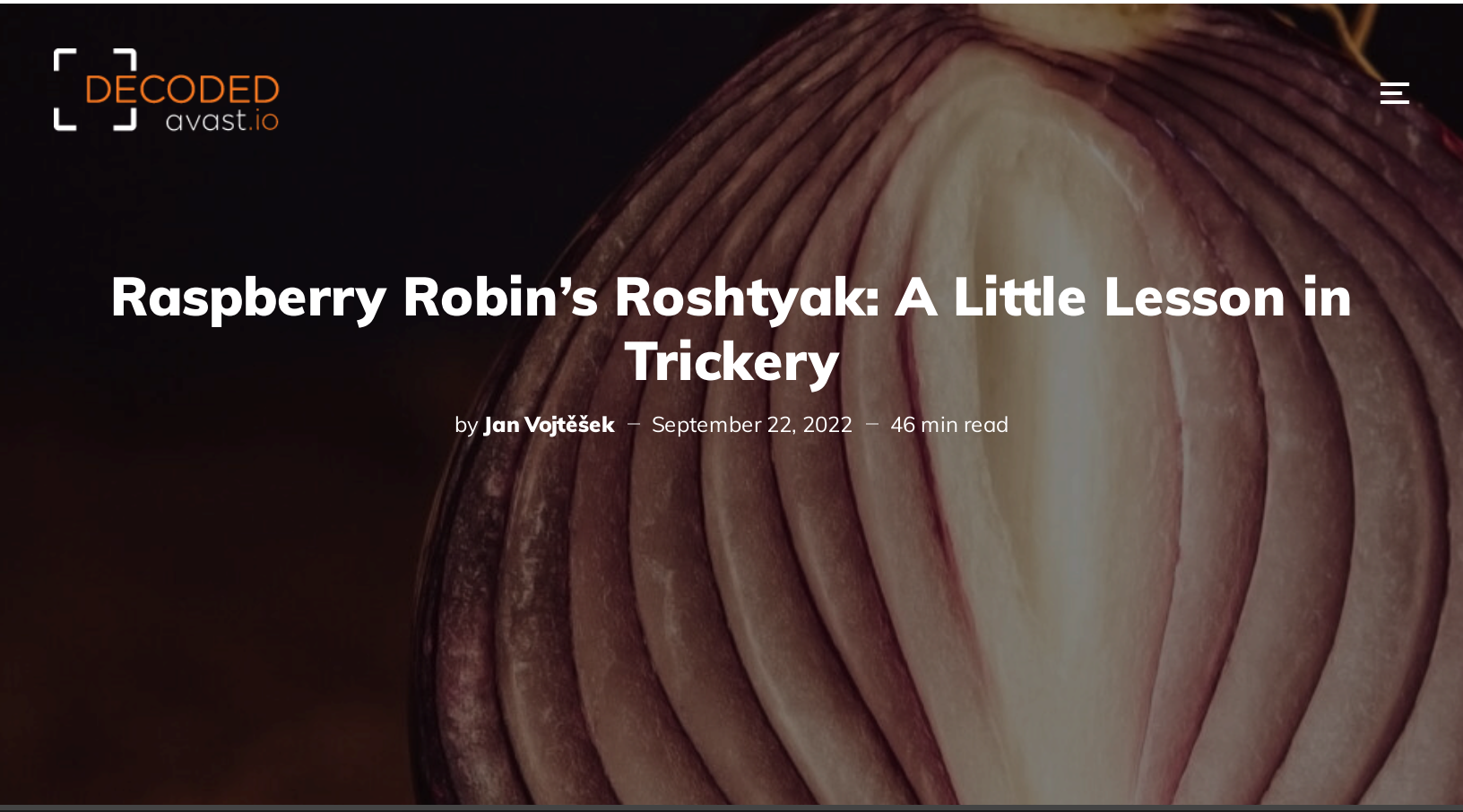


Raspberry Robin's Roshtyak: A Little Lesson in Trickery

by **Jan Vojtěšek** – September 22, 2022 – 46 min read

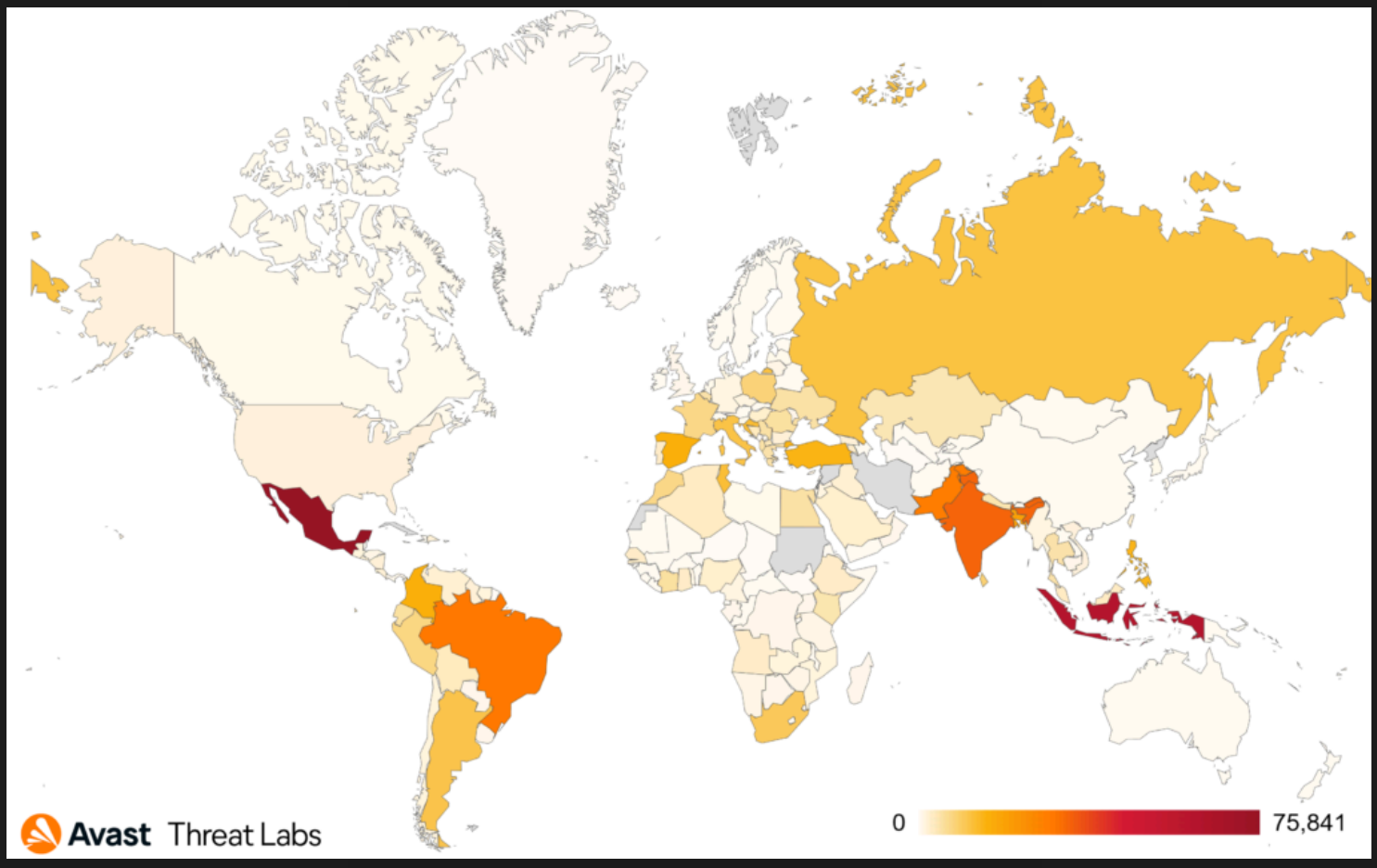
There are various tricks malware authors use to make malware analysts' jobs more difficult. These tricks include obfuscation techniques to complicate reverse engineering, anti-sandbox techniques to evade sandboxes, packing to bypass static detection, and more. Countless deceptive tricks used by various malware strains in-the-wild have been documented over the years. However, few of these tricks are implemented in a typical piece of malware, despite the many available tricks.

The subject of this blog post, a backdoor we dubbed Roshtyak, is not your typical piece of malware. Roshtyak is full of tricks. Some are well-known, and some we have never seen before. From a technical perspective, the lengths Roshtyak takes to protect itself are extremely interesting. Roshtyak belongs to one of the best-protected malware strains we have ever seen. We hope by publishing our research and analysis of the malware and its protection tricks we will help fellow researchers recognize and respond to similar tricks, and harden their analysis environments, making them more resistant to the evasion techniques described.

Roshtyak is the DLL backdoor used by Raspberry Robin, a worm spreading through infected removable drives. Raspberry Robin is extremely prevalent. We protected over 550K of our users

from the worm this year. Due to its high prevalence, it should be no surprise that we aren't the only ones taking note of Raspberry Robin.

Red Canary's researchers published the first analysis of Raspberry Robin in May 2022. In June, Symantec published a report describing a mining/clipboard hijacking operation, which reportedly made the cybercriminals at least $1.7 million. Symantec did not link the malicious operation to Raspberry Robin. Nevertheless, we assess with high confidence that what they analyzed was Raspberry Robin. This assessment is based on C&C overlaps, strong malware similarity, and coinfections observed in our telemetry. Cybereason, Microsoft, and Cisco published further reports in July/August 2022. Microsoft reported that Raspberry Robin infections led to DEV-0243 (a.k.a Evil Corp) pre-ransomware behavior. We could not confirm this connection using our telemetry. Still, we find it reasonable to believe that the miner payload is not the only way Raspberry Robin infections are being monetized. Other recent reports also hint at a possible connection between Raspberry Robin and Evil Corp.



