Raspberry Robin Analysis | ThreatLabz

By Nikolaos Pantazopoulos

Published: 2024-11-19 · Archived: 2026-04-06 00:07:08 UTC

Technical Analysis

In the following sections, we present the features and functionalities of Raspberry Robin, including its obfuscation and anti-analysis methods.

Due to the amount of different components, the analysis is divided into four major sections.

1. Execution layers
2. Obfuscation methods
3. Decoy payload
4. Core layer

Execution layers

The core functionality of Raspberry Robin is unwrapped after executing a series of different layers. Each layer serves a different purpose with three of them (highlighted in bold) being the most important. In the table below, we briefly describe the functionality of each layer.

Execution Layer Name/Index	**Description**
First Layer	**Uses the segments registers GS/CS for code emulation detection and (XOR) decrypts the next layer.**
Second Layer	Decompresses (using a modified aPLib algorithm) and executes the next layer.
Third Layer	**Measures the performance of the CPU to verify if Raspberry Robin is executed in an analysis/sandbox environment and decrypts the next layer using the RC4 algorithm.**
Fourth Layer	Decrypts (via XOR) and executes the next layer.
Fifth Layer	Decompresses (with a modified aPLib algorithm) and executes the next layer.
Sixth Layer	**Runs a series of anti-analysis techniques and executes a decoy payload if an analysis environment is detected. Otherwise, it decrypts the next layer using the Rabbit stream cipher.**
Seventh Layer	Decrypts (using XOR) and executes the next layer.
Eighth Layer	Decompresses (using a modified aPLib algorithm) and executes the core layer.

Table 1: Raspberry Robin execution layers.

ANALYST NOTE: Even though this is not mentioned in the table above, recent samples of Raspberry Robin do a filename check in the initial executable file. Specifically, the binary compares its filename with the string `loaddll`*. We assess that the purpose of this check is to detect commercial sandboxes. Moreover, there are code checks to detect inline hooking of Windows APIs.*

The relationship among the different layers is shown in the figure below:

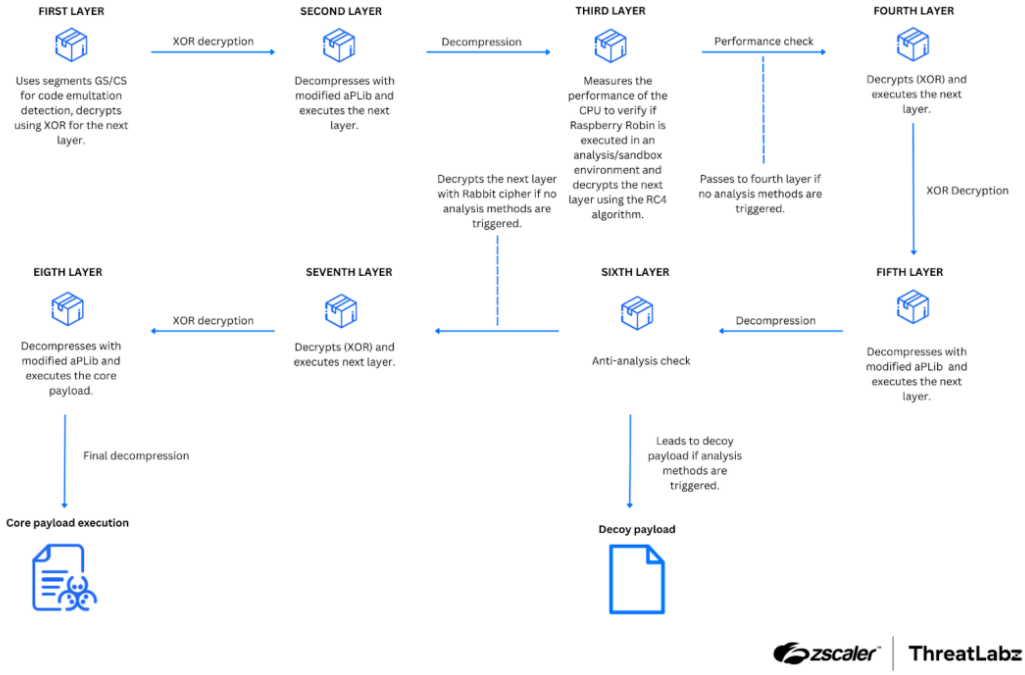

Figure 1: High-level diagram of the multi-layered architecture of Raspberry Robin.

First execution layer

The first layer performs the following anti-analysis actions to detect code emulators:

- Verifies that the code segment register (CS) holds the correct value (`0x23`). In the event of an incorrect value, the process crashes.
- Switches the value of the global segment register and calculates how many iterations were required to reset it to the default value (`0x2B`) due to context switches. Then, the first layer uses the calculated value to determine if the code has been emulated. This operation takes place multiple times (4,070 times in the samples we analyzed) and can be reproduced with the following Python code:

```
 counter = 0
for _ in range(4070):
        calculated_gs_value = switch_gs_value()
        seeder = ((-counter - 1 ) ^ 0x1f) & 0xff
val_1 = rol(counter, seeder, 32) & 0xffffffff
val_2 = rol(calculated_gs_value, seeder, 32) & 0xffffffff
rotation_bits = (-((-counter - 1) ^ 0x3F)) & 0xff
counter = rol( (val_2 | val_1) & 0xffffffff, rotation_bits, 32)
if counter == 0xFFFF:
        print("Detection passed")
```

ANALYST NOTE: Emulation detection with the segment register GS has been previously documented. If the last anti-analysis technique does not pass successfully, the code enters an infinite loop state since it passes an incorrect key value for the array table of the flattened control flow (which will be discussed later).

Third execution layer

The third layer uses an interesting trick, which is quite effective in detecting an analysis environment, even when set up on a physical machine. Specifically, Raspberry Robin (ab)uses a write-combining technique. The use cases for this method are limited and usually applied only in critical-performance operations (e.g. in graphics programming). Consequently, this technique affects the system's cache model in different ways:

- Prevents data caching.
- Reduces the CPU overhead since there are less write-operations.

Combining the information above with the fact that analysis environments (virtual or physical) have a limited amount of resources, Raspberry Robin uses this limitation to detect analysis environments, which either do not provide hardware support for this operation or their write/read operations are slow. The anti-analysis method can be reproduced with the following steps and the C++ code here.

