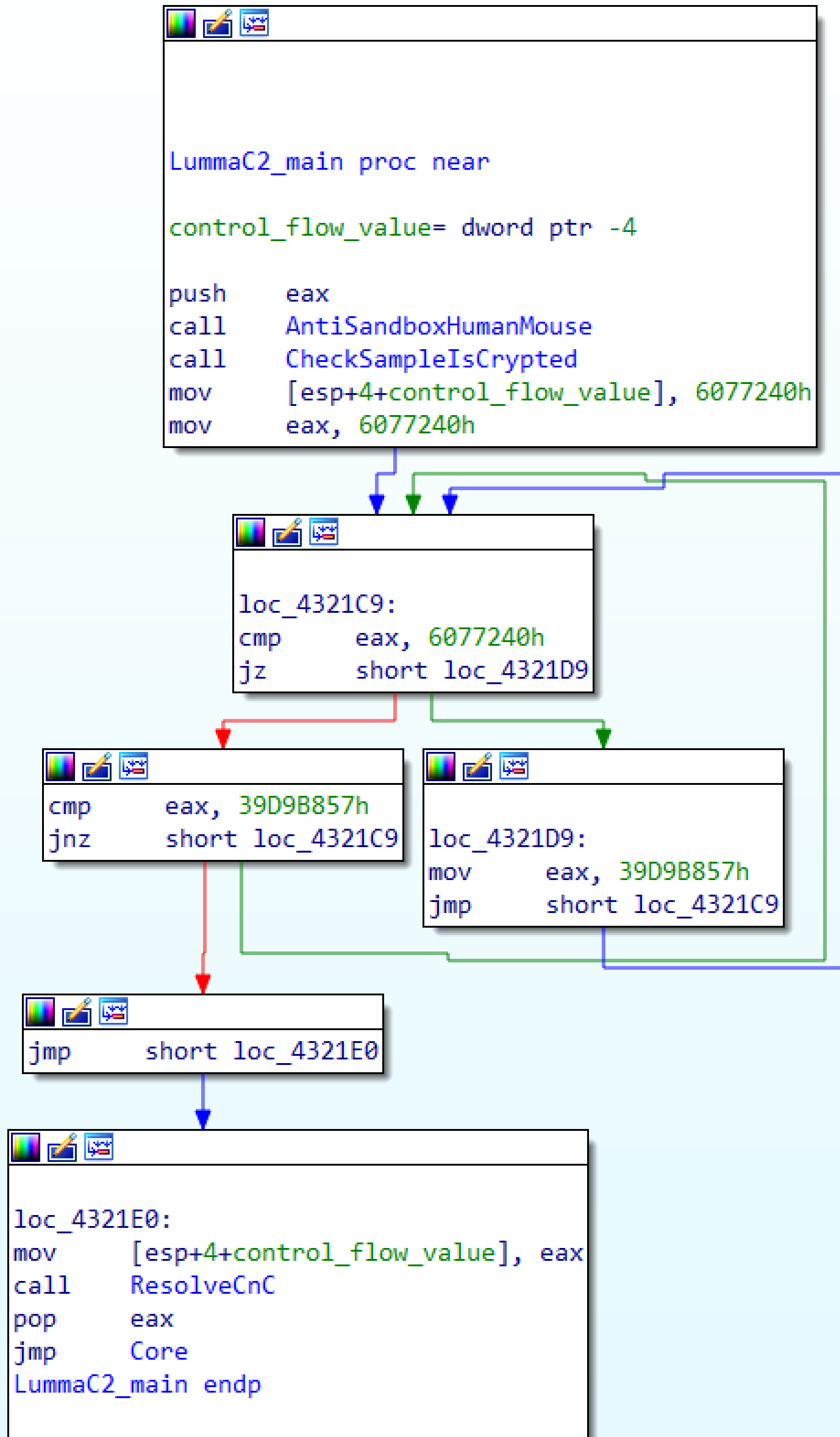

Figure 9. LummaC2 v4.0 main routine

Control Flow Flattening

Control Flow Flattening is an obfuscation technique aimed at breaking the original flow of the program and complicating its analysis. Furthermore, it makes use of **opaque predicates** and **dead code** to complicate analysis and make identification of relevant blocks more difficult.

Opaque predicates are employed to introduce intricacy into the primary program logic. Typically, this is achieved by creating additional branches within the control flow, often through the utilization of conditional jumps. Even though these conditional jumps are present, the overall behavior of the program remains deterministic.

Dead code refers to parts of the malicious code that are inactive or unreachable during the execution of the malware. Dead code may include obsolete functions, unused variables, or entire blocks of instructions that do not contribute to the malware's intended functionality. In LummaC2 v4.0 some dead code blocks can be recognized by the use of calls to other identified routines of the analyzed malware with invalid parameters, for instance.

The following picture represents the control flow of a program before and after Control Flow Flattening has been applied. As can be seen, this obfuscation technique is used to break the flow of a program by **flattening** it.

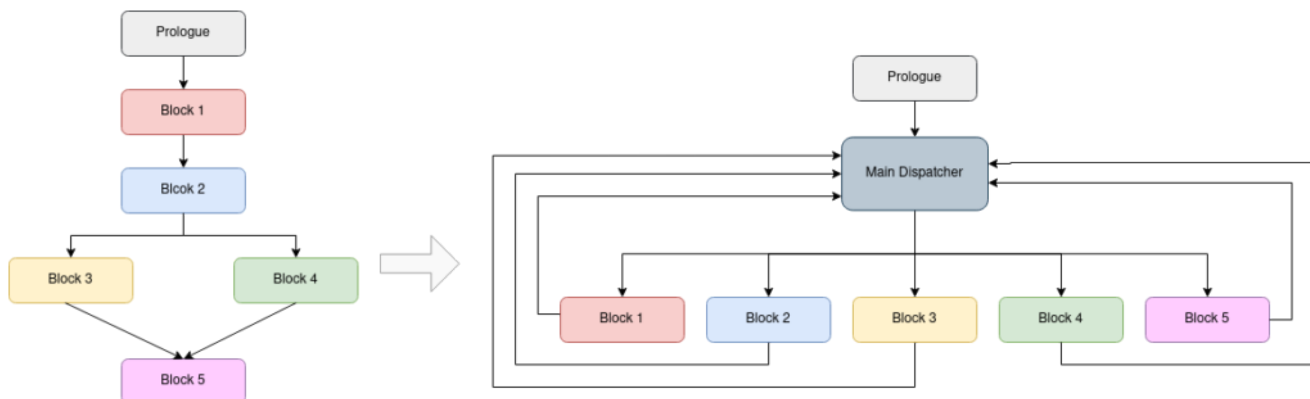

Figure 10. Example of program flow after applying Control Flow Flattening obfuscation

This detailed depiction of Qarkslab provides a more in-depth exploration of the distinct elements within a control flow graph that has been subjected to Control Flow Flattening obfuscation.