- Allocates a large amount of virtual memory (size of 16,474,112 bytes) with the flags `PAGE_WRITECOMBINE / PAGE_READWRITE`.
- Once the memory has been allocated, a new thread starts to increment a global variable by one.
- The primary thread reads the global variable and starts writing a hardcoded-byte (different per sample) to the entire allocated memory area.
- After completing the operation, the completion time is calculated by reading again the (now increased) global variable and subtracting from it the old value.
- Repeats the same process with a read operation from the allocated memory.
- Lastly, the result from the write operation is divided with the result of the read operation. If the result is higher than or equal to the value 6 for six times, then the check is passed successfully. Otherwise, the code does not proceed to the next layer.

ANALYST NOTE: The expected minimum return value for the method above might vary from sample to sample. For example, recent samples have increased this variable.

If the anti-analysis method does not detect an unusual behavior (as described above), then the fourth layer is decrypted using the RC4 algorithm.

Sixth execution layer

The sixth execution layer includes the majority of the anti-analysis techniques. Moreover, if an anti-analysis method is not passed in this layer successfully, then Raspberry Robin deploys a decoy payload.

Interestingly, this layer does not execute its functionality (the anti-analysis methods) immediately. Instead, it replaces any stack addresses, that point to any address of the code segment of the initial executable file, with the address of the malicious function. This results in the execution of this layer when a return (*ret*) instruction executes any of the modified pointers.

To accomplish this, this layer follows the procedure described below.

- Starts by reading the name of the current file and compares its checksum with the checksum of two (currently unknown) filenames. If there is a match, then the `LDR_DATA_TABLE_ENTRY` field `ObsoleteLoadCount/LoadCount` is set to `-1` (to mark the current executable file as a static DLL load) along with the `Flags` field that is set to `0x20` (`ProcessStaticImport`). Also, a global variable, which is used later during the decryption of the next layer, is set to `true`.
- Attempts to fetch the base address and the code segment address range of an (unknown) loaded module.
- Retrieves the address range of the code segment of the initial executable file.
- Obtains the base of the stack by reading the fourth offset of the Thread Information Block (TIB).
- Starts iterating the stack and checks how many addresses (that point to the code segment) can be replaced with the malicious function address of this layer. The same check applies to the address range of the unknown module (if found).
- When the iteration has been completed, the algorithm chooses which module to target (initial file or unknown loaded module) by picking the one with the highest occurrences of addresses to replace. If the (unknown) module is chosen, then a global variable is set to `true`.
- The stack iteration starts again and replaces the candidate addresses with a pointer to the malicious function, as shown in the figure below.

Figure 2: Raspberry Robin return address patching.

Having completed the steps above, Raspberry Robin returns the execution to the initial executed file, which in return executes the malicious patched address.

Upon execution, the sixth layer starts running a series of anti-analysis techniques. A notable observation is the marking of any detected anti-analysis method using the reserved field of the Process Environment Block (PEB) at offset `0x30`. Specifically, this field is set to zero by default and Raspberry Robin updates it with a constant value, which indicates the anti-analysis method that was triggered (as shown in the table below). For example, detecting an unwanted process sets the field to the value `0x100000` plus the index of the detected process name in the list.

Base Value	**Anti-Analysis Method**
`0x20000`	• Checks for presence of a debugger by reading the PEB fields `BeingDebugged`, `NtGlobalFlag`, and `ForceFlags`. • Checks for the presence of a kernel-mode debugger by reading the field `KdDebuggerEnabled` of the `KUSER_SHARED_DATA` structure. • Checks if the number of active processors is less than two.
`0x30000`	• Obtains the value of field `NumberOfPhysicalPages` from the `KUSER_SHARED_DATA` to get the total RAM size and checks if the size is less than 800MB.
`0x70000`	• Checks the username of the compromised user against an embedded blocklist.
`0x80000`	• Checks the filename of the currently executed file against an embedded blocklist.
`0x90000`	• Collects the loaded modules and checks if any of them reside in an embedded blocklist.
`0xA0000`	• Checks if the CPU name is included in an embedded blocklist.
`0xB0000`	• Uses the assembly instruction `cpuid` to detect a virtualized environment. It is important to note that if the hypervisor leaf is hidden, the check proceeds and verifies that the machine is indeed not a hypervisor.
`0xC0000`	• Checks for the presence of files that are commonly present in sandbox/analysis environments.
`0xD0000`	• Checks the product ID of the current physical drive (`PhysicalDrive0`) of the hard disk.
`0xE0000`	• Checks if the monitor’s display name is included in an embedded blocklist.

Base Value	Anti-Analysis Method
`0xF0000`	• Checks the MAC address(es) of the host against an embedded blocklist.
`0x100000`	• Collects the process names and checks if any of them reside in an embedded blocklist.
`0x110000`	• Retrieves the firmware table using the Windows information class `SystemFirmwareTableInformation`, iterates the table, and checks if any of its values are present in an embedded blocklist. • Uses the Windows information class `SystemVhdBootInformation` and reads the structure member `OsDiskIsVhd` to verify if the disk is virtual.
`0x120000`	• Checks if the registry value `VBAWarnings` is set to `true`. This check applies to all versions of Microsoft Office.
`0x160000`	• Retrieves the sessions' named objects and verifies if these are included in an embedded blocklist. Names such as `HGFSMUTEX` or `HGFSMEMORY` are detected.

Table 2: Raspberry Robin anti-analysis techniques.

ANALYST NOTE: In addition to the methods above, Raspberry Robin uses other minor evasion techniques such as using random offsets in a memory-allocated area. Furthermore, certain anti-analysis methods, which were described by other researchers in the past, were not present in the recent samples we analyzed. For example, detection by using the `MulDiv` Windows API or mapping a large virtual memory blob with junk data were removed.

The same PEB field is updated before executing the next stage with the following operations.

- Bitwise OR operation with `0x20000000` if any of the two aforementioned unknown modules were detected during the stack modification process. This is set to `true` by default and does not appear to have any actual effect.
- Bitwise OR operation with `0x40000000` if no file path could be retrieved from the LDR table.
- Bitwise OR operation with `0x80000000` if the unknown module was targeted during the stack patching process.

Based on our analysis, the final value serves an important role in both the decoy payload and the final stage in the execution chain.

Lastly, if all anti-analysis checks have passed successfully, the current layer decrypts the seventh layer using the Rabbit stream cipher and executes it. Otherwise, the layer executes a decoy payload.

Decoy payload

The decoy payload is executed using a reflective loader and does not have any obfuscation methods applied (unlike all other layers). Despite this, all of the following conditions must be met in order to proceed:

- Verifies that User Account Control (UAC) is enabled by checking if the flag `DbgElevationEnabled` is set in the structure member `SharedDataFlags` of `KUSER_SHARED_DATA`. If UAC is not enabled, then the execution stops.
- Checks if the process's session ID is not zero.
- Reads the PEB field `SystemReserved`, which is modified from the previous layer, and determines whether the execution should proceed or not. As an example, assuming that only an anti-analysis method triggered the modification of this field, the decoy payload proceeds only if this value is `0xC0000` or higher.
- Checks if the compromised host is running for more than 13 minutes.

- Read the registry key `COMPUTERNAME` from `SOFTWARE\Microsoft\Mediaplayer`. If the key exists already, then execution stops. Otherwise, the decoy payload creates the aforementioned key and sets its value to zero. We assess that this is a method to mark the system as an analysis environment.
- Creates a UUID, calculates its CRC32 checksum, and compares the calculated checksum with the value `0x2AE5FD01`. In case of a match, the execution stops. It should be noted that it is highly unlikely to generate a UUID value that matches this checksum. We assess with low confidence that the developer(s) attempted to detect the presence of inline Windows API hooking.
- Reads the environmental variable `COMPUTERNAME` and stops if the returned value is included in an embedded list of CRC32 checksums (e.g., *JOHN-PC*, *USER-PC*, *LISA-PC*, *ANNA-PC* and *FRANK-PC*).

After passing the checks above, the decoy payload collects system information and sends the following data to a hardcoded domain.

- A generated UUID.
- The value of the `SystemReserved` field of the PEB that represents the anti-analysis method that was detected.
- A hard-coded integer value. e.g., `0xDC656D80`
- Running time of the compromised host (in minutes).
- Hostname and username.
- Executable's file path and the command-line.
- CPU name.
- Information about display monitors (device name, ID, monitor resolution, and refresh rate).

Once the information above has been collected, the decoy payload prepends the CRC32 value of the username, the `Cookie` value from `KUSER_SHARED_DATA`, and the system's timestamp.

If the length of this data is greater than 105 bytes, the code encrypts the data after this offset using the RC4 encryption algorithm (the encryption key is the first 12 bytes of the packet, which consists of the CRC32 of the username, `Cookie` value, and system's timestamp). The RC4 encrypted data is not used by Raspberry Robin, thus it may be an artifact from older code or an oversight by the malware author. The first 105 bytes of the data (including the CRC32 of the username, `Cookie` value, and system's timestamp) are encrypted using the RSA algorithm (with a hard-coded key) and encoded with Base64.

As a last step, the decoy payload replaces the characters `+`, `/`, `=` with `-`, `_`, `0` respectively and uses the output as a URI. Lastly, the expected response is a Windows executable file, which is saved to a filename that is the hex-encoded CPU name of the compromised host.

ANALYST NOTE: Considering the amount of information collected from the system along with the rest of the operational checks, we assess with medium confidence that this payload might (also) be used as a method for the threat actors to track unexpected compromised hosts (for example, sandbox environments).

Obfuscation methods

Raspberry Robin extensively uses a variety of different obfuscation methods. These are:

- Control flow flattening.
- Bogus control flow.
- Strings obfuscation - The decryption routine is unique per string.
- Mixed Boolean-Arithmetic Operations (MBA) - In many occasions, the open-source projects msynth and SiMBA can assist with the deobfuscation.
- Indirect calls combined with MBA obfuscation.
- Encryption and checksum algorithms.

The first five methods are applied during compilation at the intermediate representation (IR) level.

Obfuscated function key

Each obfuscated function includes an encrypted array table (unique per function). This array table is used during the entire execution of the function for any of the following reasons:

- Mapping the variables of the conditional statements of the flattened control-flow.

- Decrypting strings.
- Calculating the offset of indirect calls and as a result, their addresses.

An example is shown in the figure below.

```
v109 = a2;
vars0 = retaddr;
HIDWORD(v4) = global_variable_D3FC78C2;
LODWORD(v4) = global_variable_D3FC78C2;
v5 = *(a1 + 8);
v65 = *(a1 + 4);
v63 = v5;
v6 = (v4 >> 7) & 0x49FB0B89;
v7 = ~(v4 >> 7);
v8 = (-2 * (v4 >> 7) - 2) ^ 0x6C09E8EC;
v9 = (-2 * (v4 >> 7) - 2) | 0x6C09E8EC;
HIDWORD(v4) = v4 >> 7;
LODWORD(v4) = HIDWORD(v4);
v10 = -(v4 >> 19);
v11 = ((2 * (v9 - v8)) & (2 * ~(v6 + (v7 & 0xB604F476)))) - ((v6 + (v7 & 0xB604F476)) ^ (v9 - v8));
v12 = 1634806078 - (v4 >> 19) - ((v10 - 1) & 0x6171293F);
HIDWORD(v4) = (v12 ^ (v10 - 1)) - ((-2 * (v4 >> 19) - 2) & (2 * ~v12));
LODWORD(v4) = HIDWORD(v4);
v13 = (~(2 * v11) & (4 * (v4 >> 13))) + 2 * v11;
v14 = 2 * (v4 >> 13) - v11 + 2 * (~(2 * (v4 >> 13)) & v11);
HIDWORD(v4) = (v14 ^ v13) - 2 * (~v13 & v14);
LODWORD(v4) = HIDWORD(v4);
v71 = v4 >> 25;
v55 = &unk_7706FA19 ^ ((v71 ^ 0xA2) - 87);
v108 = (v71 ^ 0x468622A2) + 1798582953;
v104 = (v71 + 2060620908) ^ 0xD2F7A5EB;
v15 = 0;
v16 = v55;
for ( i = LOBYTE(function_encrypted_array_table[0]); v15 < i; v15 = (v15 + 1) )
{
  HIDWORD(v17) = (((v16 + 607150591) >> 18) ^ ((v16 << 14) + 411025408)) + 2050287481;
  LODWORD(v17) = HIDWORD(v17);
  v16 = v17 >> 23;
  *(array_table + v15) = v16 ^ function_encrypted_array_table[(v15 + 1)];
}
*(array_table + v15) = 0;
GetFileVersionInfoSizeW = 0;
state_variable = array_table[20];
while ( 1 )
{
  if ( state_variable - array_table[36] == array_table[6] )
  {
    v18 = 0;
    v66 = v104 ^ 0xB7E5ABF4;
    v19 = v66;
    do
    {
      // "%u.%u.%u.%u"
      v100[v18] = dword_28AA548[v18] + ((v19 + 1499948535) >> 6) + 125;
      v19 = (((v19 + 1499948535) >> 6) | ((v19 << 26) - 603979776)) - 399245187;
      v18 = (v18 + 1);
    }
    while ( v18 < 14 );
    GetFileVersionInfoSizeW = snprintf_s(
                                v63,
                                v65,
                                -1,
                                v100,
                                HIWORD(*(v67 + 16)),
                                *(v67 + 16),
                                HIWORD(*(v67 + 20)),
                                *(v67 + 20),
                                i,
                                v52) != -1;
    state_variable += array_table[29];
    goto LABEL_59;
  }
  if ( array_table[38] + state_variable == array_table[12] )
  {
    state_variable -= array_table[18] - array_table[49] * (1 - v68);
    goto LABEL_59;
  }
```