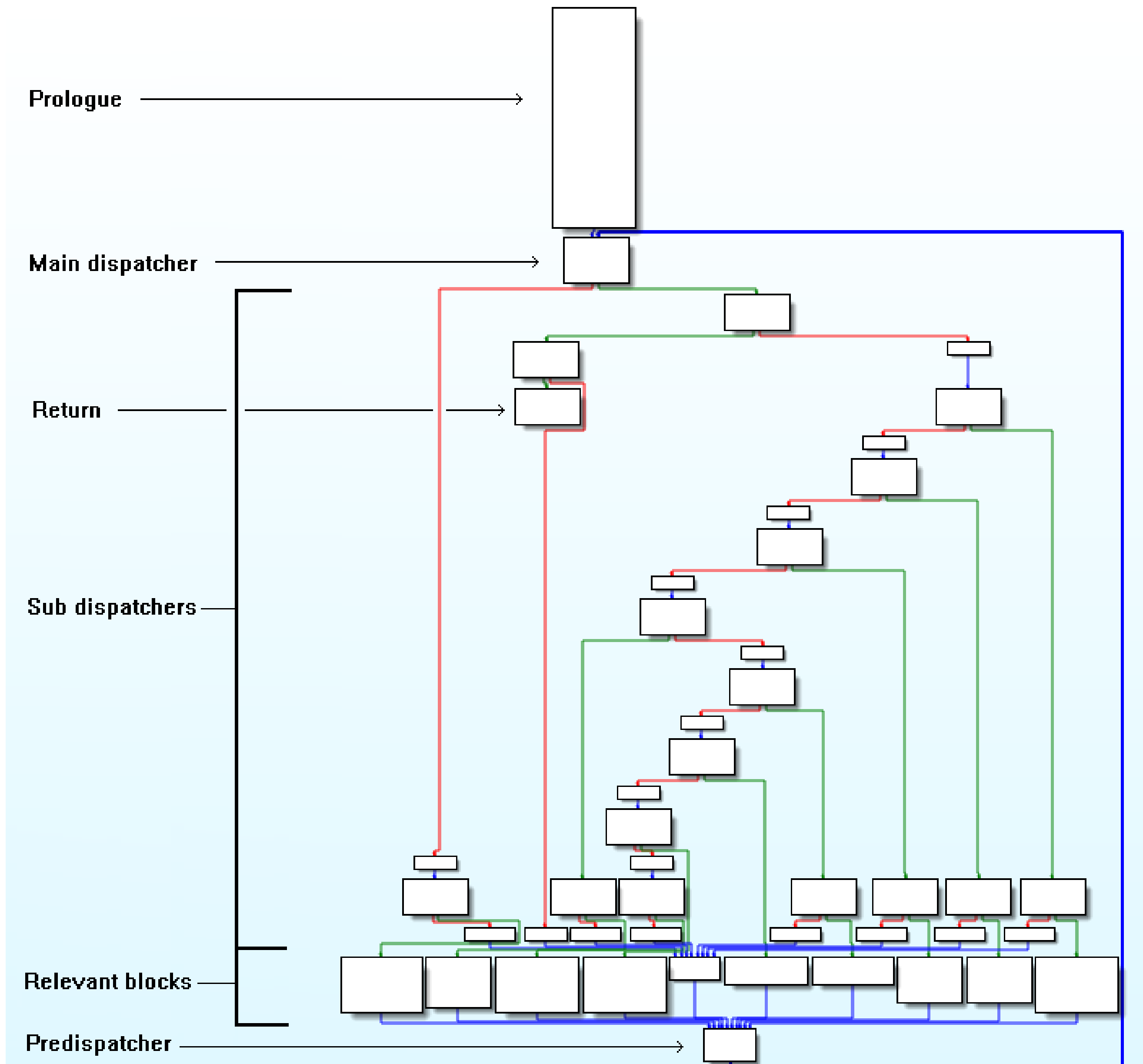

Figure 11. Schematic representation of the different elements in a control flow graph subjected to CFF

When dealing with this form of obfuscation, the initial step involves identifying the **main dispatcher**, the **relevant blocks**, and the **predispatcher**.

The **main dispatcher** represents the program block where execution always returns after completing one of its branches.

The **relevant blocks** encompass those blocks integral to the functional program, as evident in the initial representation, such as Block 1 and Block 2, etc.

The **predispatcher** holds the responsibility of altering the parameter's value, which the main dispatcher evaluates to determine the redirection of the execution flow toward a specific branch.

We will now delve into the initial routine executed by the unpacked LummaC2 v4.0, which incorporates an **anti-sandboxing** technique that we will examine in more detail shortly.

Control structure on stack

Trying to identify relevant blocks is not easy when CFF has been applied. We are going to analyze how this routine manages its control flow in order to make analysis easier. LummaC2 v4.0 will mostly store the value needed for the control flow dispatchers (sometimes more than one value is used to redirect the flow of the program) either in a local variable or at an offset to a memory location pointed to by a register.

The following picture shows the first instructions executed in the routine, (which is the responsible for executing anti-sandbox protection), the "prologue". As can be seen, an **extra stack space** of **0xD8 bytes** is reserved and assigned to **ESI** register. This register will be used throughout the entire routine accessing at offsets relative to it.

```
anonymous_33= dword ptr -5Ch
anonymous_34= dword ptr -58h
anonymous_35= qword ptr -54h
anonymous_36= dword ptr -38h

push    ebp
mov     ebp, esp
push    ebx
push    edi
push    esi
and     esp, 0FFFFFFF8h
sub     esp, 0D8h
mov     esi, esp
mov     dword ptr [esi], 7B829D7Eh
fld     ds:flt_46A3D0
fstp    qword ptr [esi+0A4h]
mov     ebx, ds:GetCursorPos
```

Figure 12. LummaC2 v4.0 assigning extra stack space to ESI. Other routines use alloca_aprobe instead

To ease analysis, we can create a **Structure** that will hold a maximum of the extra stack space allocated and fill its values while we are reverse engineering the sample.

```
; Attributes: bp-based frame fuzzy-sp

; int AntiSandboxHumanMouse()
AntiSandboxHumanMouse proc near

STACK= STACK ptr -0E4h

push    ebp
mov     ebp, esp
push    ebx
push    edi
push    esi
and     esp, 0FFFFFFF8h
sub     esp, 0D8h
mov     esi, esp
mov     [esi+STACK.control_flow_value], 7B829D7Eh
fld     ds:angle_threshold
fstp    qword ptr [esi+STACK.angle_threshold]
mov     ebx, ds:GetCursorangle_threshold dd 45.0
```

Figure 13. After applying structure type, we can see the angle of 45.0 is assigned to a member of the structure

```
    STACK.control_flow_value = 1546048066;
    STACK.hardcoded_status_value__ = STACK.hardcoded_status_value;
  }
  else if ( STACK.control_flow_value == -1109338672 )
  {
    calculate_difference_between_points(
      (POINTL *)STACK.ptr_1_coord,
      (POINTL *)STACK.ptr_2_plus_1_coord,
      (POINTL *)STACK.ptr_differences_n2_n1);
    STACK.control_flow_value = -1230812222;
  }
```

Figure 14. Naming structure fields while reverse engineering will help determine the routine's functionality

Relevant blocks

Unit42 researchers published an IDA Python script to help reverse engineering complex code. The following code is a slightly modified version of the one presented so that it runs in IDA Python 7.4+ and allows to execute in a running debugging session providing an *end_address*.

```
def init_heads(start_addr=None, end_addr=None):
    if not start_addr or not end_addr:
        start_addr = idc.get_segm_start(idc.get_screen_ea())
        end_addr = idc.get_segm_end(idc.get_screen_ea())

    heads = Heads(start_addr, end_addr)

    for i in heads:
        idc.set_color(i, CIC_ITEM, 0xFFFFFF)


def get_new_color(current_color):
    colors = [0xffe699, 0xffcc33, 0xe6ac00, 0xb38600]
    if current_color == 0xFFFFFF:
        return colors[0]
    if current_color in colors:
        pos = colors.index(current_color)
    if pos == len(colors)-1:
        return colors[pos]
    else:
        return colors[pos+1]
    return 0xFFFFFF # Default color


def main_debug(end_addr):
    if not end_addr:
        print("[!] Error, end_addr for main_debug not specified.")
        return

    idc.enable_tracing(TRACE_STEP, 1)
    event = ida_dbg.wait_for_next_event(WFNE_ANY|WFNE_CONT, -1) # Continue the
debugger, now configured for step tracing

    while True:
        event = ida_dbg.wait_for_next_event(WFNE_ANY, -1)
        addr = idc.get_event_ea()

        if end_addr:
            if addr == end_addr:
                print("[.] End address reached.")
                break

        current_color = idc.get_color(addr, CIC_ITEM)
        new_color = get_new_color(current_color)

        idc.set_color(addr, CIC_ITEM, new_color)

        if event <= 1:
            print("[.] Finished debugging.")
            break
```

```
    ida_dbg.suspend_process() # Wait for debugger to reach this point. PROGRAM
FINISHED after this


if __name__ == '__main__':
    start_addr = None
    end_addr = 0x011D21B8
    init_heads(start_addr, end_addr)    # Set white color (default)
    main_debug(end_addr)
```

With this script, we can (in a debugging session) select an *end_address* and let the debugger execute until this address is reached (or the debugging session is finished by other means). As the debugger runs, it will **color** the instructions it executes. The darker the instruction looks, the more times the debugger executed it.

As a result, we can easily identify those code blocks that got executed multiple times (they may be pre-dispatcher blocks), those that got executed only once and even detect **dead code** blocks that never get executed. This can significantly improve the reversing experience of the malware sample, especially if we can avoid looking at dead code.