Figure 3: Raspberry Robin encrypted strings and control-flow flattening example.

However, the major issue is that in order to decrypt the array table, an integer key needs to be used. This key is passed into the function in one of the following ways:

- Passed directly from the caller function as a parameter.

- Uses the value of a global variable (as seen in the figure above) or a variable passed from a previous layer.
- The key is already part of the function's local variables.

The first two cases are related to each other because each key passed to a function has been derived either from the global variable of the current layer or the variable passed from the previous layer. For example, in the figure below, the global variable has its value calculated based on a value passed from the previous execution layer.

```
v73 = (((key_from_previous_layer - 77) ^ 0x99) + 111) ^ 0x48D84580;
v65 = ((key_from_previous_layer + 1974024883) ^ 0xF37E8799) - 1954907025;
v50 = 0;
v51 = v73;
do
{
  HIDWORD(v53) = v51;
  LODWORD(v53) = v51;
  v52 = v53 >> 18;
  v54 = *(v50 + 0x2F915BC);
  v51 = v52 + 0xF636E95;
  *(v66 + v50) = v54 + v52 - 107;
  v50 = (v50 + 1);
}
while ( v50 < 116 );
v55 = 0;
v69 = 0;
*(v66 + v50) = 0;
v56 = v66[27];
v71 = ((((((key_from_previous_layer ^ 0x4C9EEF2) - 182408596) ^ 0xEABB3451) - 1477305201) ^ 0xC2223DD0) + 1959582242) ^ 0xAC7D4E97)
    - 0x277FB9C3;
while ( 1 )
{
  if ( v56 - v66[16] == v66[25] )
  {
    v56 += v66[28] - v66[8] * (1 - v70);
  }
  else if ( v56 - v66[23] == v66[13] )
  {
    v68 = v54;
    v67 = v55;
    v57 = ((((key_from_previous_layer + (key_from_previous_layer & 0xBFC17C4B ^ 0xBFC17C4B)) ^ key_from_previous_layer)
          - 2
          * (key_from_previous_layer & ~(key_from_previous_layer + (key_from_previous_layer & 0xBFC17C4B ^ 0xBFC17C4B)))) | 0xBFC17C4B
        - ((((key_from_previous_layer + (key_from_previous_layer & 0xBFC17C4B ^ 0xBFC17C4B)) ^ key_from_previous_layer)
          - 2
          * (key_from_previous_layer & ~(key_from_previous_layer + (key_from_previous_layer & 0xBFC17C4B ^ 0xBFC17C4B)))) & 0xBFC17C4B
    v56 += v66[14];
    v58 = ((2 * key_from_previous_layer) & 2)
        + (key_from_previous_layer ^ 1)
        + 2 * (~(((2 * key_from_previous_layer) & 2) + (key_from_previous_layer ^ 1) - 1) & 0x3FC17C4B)
        + 1077838772;
    v59 = ((((2 * key_from_previous_layer) & 2) + (key_from_previous_layer ^ 1) - 1) & 0x403E83B4)
        - (((2 * key_from_previous_layer) & 2)
         + (key_from_previous_layer ^ 1));
    v60 = v59 + v58 - ((2 * v59) & (2 * v58));
    v61 = ((2 * v60) ^ (2 * v57)) + ((2 * v60) & (2 * v57));
    v62 = ((2 * ~v57) & (2 * v60)) + v57 - v60;
    v70 = a3 == 0;
    global_variable_958FE894 = ~(v61 * ~v62) + v62 * ~v61 + 1;
    v55 = v67;
  }
  else if ( (v66[17] ^ v56) == v66[5] )
  {
    v56 ^= v66[19];
  }
  else if ( v66[9] + v56 == v66[4] )
```


Figure 4: Raspberry Robin layer decryption using a global variable.

As a result, retrieving the initial/global variable of a layer is important since without it, it is not possible to conduct any analysis (with a few exceptions).

Even though it was rarely required, our approach to solve this problem was the following:

1. Reverse the decryption algorithm of the given function's array table.
2. Brute-force until there is a match in any conditional statement.

Control flow flattening

The implemented control flow flattening method, which Raspberry Robin uses, is encountered in all layers (including the final stage) and makes the analysis more tedious and time-consuming.

Each conditional statement is calculated by using a substitution, addition, or bitwise XOR operation on the state variable with an integer from the decrypted array table and then comparing the output with another value from the same array table.

Raspberry Robin takes care of boolean conditional cases by using a similar approach. In general, boolean comparisons are treated with one of the following methods:

- Multiply the boolean value directly with values from the decrypted array table and update the state variable. Example:

`state_variable -= array_table[21] - array_table[25] * boolean_value_output`

- Subtract the boolean value from the integer value1 and then update the state variable. Example:

`state_variable ^= array_table[9] + array_table[31] * (1 - boolean_value_output)`

As expected, the formulas above might vary from function to function (for example, the order might be different), but the core concept remains the same in all of them.

Lastly, considering all of the information above, encountering the obfuscated flow was a priority during our analysis and as a result we ended up creating our own IDA (decompiler) plugin.