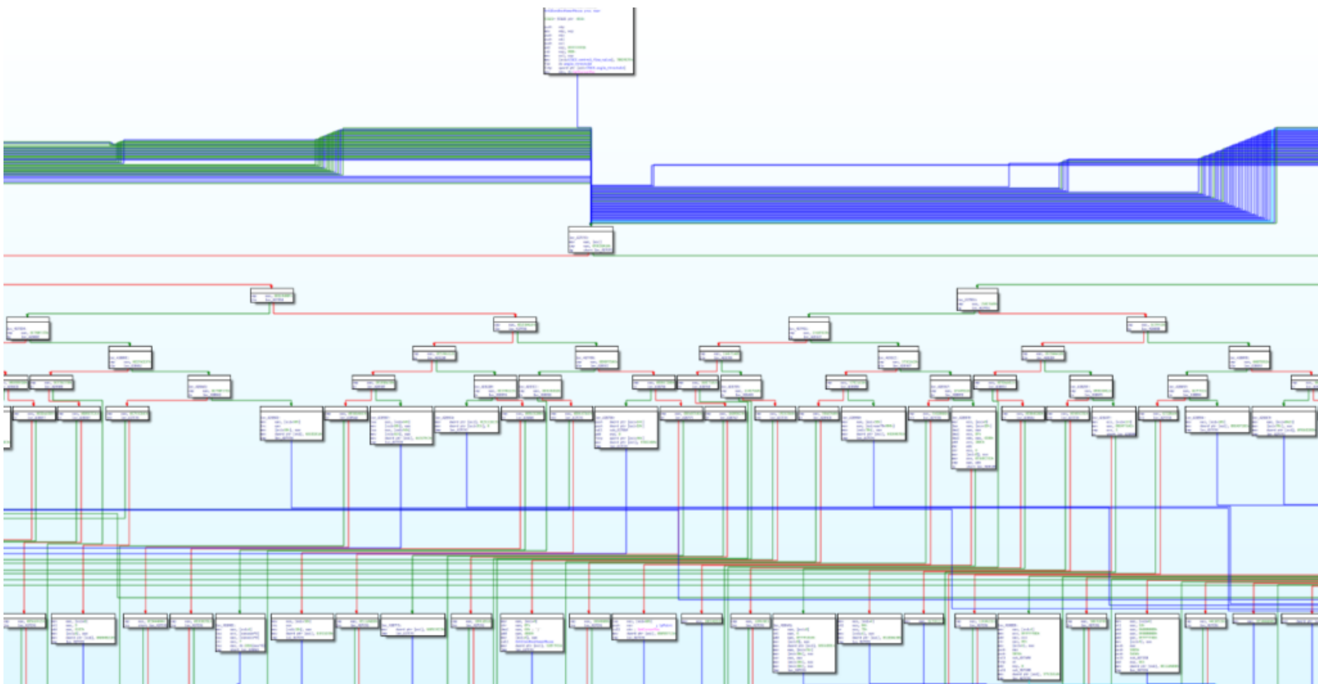

Figure 15. Routine with CFF obfuscation before running IDAPython script

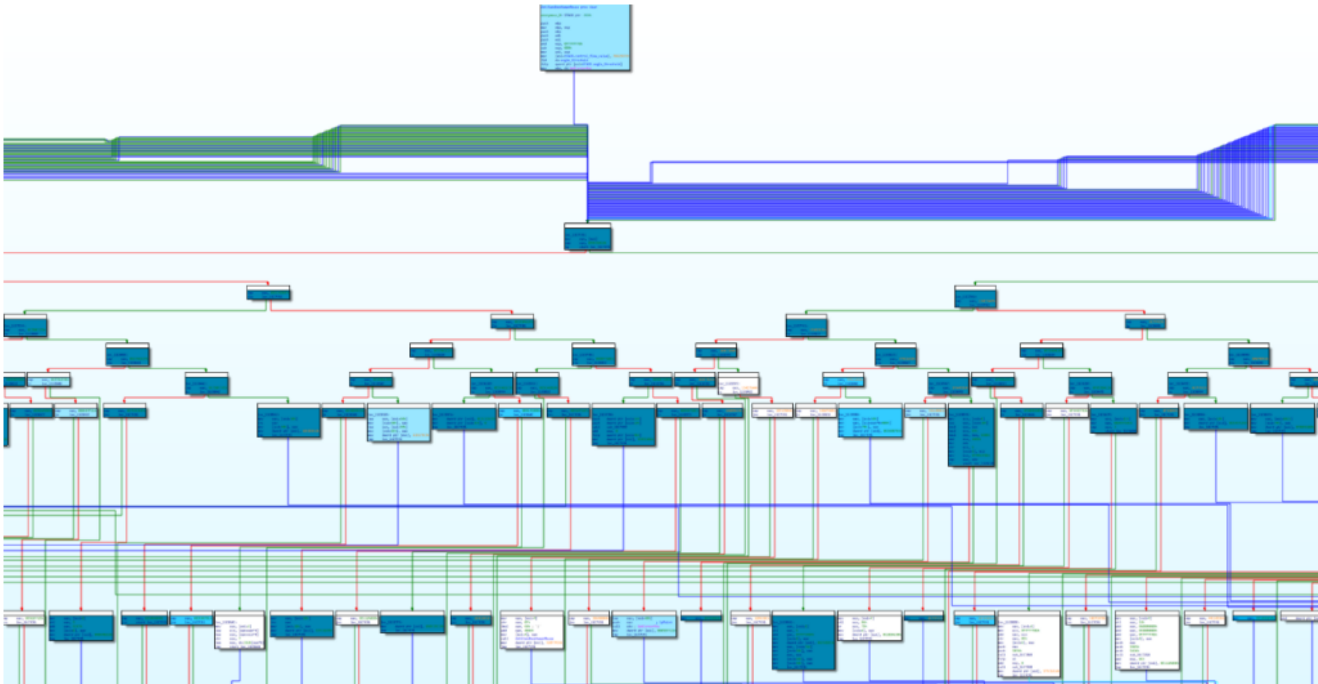

Figure 16. Routine with CFF obfuscation after running IDAPython script

New Anti-Sandbox technique: Using trigonometry to detect human behavior

LummaC2 v4.0 makes use of a **novel anti-sandbox** technique that forces the malware to wait until “human” behavior is detected in the infected machine. This technique takes into consideration different **positions of the cursor** in a short interval to detect human activity, **effectively preventing detonation** in most analysis systems that do not emulate mouse movements realistically.

The malware first starts by getting the **initial position** of the cursor with *GetCursorPos()* and waits in a loop for **300** milliseconds until a new capture of the cursor position is different from the initial one.

Once it has detected some mouse movement (as the cursor position has changed) it will save the next **5 positions** of the cursor executing *GetCursorPos()* with a small *Sleep* of **50** milliseconds between them.

After these 5 cursor positions have been saved, (let’s name them for now on **P0, P1, …, P4**), the malware checks if every captured position is different from its preceding one. For instance, it checks if ((P0 != P1) && (P1!= P2) && (P2 != P3) && (P3 != P4)). If this condition is not met, LummaC2 v4.0 will start again capturing a new “initial position” with a *Sleep* of 300 milliseconds until mouse movement is found and save new 5 cursor positions. This process will repeat **forever** until all consecutive cursor positions **differ**.

At this point we know the malware ensures that detonation will not occur until mouse is moved “quickly”. (Sandbox solutions that emulate mouse with less frequency will never detonate the malware).

After checking that all 5 captured cursor positions meet the requirements, LummaC2 v4.0 uses **trigonometry** to detect “human” behavior. If it does not detect this human-like behavior, it will start the process all over again from the beginning. The following lines explain the logic the malware follows.

Let’s assume here we have our 5 captured cursor positions (with a gap of 50 milliseconds between them) named P0, P1, …, P4.

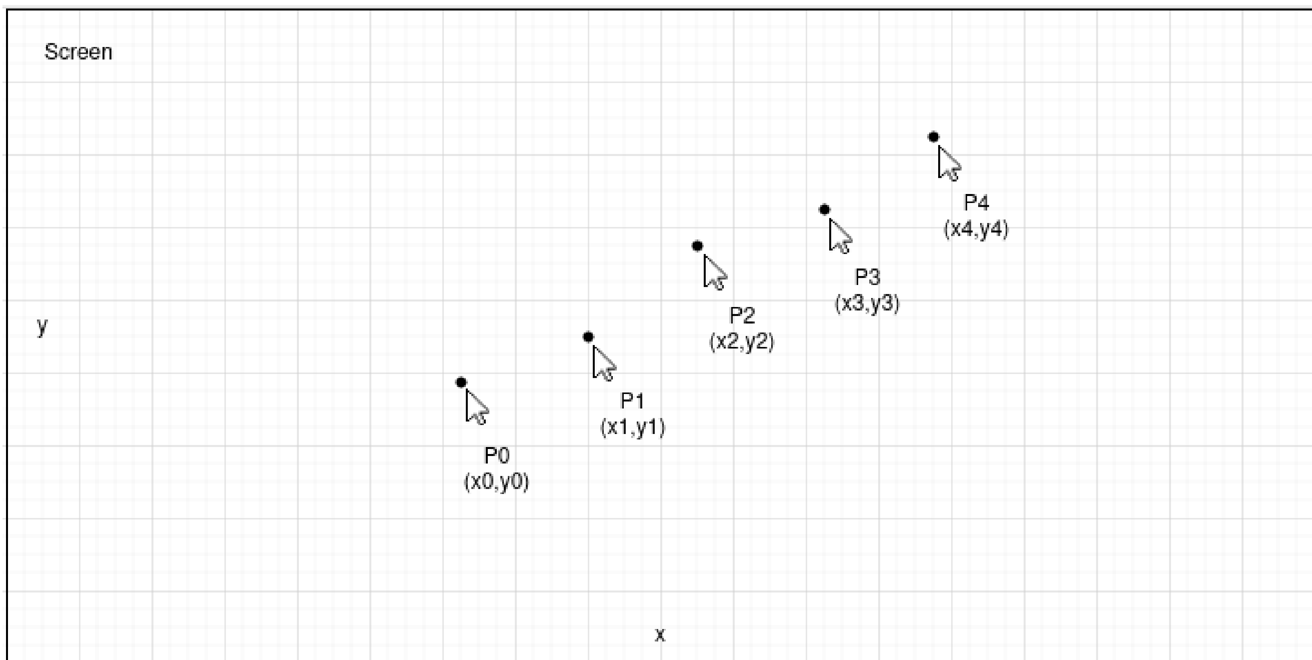

Figure 17. Representation of 5 captured mouse positions with a Sleep(50) between them

LummaC2 is going to treat these coordinates points as Euclidean vectors. As a result, captured mouse positions form **4 different vectors**: P01, P12, P23, P34.

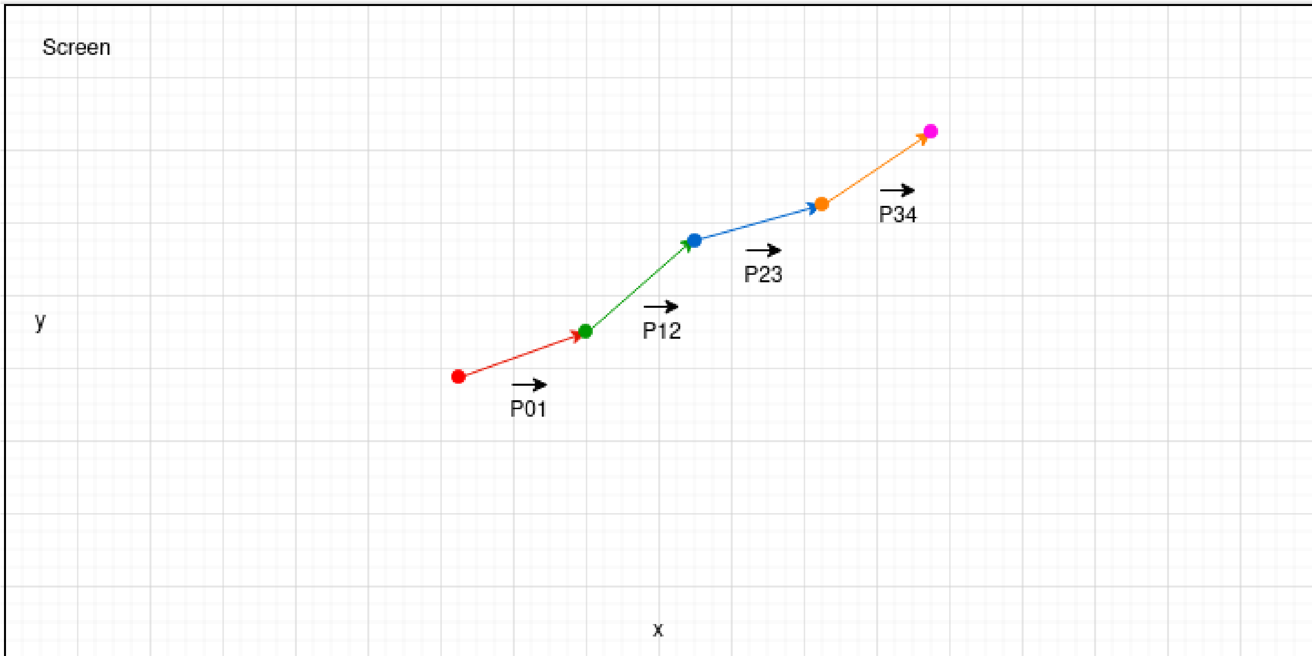

Figure 18. Representation of vectors formed by consecutive points from previous captured mouse positions

LummaC2 v4.0 then calculates the angle that is formed between consecutive vectors P01-P12, P12-P23 and P23-P34 sequentially. Here we can see an example of the different angles that are calculated:

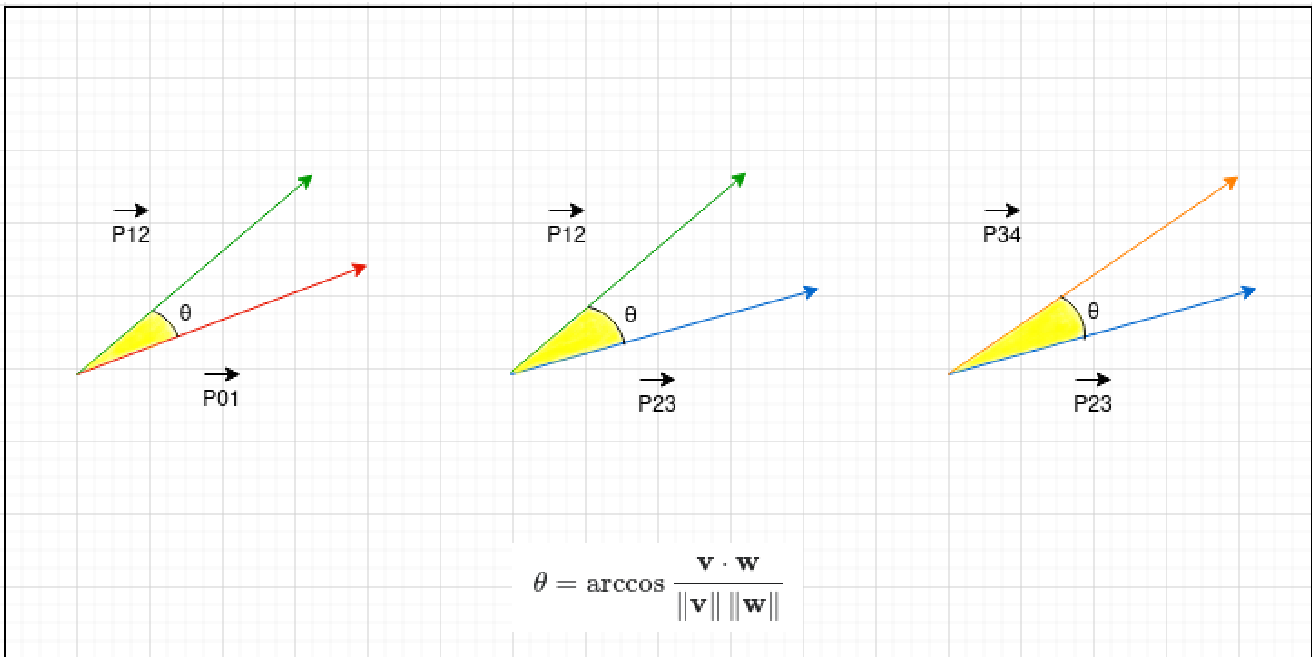

Figure 19. Representation of angles between previous vectors being calculated using dot product of vectors