Obfuscation algorithms

In addition to the previous obfuscation methods, both the execution layers and the core layer of Raspberry Robin contain different algorithms to either obfuscate or decrypt parts of the code. These are:

- Modified aPLib decompression algorithm (used only in the execution layers) - The developers have added an extra layer by adding the following formula in the `tag` member (in `aP_getbit` function) and on each read source byte:

`(input ^ 0xE9) - 0x66)`

- Custom XOR decryption loop (used only in the execution layers) - This can be replicated with the following Python code:

```
 data = data[8:]
key = int.from_bytes(data[:4], byteorder='little')
for idx in range(len(data)):
    enc_key = (0xb85ce6a1 + (key ^ 0x6ddfff7f & 0x5EB467BF)) & 0xffffffff
    dec_key = enc_key > 96) & 0xffffffff
    carry_field = enc_key >> (0x5f % 32)
    key =  (carry_field + dec_key) & 0xffffffff
    decrypted_byte = key ^ data[idx + 4]
    decrypted_byte &= 0xff
```

- Custom checksum algorithm (also available on GitHub) to replace plaintext strings comparisons. (Used across the entire execution process.)

ANALYST NOTE: Constant numbers in the algorithms above are different per sample. For example, the constant numbers 0xe9 and 0x66 (in the modified aPLib algorithm) are expected to be different.

Core layer

In this section, we describe the functionality of the core features for the final and main layer of Raspberry Robin.

Synchronization and code-execution behavior

The dynamic behavior of Raspberry Robin mostly depends on three factors:

- **File path parameter** - This parameter is passed to the final layer from the previous layers and contains the file path of the initial executable file. This parameter is used in different features of Raspberry Robin and as a result, the behavior might differ depending on its presence. Even though the majority of the available functionalities of Raspberry Robin require this parameter to point to valid data, there are checks to verify its presence in the unlikely event of not being available.
- **Semaphores and mutants** - As with other malware families, Raspberry Robin ensures that certain features do not run simultaneously by creating semaphores and mutants. The object name for each one of them is generated based on an input seed using the mulberry PRNG algorithm.
- **Modified PEB field** - The modified field of the PEB from the sixth layer appears to affect the behavior of the last layer. In this case, Raspberry Robin searches and obtains a handle to the process `msiexec` and hides its window (if visible). However, it is necessary to mention that we have not identified any value in the previous layers that can trigger this functionality.

Evasion and anti-analysis

Despite the amount of anti-analysis techniques used in previous layers, the final layer includes its own set of anti-analysis methods.

The implemented anti-analysis and evasion methods in the core layer are mentioned below.

- Hides any new thread by using the Windows class `ThreadHideFromDebugger`.
- Uses the assembly instruction `cpuid` to detect a virtualized environment.
- Uses the Windows classes `ProcessDebugFlags`, `ProcessDebugObjectHandle`, and `ProcessDebugPort` for debugger detection.
- Modifies the Windows API `DbgBreakPoint` to prevent a debugger from attaching to the process.
- Uses the Windows class `ObjectAllTypesInformation` with the Windows API `NtQueryObject` for debugger detection.
- Uses the WMI query `SELECT * FROM UWF_Filter` to verify if the Unified Write Filter (UWF) feature option is enabled.
- Even though not strictly an anti-analysis technique, the core layer compares the system's timestamp with a hardcoded one and exits if this date has passed. This is a constant check, which takes place before and after each network request.
- Monitors the system's activity. In case of a shutdown event, Raspberry Robin attempts to prevent this action by showing the message "*Adding features Don't turn off your computer shutdown*". This string is obtained from the legitimate DLL file `CbsMsg.dll`.
- Uses a `VMEXIT` assembly instruction (`cpuid`) to detect a virtualized environment. This method has been publicly documented already.
- Sets custom exception handlers and triggers debugging exceptions to detect debugger single-stepping and breakpoints (including hardware breakpoints). In any other case, the exception handler terminates the process except when the downloaded payload is running or the exception code is lower than `0xC0000000`.
- Detects process suspension by creating a thread with flags `THREAD_CREATE_FLAGS_SKIP_THREAD_ATTACH`, `THREAD_CREATE_FLAGS_HIDE_FROM_DEBUGGER`, and `THREAD_CREATE_FLAGS_BYPASS_PR(` The technique has already been described in the public domain.
- Usage of a direct Import Address Table (IAT) and as a result rebuilding a memory dump becomes a complicated task. Tools such as the Universal Import Fixer (UIF) can assist with solving this issue, but manual modifications are still required.
- When required, certain files/processes are added to the Microsoft Windows Defender exclusions list.
- Adds junk data into the injected code of a target process to evade detection.
- Attempts to evade detection when allocating memory space by writing at a random offset and not at the start of the allocated memory area.
- Removes the command-line of the current Raspberry Robin process by setting the PEB field `CommandLine` of `ProcessParameters` to zero.
- Disables Windows crash reporting.
- Removes the Image File Execution Options (IFEO) registry keys of specific processes (`regsvr32.exe`, `dllhost.exe`, `hh.exe`, `msiexec.exe`, `regasm.exe`, `explorer.exe`).

Obscure registry modification

In addition to the techniques above, Raspberry Robin uses an interesting approach to avoid detection while adding registry data.

Rather than modifying the Windows registry directly using common Windows API functions (e.g. `RegOpenKey`, `RegSetValueEx`), Raspberry Robin first renames the target registry key to a random one, writes the registry data into the renamed key, and renames it back to its original name.

However, if administrator privileges are available, Raspberry Robin uses a different approach. At first, it renames the registry key, creates an offline registry hive in the Windows temporary directory with a random filename. Then, it writes the registry data in the offline registry hive and loads the offline hive to the global registry tree using `ZwRestoreKey`.

Persistence

Raspberry Robin uses the Windows registry `Run` keys for adding persistence on the compromised host.

Upon execution, it adds the `ProgramData` directory in the Windows Defender's exclusion list and generates a random lowercase-alphabetic string ranging in size from 3 to 9 characters. The file's extension is randomly chosen from the list below.

- log
- tmp
- pol
- edb
- sdb
- jrs
- chk
- xml
- csv
- cmtx
- etl
- dit
- pat
- jdb
- dat

Then, the Raspberry Robin file is moved to a new directory using the previously generated string as a filename. Similarly, the directory created uses the same attributes of the `ProgramData` directory and has a randomly generated lowercase-alphabetic name ranging between 4 and 7 characters in length. Furthermore, the file moved has its alternate stream name `Zone.Identifier:$DATA` stripped and the `Modified` timestamp changed to the current system time.

Raspberry Robin uses the registry key `SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx` for adding persistence on the compromised host. The created key and value names are random lowercase-alphabetic strings ranging in size from 3 to 7 characters

It is crucial to highlight that Raspberry Robin checks if any of the following conditions are met:

- The process does not have administrator privileges.
- The process name `avp.exe` (part of Kaspersky's Antivirus product) is present on the compromised host.