To get the **magnitude** of a vector (shortest distance between two points), the **Euclidean distance** formula is used. To calculate the **angle** between two Euclidean vectors, it makes use of the **dot product of vectors** and transforms the result from radians to degrees. This can be seen in the following figures:

```
else if ( i == -1394638678 )
{
  control_value1 = (120 * control_value1) >> 26;
  i = -361003087;
}
else
{
  sum_square_dif_n1_n0_xy_ = (double)sum_square_dif_n1_n0_xy;
  sqrt_result2 = sqrt(sum_square_dif_n1_n0_xy_);
  if ( flag_ )
    sqrt_result2 = sqrt(sum_square_dif_n1_n0_xy_);
  sqrt_sum_square_dif_n1_n0_xy_distance = sqrt_result2;
  square_dif_n2_n1_x = ArgPtr_differences_n2_n1->x * ArgPtr_differences_n2_n1->x;
  i = 1951384782;
}
}
```


Figure 20. LummaC2 v4.0 uses Euclidean Distance formula to calculate the magnitude of vectors

```
if ( i > -379645203 )
{
  if ( i == -379645202 )
  {
    result_mult_difference_y = LongDifferences_n2_n1_y_ * LongDifference_n1_n0_y;
    i = 1022601060;
  }
  else
  {
    control_value1 = 1124073472 * control_value1 - 285212672;
    AntiSandboxHumanMouse();
    AntiSandboxHumanMouse();
    i = -428107481;
  }
}
else
{
  LongDifferences_n2_n1_y_ = *PtrDifferences_n2_n1_y;
  i = -379645202;
}
}
else if ( i <= 74927284 )
{
  result_dot_product_before_acos = result_sum_mult_dif_x_y// This is the Dot product between two vectors to determine the angle
                                 / (sqrt_sum_square_dif_n1_n0_xy_distance// vector magnitude p0-p1 (distance) and p1-p2 (distance)
                                  * sqrt_sum_square_dif_n2_n1_xy_distance);
  i = 1751782807;
}
else if ( i == 74927285 )
{
  PtrDifferences_n2_n1_y = &ArgPtr_differences_n2_n1->y;
  i = -428107481;
}
```


Figure 21. LummaC2 v4.0 uses dot product of vectors to calculate the angle between two vectors

The result is finally converted to **degrees** and compared against a threshold of **45.0º** degrees, which is **hardcoded** in the malware sample and initialized at the beginning of the anti-sandbox routine (as it can be seen in previous *Figure 13*).

```
if ( i == 1912906044 )
{
  control_value1 = (control_value1 << 6) - 21;
  calculate_difference_between_points((POINTL *)0x4879, (POINTL *)control_value1, (POINTL *)0x3AB);
  AntiSandboxHumanMouse();
  v3 = 1145292426;
  if ( control_value1 >= 0xC0 )
    v3 = 181264932;
  goto LABEL_71;
}
if ( i != 1951384782 )
  break;
control_value2 = 4684800 * control_value2 - 532599296;
}
return result_dot_product_radiants * 180.0 / 3.141592653589793;
```

Dead Code

Convert radians to degrees

Figure 22. The angle calculated between vectors is converted from radians to degrees

If all the calculated angles are lower than **45º**, then LummaC2 v4.0 considers it has detected "human" mouse behavior and continues with its execution. However, if **any** of the calculated angles is **bigger** than 45º, the malware will start the process all over again by ensuring there is mouse movement in a 300-millisecond period and capturing again **5 new cursor positions** to process.

In the following figure we can see how the anti-sandbox check is performed, once obtained the angle between two vectors in *angle_between_vectors_degress.* The *difference_threshold* is initialized at the beginning of the routine (which can be seen in *Figure 13*).

```
if ( STACK.control_flow_value == -1307007686 )
{
  STACK.field_4 = ((unsigned int)((STACK.field_4 << 7) - 46) >> 11) & 0xFFFFFFF8;
  v18 = STACK.field_4;
  K32Enum_ExitIfFewProcesses();
  v5 = 2072157566;
  if ( v18 >= 0x5E )
    v5 = 1490534512;
  goto LABEL_162;
}
if ( STACK.control_flow_value != 0xB3955F7B )
  return result;
HIBYTE(STACK.bool_threshold_FLAGGED) = *(double *)&STACK.angle_between_vectors_degrees > *(double *)&STACK.difference_threshold;
STACK.control_flow_value = 1115581340;
```

Dead Code

Return Block

Anti-Sandbox check

Figure 23. The angle calculated between vectors is compared against a hardcoded threshold of 45.0 degrees

How to bypass LummaC2 v4.0 Anti-Sandbox technique

As we have seen, this technique allows the malware to **delay detonation** until "human" mouse movement is detected. Moreover, this mouse movement must be continuous and smooth. Moving the cursor tortuously, quickly to random positions, circles or changing abruptly the direction of the cursor movement, etc., will most likely result in the malware **never** detonating.

In order to detonate LummaC2 v4.0, mouse movement must be continuous (5 different positions will be captured in approximately 0.25 seconds) and it must be a smooth movement (without changing directions abruptly), given that **angles** between the vectors

formed by these positions **cannot exceed 45 degrees**.

Forcing threat-actors to use a crypter for their builds

As we saw in earlier LummaC2 advertisements in underground forums, protecting the malware with a **crypter** was recommended to avoid leaking the malware anywhere in its pure form.

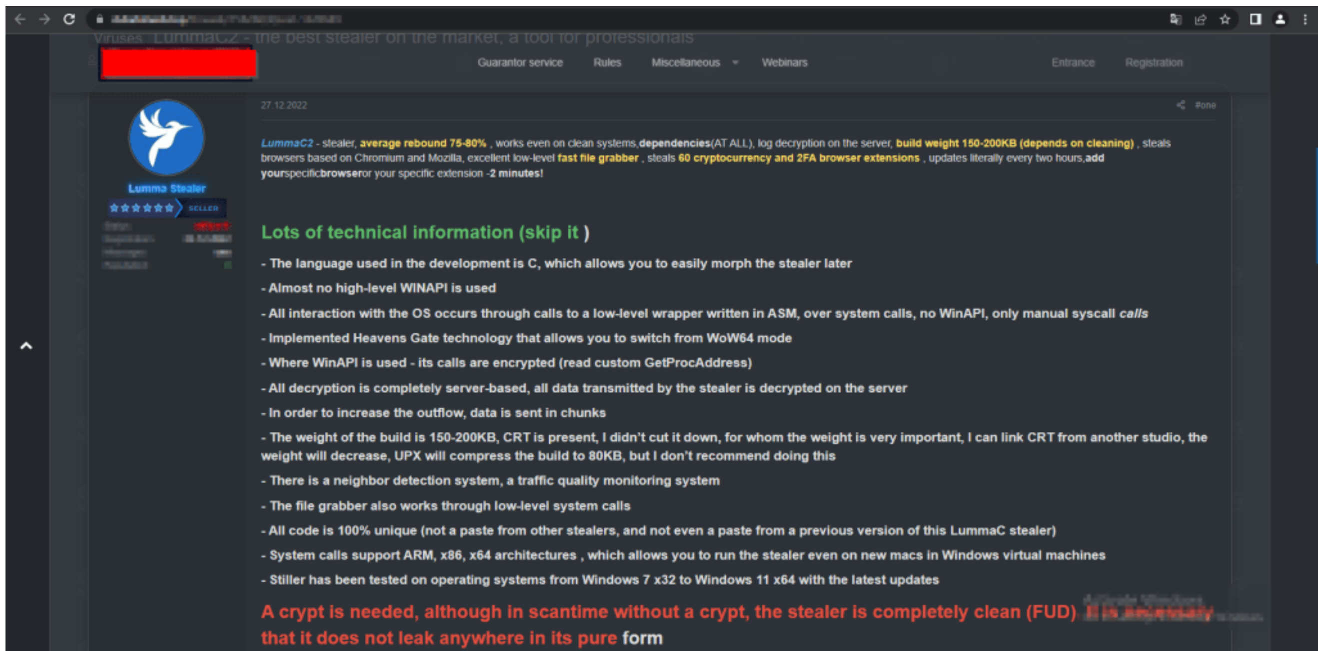

Figure 24. Dark Web Post for LummaC2 Stealer. Lines in red show threat actor encouraging to use a crypter

Newer versions of the malware added a new feature to avoid leaking the unpacked samples. In the case in which the sample being executed is not packed, the infected user will be presented with the following **alert** that allows them to prematurely finish the execution of the malware without **any harm** being caused to the infected machine.

This shows the lengths at which the malware developer is willing to go to avoid leaking unpacked samples.

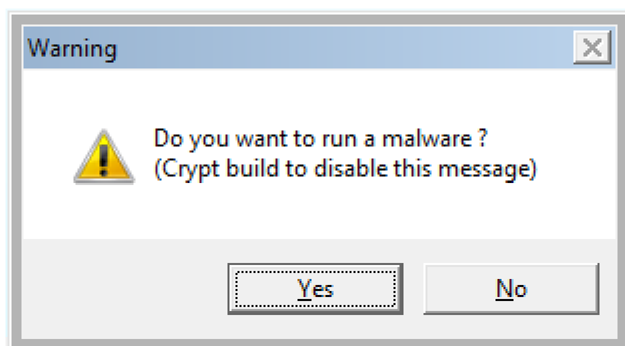

Figure 25. Alert message shown to the user when executing LummaC2 v4.0 in its unpacked form

LummaC2 v4.0 detects if the running executable is **crypted** by searching for a **specific value** at certain **offset** from the PE file. If this value is found, then the sample is not crypted. As we have previously seen, the packer sample does not create a new process to run the malware but uses a thread instead. Analyzing or executing the unpacked sample will result in this message box being displayed.

It will start by executing *GetModuleFileNameW* with *hModule* parameter 0 (to retrieve the path of the executable file of the current process) and obtaining a handle to the file for future reading via *CreateFileW* with *dwDesiredAccess* parameter set to *GENERIC_READ*. Then, it will get the size of the file with *GetFileSizeEx* and allocate a buffer of the obtained size using *malloc*. Following, a call to *ReadFile* is executed to fill the buffer with the contents of the executable file of the current process, as shown in the following Figure:

```
{
  **(_DWORD **)&STACK.bytes_read = 0;
  ReadFile(
    *(HANDLE *)&STACK.hFile,
    *(LPVOID *)&STACK.lpBufferRead,
    *(DWORD *)&STACK.bytes_to_read,
    *(LPDWORD *)&STACK.bytes_read,
    0);
  CloseHandle(*(HANDLE *)&STACK.hFile);
  STACK.control_flow_value = -302263345;
}
```

Figure 26. LummaC2 v4.0 reading the executable file of the current process to check if the malware is unpacked

Afterwards, it will check using *memcmp* if a **hardcoded mark** is present at a **hardcoded offset** plus 8 bytes. If the contents match, the user is presented with the alert preventing detonation via *NtRaiseHardError*.

```
while ( STACK.control_flow_value > 317470897 )
{
  if ( STACK.control_flow_value > 1077214315 )
  {
    if ( STACK.control_flow_value <= 1673607359 )
    {
      if ( STACK.control_flow_value <= 1140040903 )
      {
        if ( STACK.control_flow_value != 1077214316 )
        {
          IsCrypted = memcmp(
                        (const void *)(*(_DWORD *)&STACK.hardcoded_offset_flag_value
                                     + *(_DWORD *)&STACK.lpBufferRead
                                     + 8),
                        &hardcoded_mark,
                        20u);
          v17 = -2114303558;
          if ( IsCrypted )
            v17 = -1015987725;
          STACK.control_flow_value = v17;
          v1 = 1;
          goto LABEL_2;
        }
```

Figure 27. LummaC2 v4.0 checks if the malware is unpacked looking for a hardcoded value at a certain offset

This **hardcoded mark** (which changes between samples) has been previously **hashed** to check for file integrity. If this integrity check does not match, the malware ends abruptly. However, this should not be any issue if the unpacking of the malware was successful.

The following picture shows the hardcoded mark and the hardcoded offset. As well as the contents of the **unpacked version** of the malware, which would show the alert upon execution.

```
.data:014241E8 hardcoded_offset_flag_value dd 729E4h
.data:014241E8
.data:014241EC hardcoded_mark  db 0ABh
.data:014241ED                 db  9Eh
.data:014241EE                 db 0CFh
.data:014241EF                 db 0B1h
.data:014241F0                 db  19h
.data:014241F1                 db  38h ; 8
.data:014241F2                 db  88h
.data:014241F3                 db 0F9h
.data:014241F4                 db 0EDh
.data:014241F5                 db 0B5h
.data:014241F6                 db  56h ; V
.data:014241F7                 db  57h ; W
.data:014241F8                 db  5Dh ; ]
.data:014241F9                 db 0A0h
.data:014241FA                 db 0A3h
.data:014241FB                 db 0E6h
.data:014241FC                 db  66h ; f
.data:014241FD                 db  76h ; v
.data:014241FE                 db  5Fh ; _
.data:014241FF                 db    0
```

Figure 28. LummaC2 v4.0 hardcoded mark and offset used to check if the executing malware is unpacked