If any of the above is true, Raspberry Robin changes the following aspects of its behavior.

- Uses the registry key RunOnce (under `HKEY_CURRENT_USER`), instead of `RunOnceEx`.
- The file location does not change and uses its current file path for persistence.

Lastly, depending on certain conditions, Raspberry Robin uses a different command-line (shown in the table below) for the persistence registry key.

Requirements	**Command-Line**
• The process does not have administrator privileges. • The process name `avp.exe` (part of Kaspersky's Antivirus product) is present on the compromised host. • A file path parameter has been passed. • The process is running under a SysWOW64 environment.	`RUNDLL32.EXE SHELL32.DLL,Control_RunDLL "filepath"`
• The process does not have administrator privileges. • The process name `avp.exe` (part of Kaspersky's Antivirus product) is present on the compromised host. • A file path parameter has been passed. • The process is not running under a SysWOW64 environment (runs on a 32-bit operating system).	`CONTROL.EXE "filepath"`

Requirements	Command-Line
• The process does not have administrator privileges. • The process name `avp.exe` (part of Kaspersky's Antivirus product) is present on the compromised host. • A file path parameter has not been passed. • The process is not running under a SysWOW64 environment (runs on a 32-bit operating system).	`RUNDLL32 shell32 ShellExec_RunDLL "filepath"`
• The process has administrator privileges. • The process name `avp.exe` (part of Kaspersky's Antivirus product) is not present on the compromised host. • A file path parameter has been passed. • If the process is running under a SysWOW64 environment.	`shell32.dll|Control_RunDLL|"filepath"`
• The process has administrator privileges. • The process name `avp.exe` (part of Kaspersky's Antivirus product) is not present on the compromised host. • A file path parameter has been passed. • The process is not running under a SysWOW64 environment (runs on a 32-bit operating system).	`SHELL32|ShellExec_RunDLL|regsvr32 /I "filepath"`

Table 3: Raspberry Robin persistence command-line.

Another interesting feature is the use of the Windows Encrypted File System (EFS) in case of having administrator privileges, but failing to add persistence on the compromised system. In this case, Raspberry Robin sets the value for the registry key `Software\Microsoft\Windows NT\CurrentVersion\EFS\CurrentKeys\NumBackupAttempts` to 3 and encrypts the Raspberry Robin binary file. Since this operation invokes the `efsui.exe` process, Raspberry Robin attempts (10 times in total) to terminate it.

Network propagation

One of the features of Raspberry Robin is to propagate itself and compromise Windows devices on the same network.

RDP propagation

At first, Raspberry Robin uses the Windows API `GetSystemMetrics` with parameters `SM_REMOTESESSION` and `SM_REMOTECONTROL` in order to determine if the compromised host is remotely controlled with an Remote Desktop Control (RDP) session. In the event of an unsuccessful return, Raspberry Robin retrieves the local session ID by reading the value of the registry key `GlassSessionId` and compares it with the session ID of the current process. If these two values are equal then the propagation process stops.

Next, Raspberry Robin constructs a file path using the prefix `\\tsclient\c` along with the current file path of Raspberry Robin and searches for target directories in the `users` directory. A target directory must meet certain criteria such as:

- Not be the *public* directory.
- Not be a hidden or an operating system directory.

For each directory that is discovered, the Raspberry Robin file is copied into the target directory with a randomly generated filename with the extension `.cpl` under the default Windows path `\start menu\programs\startup\` of the selected user.

If the operation above fails, Raspberry Robin verifies that the current user has administrator privileges and starts to iterate all network drives from `c` up to `z`. If any of them exist, it creates a symbolic path to `C:\\` and copies the file of Raspberry Robin using the same method as described earlier.

SMB propagation

Raspberry Robin uses two legitimate and publicly available tools for replicating itself, PsExec and PAExec, along with the Windows built-in utility IExpress.

Notably, PsExec and PAExec are downloaded from their official websites and are not embedded in Raspberry Robin. Before proceeding with their download and usage, Raspberry Robin ensures that the current user is part of the domain's administrator group. Otherwise, the execution does not proceed.

Furthermore, Raspberry Robin stores PsExec (or PAExec in case of a download failure) in the Windows temporary folder and adds the path to the Windows Defender exclusion list. Additionally, Raspberry Robin creates a Self Extraction Directive (SED) file (shown below) and saves it as a text file, with a randomly generated filename, in the Windows temporary directory. The placeholders of the SED file specify the following properties.

- **Placeholder %1** - The filename of the executable file that will be created. For this case, the file has a randomly generated name.
- **Placeholder %2** - Path of the file to execute after the package extraction phase. This is the filename of the Raspberry Robin file.
- **Placeholder %3** - Path to the Raspberry Robin file.

```
 [oFhqZaHHe]
%3=
[VerSiOn]
sEdVErsION=3
clAsS=IExpRess
[DpgHCcDmWR]
OfHqZAHhE=
[OptIONS]
sOURceFIlES=DPghCCdMWr
targEtNaMe="%1"
APpLAUNcHeD=rEGSVR32 -S "%2"
uSeLongFIlenaME=1
PoStINSTaLlCMD=
reBoOtmoDe=0
```

Raspberry Robin executes the formatted SED file with the IExpress command `iexpress.exe /n /q sed_filepath` to generate the payload. Lastly, it creates a list of internal IPs and enumerates all Windows devices in the domain. The information collected is then written in a text file and passed to PsExec/PAExec when executing them.

In general, Raspberry Robin uses the following commands to execute the generated payload on other Windows devices connected to the network:

- `filepath_to_psexec_paexec @network_info_list.txt -accepteula -c -d -s -r random_remote_service_name iexpress_payload_path` - *Executes the IExpress payload on all hosts included in the provided list file.*
- `filepath_to_psexec_paexec \\* -c -d -s -r random_remote_service_name iexpress_payload_path` - Executes the IExpress payload on all available hosts in the network domain.

Depending on the selected tool (PsExec or PAExec), the commands might differ in their parameters. For example, to avoid notifying the user, the parameter `-accepteula` is used for PAExec while, in the case of PsExec, Raspberry Robin modifies the registry key `EulaAccepted` to `true`.

It is noteworthy to mention that, in case administrator privileges are available, Raspberry Robin adds a list of additional exclusions and rules, as shown in the table below.

Exclusion Description	**Exclusion Rule**
Windows Defender Process exclusion	IExpress file path (`iexpress.exe`).

Windows Defender Process exclusion	Filepath of the downloaded PsExec/PAExec.
Windows firewall rule to enable ICMP and to ensure that PsExec/PExec will work properly.	`netsh.exe advfirewall firewall add rule name="ICMP Allow incoming V4 echo request" protocol=icmpv4:8,any dir=in action=allow`
Windows firewall rule to allow discovery and sharing between network hosts.	`netsh.exe advfirewall firewall set rule group="File and Printer Sharing" new enable=Yes`

Table 4: Raspberry Robin exclusion rules list.

All files that were created in the Windows temporary directory during the execution of the operations above, are removed.

Local privilege escalation

One of the interesting features of Raspberry Robin is that it goes to a great extent to elevate its local privileges by using UAC-bypass methods and local privilege escalation exploits.

UAC bypass methods

Raspberry Robin includes the following UAC-bypass methods, which are copied from public sources:

- UAC elevation by using the COM interface ICMLuaUtil (invoked on Windows 8.1 and prior versions only).
- UAC elevation by using the COM interface IElevatedFactoryServer.
- UAC elevation by using SSPI datagram contexts.
- UAC elevation by modifying the registry value at `shell\open\command` of certain classes (shown in the table below) at `HKEY_CURRENT_USER\Software\Classes`.
- Attempts to elevate its privileges by using the Windows API `ShellExecuteEx` with the verb `runas`. This creates a popup window, requesting from the user to accept an elevated operation.

The UAC-bypass targeted registry classes are shown in the table below:

Registry Key Class Name	**Executable Name**
`Launcher.SystemSettings`	`slui`
`Folder`	`sdclt`
`ms-settings`	`computerdefaults`
`mscfile`	`compmgmtlauncher`
`ms-settings`	`fodhelper`
`mscfile`	`eventvwr`

Table 5: Raspberry Robin UAC-bypass targeted registry classes.

As in other cases, Raspberry Robin performs a series of verifications and checks that affect the elevation method selection process:

- Checks if the flag `DbgElevationEnabled` is set in the member `SharedDataFlags` of the `KUSER_SHARED_DATA` structure. This flag is used to determine if UAC is enabled and, as a result, Raspberry Robin does not use any elevation method if disabled. The same rule applies if the process already has administrator privileges.
- Reads values of registry keys `ConsentPromptBehaviorUser/ConsentPromptBehaviorAdmin` to verify if the user's consent is required to perform an operation that requires elevation. In case of failure to read any of them, Raspberry Robin assumes that the user's consent is required.
- Retrieves the time of the user's last input event and checks if this event occurred within the last hour.
- Ensures that the process is not at a high integrity level.
- Checks for the presence of process `avp.exe` (Kaspersky Antivirus) and for the loaded DLLs `aswhook` (AVG Antivirus) and `atcuf32` (BitDefender Antivirus). Currently, the detection of any of these causes Raspberry Robin to skip the first UAC elevation method.

The malicious file is executed using the legitimate Windows application `control.exe` or `runlegacycplelevated.exe` (depending on the Windows version) along with the file path of Raspberry Robin with junk data appended to it. For example:

```
control.exe xgdtezukspcyil rximygvdjhgmhg "file_path"
```

ANALYST NOTE: There are indications in the binary's code that instead of using `control.exe/runlegacycplelevated.exe`, Raspberry Robin might use `rundll32` along with the parameter `advpack.dll RegisterOCX` followed by the target file path. The part of this code is currently unreachable based on our analysis.

Exploits

Historically, Raspberry Robin has used a different set of exploits. At the time of publishing this blog, these are CVE-2024-26229 and CVE-2021-31969. Their payloads are decrypted at runtime using RC4 and the choice for which exploit to use depends on the Windows version of the compromised host.

Raspberry Robin uses the Donut loader to load and execute its exploits. Unlike other components, neither the Donut loader nor the exploits have any obfuscation applied to them, most likely due to the fact that the binary's target architecture is x64. The code of the exploits is injected into the target process `cleanmgr.exe` with the parameter `/sageset` by using the KernelCallbackTable injection method. Because the injected payload is x64, Raspberry Robin uses the Windows API functions `NtWow64ReadVirtualMemory64`, `NtWow64WriteVirtualMemory64`, and `NtWow64QueryInformationProcess64`.

Network communication

The main objective of Raspberry Robin is to download and execute a payload on the compromised host. However, to reach this stage, there are several tasks that need to be completed first before requesting the payload.

The first step is to verify that the host can connect to the TOR network. This is achieved by decrypting a list of legitimate onion domains (version 3) and attempting to connect to one of them (randomly chosen). It is important to note that since the network communication is over the TOR network, Raspberry Robin has its own TOR client embedded in its code.

Next, Raspberry Robin creates three file mapping objects, encrypts the onion URL using RC4, and copies into one of them the following structure.

```
 struct mapped_data
{
  char flag; // Indicates if multiple attempts (up to 2) should be made to contact the specified onion URL and also affect
  char success; // Set to true if communication with the onion domain was successful.
  uint16_t  empty_padding;
  uint32_t host_external_ip_address; // Obtained after parsing and reading the response's HTTP header "X-Your-Address-Is"
  uint64_t elapsed_timestamp; // Obtained after sending a request and doing the operation server_time - system_time.
  size_t sizeof_payload_data; //  Used for both incoming and outgoing data.
  unsigned char *payload_data;
};
```

The member `host_external_ip_address` has an important role in the behavior of Raspberry Robin. This is because the TOR module does not proceed if the external TOR IP address belongs to any of the following subnets.

- 127.0.0.0/8
- 10.0.0.0/8
- 224.0.0.0/4
- 240.0.0.0/4
- 172.16.0.0/12
- 192.168.0.0/16

Moreover, the external IP address is RC4 encrypted and written in one of the remaining two memory mappings, which are not used at any other point during the execution. The RC4 key is 4 bytes long and is derived from the `Cookie` value of `KUSER_SHARED_DATA`.

The calculated elapsed timestamp is added to the local timestamp and the resulting output is compared with the hardcoded termination timestamp described earlier.

The RC4 key for the encryption of the packet data is derived from the mulberry PRNG algorithm with the `Cookie` value of `KUSER_SHARED_DATA` structure as a seed. Similarly, the names of the mapping objects are generated in the same way as the RC4 key and therefore the injected TOR module can access this information independently.

Next, Raspberry Robin proceeds with the code injection in one of the processes from the list below.