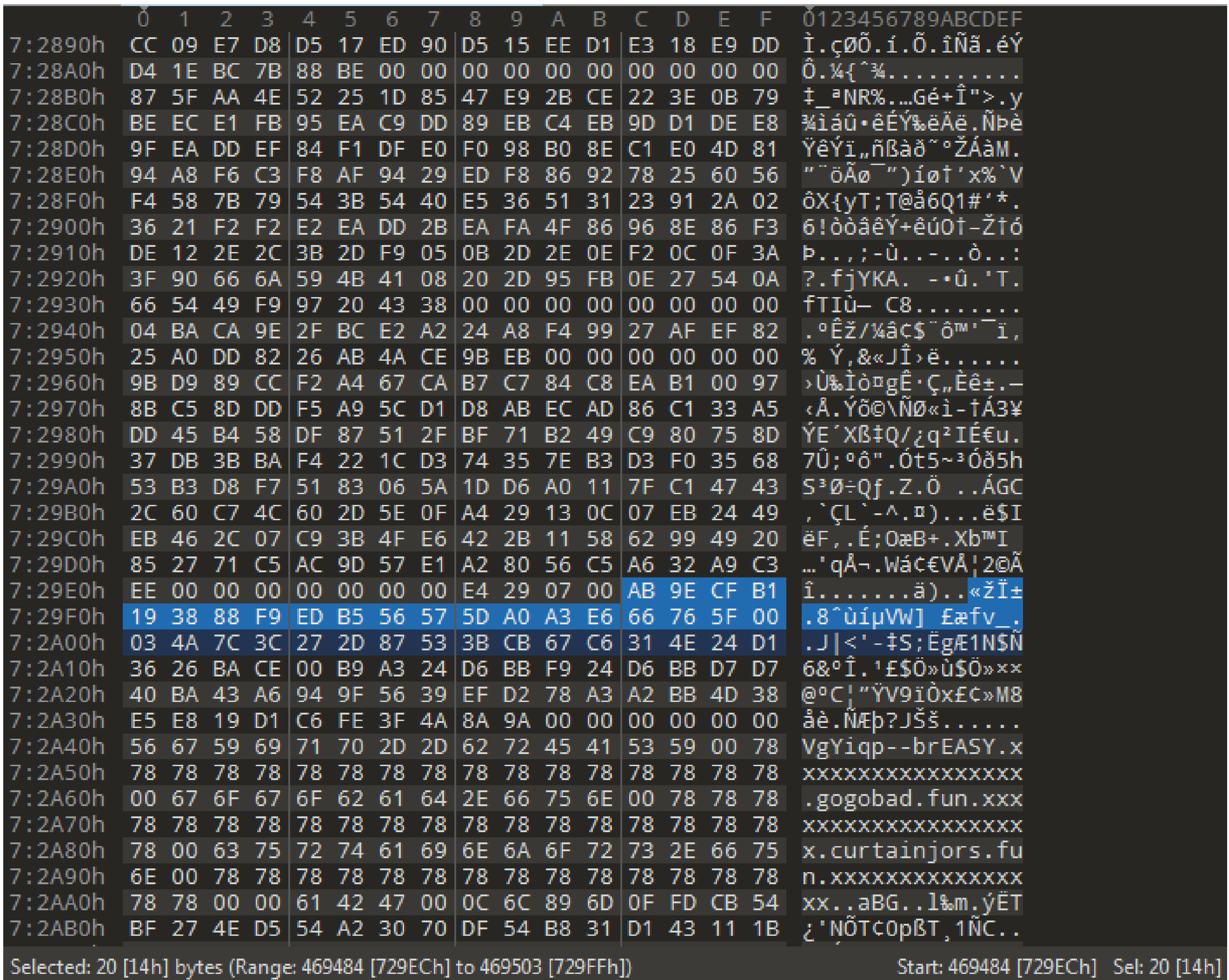

Figure 29. LummaC2 v4.0 unpacked sample contains the hardcoded_mark at hardcoded_offset+8

Conclusion

Information Stealers such as LummaC2 v4.0 pose significant risks and have the potential to inflict substantial harm on both individuals and organizations, including privacy breaches and the unauthorized exposure of confidential data. As our analysis has revealed, LummaC2 v4.0 appears to be a dynamic malware strain that remains under **active development**, constantly enhancing its codebase with additional features and **improved obfuscation techniques**, along with updates to its control panel. The ongoing usage of this malware in real-world scenarios indicates that it will likely continue to evolve, incorporating more advanced features and security measures in the future. This evolution is further facilitated by an open channel for customers to request bug fixes and propose enhancements.

Outpost24's KrakenLabs will continue to analyze new malware samples as part of our Threat Intelligence solution, which can retrieve compromised credentials in real-time to prevent unauthorized access to your systems.

IOCs

Hashes

LummaC2 v4.0 (sample 1)

- b14ddf64ace0b5f0d7452be28d07355c1c6865710dbed84938e2af48ccaa46cf (packed)
- 4408ce79e355f153fa43c05c582d4e264aec435cf5575574cb85dfe888366f86 (unpacked)

LummaC2 v4.0 (sample 2)

- de6c4c3ddb3a3ddbcbea9124f93429bf987dcd8192e0f1b4a826505429b74560 (packed)
- 976c8df8c33ec7b8c6b5944a5caca5631f1ec9d1d528b8a748fee6aae68814e3 (unpacked)

C&Cs

curtainjors[.]fun

gogobad[.]fun

superyupp[.]fun

References

“The Case of LummaC2 v4.0”, September, 2023. [Online]. Available: https://www.esentire.com/blog/the-case-of-lummac2-v4-0 [Accessed November 05, 2023]

“Everything you need to know about the LummaC2 stealer: leveraging IDA Python and Unicorn to deobfuscate Windows API hashing”, April, 2023. [Online]. Available: https://outpost24.com/blog/everything-you-need-to-know-lummac2-stealer/ [Accessed November 05, 2023]

“Using IDAPython to Make Your Life Easier: Part 4”, January, 2016. [Online]. Available: https://unit42.paloaltonetworks.com/using-idapython-to-make-your-life-easier-part-4 [Accessed November 05, 2023]

“A cryptor, a stealer and a banking trojan”, September, 2023. [Online]. Available: https://securelist.com/crimeware-report-asmcrypt-loader-lumma-stealer-zanubis-banker/110512/ [Accessed November 05, 2023]

“Deobfuscation: recovering an OLLVM-protected program”, December, 2014. [Online]. Available: https://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html [Accessed November 05, 2023]

“The Rise of the Lumma Info-Stealer”, September, 2023. [Online]. Available: https://darktrace.com/blog/the-rise-of-the-lumma-info-stealer [Accessed November 05, 2023]