- `rundll32.exe`
- `dllhost.exe`
- `regsvr32.exe`
- `hh.exe`

The process created has its parent process identifier (PPID) replaced with the PID of `explorer.exe` and its command-line generated based on the name of the selected process. For example, the format of the command-line for the target process `hh.exe` is `process_filepath random_generated_string.chm`.

The code injection method that Raspberry Robin chooses depends on the presence of the module `aswhook.dll` (component of AVG Antivirus). If this module is present, then Raspberry Robin patches the entry point of the target process using a combination of Windows API functions `ZwProtectVirtualMemory`, `ZwWriteVirtualMemory`, and `NtResumeThread`. Otherwise, it uses the APC code injection technique with the Windows API function `NtQueueApcThreadEx2` (instead of `ZwQueueApcThreadEx`) if the Windows build version is higher than 19603. As a last step, a number of random bytes are added at the start of the mapped memory area of the TOR module loader.

ANALYST NOTE: It is common practice for Raspberry Robin to check for the build number of Windows to verify the support of new low-level APIs to avoid any detection alerts. For example, to use `NtMapViewOfSectionEx` instead of `NtMapViewOfSection`.

Another important observation is the use of Return-oriented Program (ROP) during the code injection process. Raspberry Robin collects the number of available address-candidates in the `NTDLL` module, chooses one at random, and uses it. An `NTDLL` address is selected only if the following conditions are met.

- The first byte of the address must be `0xc3` (`RET` assembly instruction).
- The previous two bytes must be smaller than `0x5f` and not equal to `0x5c` (`pop esp`) respectively.

In case of failure to find a candidate address, the entry point of the injected TOR module is used instead.

The injected module loader is similar to the previous execution layers. Therefore, the loader has the same decompression (modified aPLib) and decryption (RC4 and bitwise XOR-operations) algorithms, as well as the same anti-analysis methods of the core layer.

Following a successful TOR connectivity check, Raspberry Robin constructs a packet that includes the following information from the compromised host:

- External IP address of the compromised host.
- Embedded hardcoded string `AFF123` (this value has been the same across all samples).

- A 2-byte value derived from the CRC64-checksum value of the PE headers of the initially executed PE file, after modifying them.
- Volume serial number (8 bytes long, retrieved from the Windows structure `NTFS_VOLUME_DATA_BUFFER`).
- Creation timestamp of the directory `Documents` and `Settings` (in LDAP format).
- Number of active processors.
- Boolean value indicating whether the process is running under WOW64.
- Windows versioning information.
- Boolean values to indicate if the process has administrator privileges on a local and domain level.
- System time (in LDAP format) and locale information.
- Username, hostname, NETBIOS domain name, DNS domain name, and logon server name.
- Java version (if available).
- CPU name.
- Process's filepath (including the parent process, for example the path of `explorer.exe`). Special UTF-16-LE characters are encoded in their hex format representation. For example, the backslash character is encoded as `\x005C`.
- Product ID and serial number of the physical drive.
- MAC addresses.
- List of installed antivirus products (including the `ProductState` value, which represents the operational status).
- Display monitor information (e.g., device name).
- List of running process names.
- Desktop screenshot (in Base64 format) and width/height of the display monitor.

To avoid issues with a large data size, Raspberry Robin sets a maximum length of 131,072 bytes. This limit has a critical role when a desktop screenshot is taken. In case the screenshot's data size makes the total network packet size exceed the maximum limit, the screenshot is not included in the packet.

The 2-byte calculated value is of a particular interest. This value is calculated by reading the PE headers of the initial executable file (in the event that a filepath parameter is not included, this value is set to 5) and setting specific headers to zero (the size of the headers must not be over 256 bytes and the file must be larger than 65,536 bytes). The names of these PE headers are mentioned below:

- Security Directory RVA
- Timestamp
- Size of Security Directory
- CheckSum

Also, depending on the number of PE sections and the raw size/virtual-address of the first section, the header information (`IMAGE_SECTION_HEADER`) of the first section might be set to zero.

Then, Raspberry Robin calculates the CRC-64 checksum of the modified PE headers and bitwise-XORs the output with the lower part of the PE timestamp followed by a hard-coded constant value. If the value of the output is higher than 1,000, then Raspberry Robin changes the value to 4.

ANALYST NOTE: There are certain conditions, which can skip the process above and set a hardcoded value instead. For instance, the value is set to 1 if the session ID of the process is different than zero and the checksum of the value of the environmental variable `userdomain` does not match an (unknown) output.

Having collected all the information above, Raspberry Robin generates 8 random bytes and appends them to a hardcoded 8 byte value that acts as a salt and uses this value as an RC4 key to encrypt the data. The randomly generated part of the key is prepended to the packet after the encryption since the C2 server requires it. Furthermore, it generates random padding that is appended at the end of the RC4 encrypted data (the size of the random data is randomly chosen up to 2,048 bytes).

As the final step in the encryption process, a checksum value of the data is calculated (CRC-64) and appended. Raspberry Robin encrypts the packet produced using AES-CTR with a randomly generated initialization vector (IV) using the mulberry algorithm and encodes the encrypted output using Base64. The layout of the header for the final packet is provided below.

```
 #pragma pack(push, 1)
struct host_info_packet_header
{
```

```
  unsigned char aes_iv[16];
  uint64_t crc64_checksum;
  size_t sizeof_data;
  unsigned char* data;
};
```

The Base64 output is appended as a URI to an onion domain (version 3) and a request is sent using the TOR module (in the same way as described above).

ANALYST NOTE: It is important to point out that the CNCs of Raspberry Robin are not decrypted all at once. Instead, Raspberry Robin decrypts only a randomly-chosen one. One more interesting observation is the incorrect size of the onion domains, which are short by one character. This is an intentional choice by the developers. The character `d` *is added at the end of the onion domain by the TOR module.*

After sending the request, the C2 server of Raspberry Robin replies with a payload that is mapped directly into the current process and executed in a new thread.

However, before proceeding with its execution, Raspberry Robin parses the downloaded payload and prepends the following data:

- A hard-coded byte-array of 144 bytes with the first sixteen bytes being the sizes of the embedded members of the data.
- The formatted string `AFF123_file_checksum`, where `file_checksum` is the calculated checksum value based on the initial executed file (the third item in the host's information list above).

Moreover, Raspberry Robin creates a registry key (host-based name) at `SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall`, which indicates that the payload has been downloaded.