A map showing the number of users Avast protected from Raspberry Robin

There are many unknowns about Raspberry Robin, despite so many published reports. What are the ultimate objectives behind the malware? Who is responsible for Raspberry Robin? How did it become so prevalent? Unfortunately, we do not have answers to all these questions. However, we can answer an important question we saw asked multiple times: What functionality is hidden inside the heavily obfuscated DLL (or Roshtyak as we call it)? To answer this question, we fully reverse engineered a Roshtyak sample, and present our analysis results in this blog post.

Overview

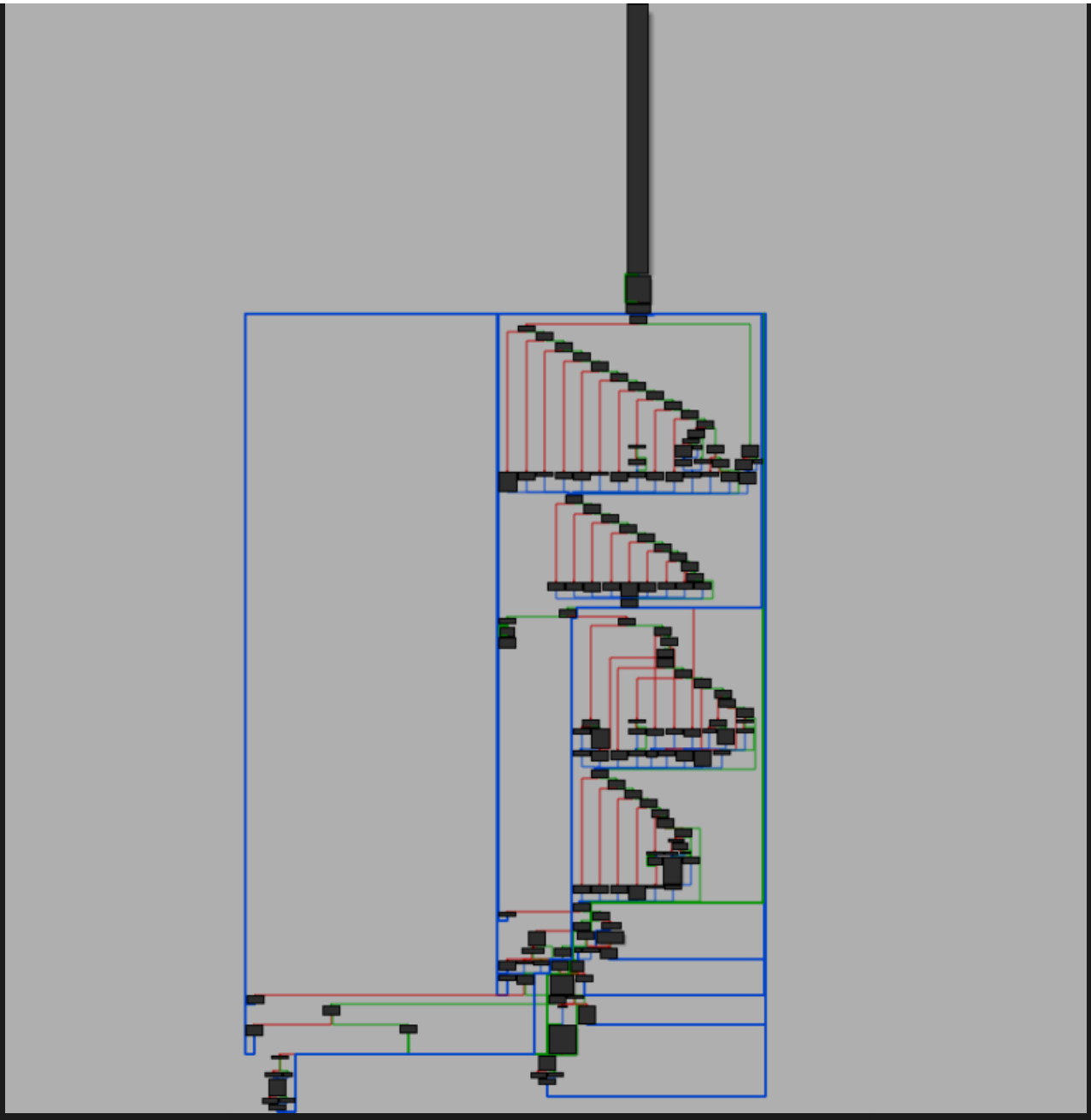
Roshtyak is packed in as many as 14 protective layers, each heavily obfuscated and serving a specific purpose. Some artifacts suggest the layers were originally PE files but were transformed into custom encrypted structures that only the previous layers know how to decrypt and load. Numerous anti-debugger, anti-sandbox, anti-VM, and anti-emulator checks are sprinkled throughout the layers. If one of these checks successfully detects an analysis environment, one of four actions are taken.

1. The malware calls `TerminateProcess` on itself to avoid exhibiting any further malicious behavior and to keep the subsequent layers encrypted.

2. Roshtyak crashes on purpose. This has the same effect as terminating itself, but it might not be immediately clear if the crash was intentional or because of a bug thanks to Roshtyak's obfuscated nature.

3. The malware enters an infinite loop on purpose. Since the loop itself is located in obfuscated code and spans thousands of instructions, it might be hard to determine if the loop is doing something useful or not.

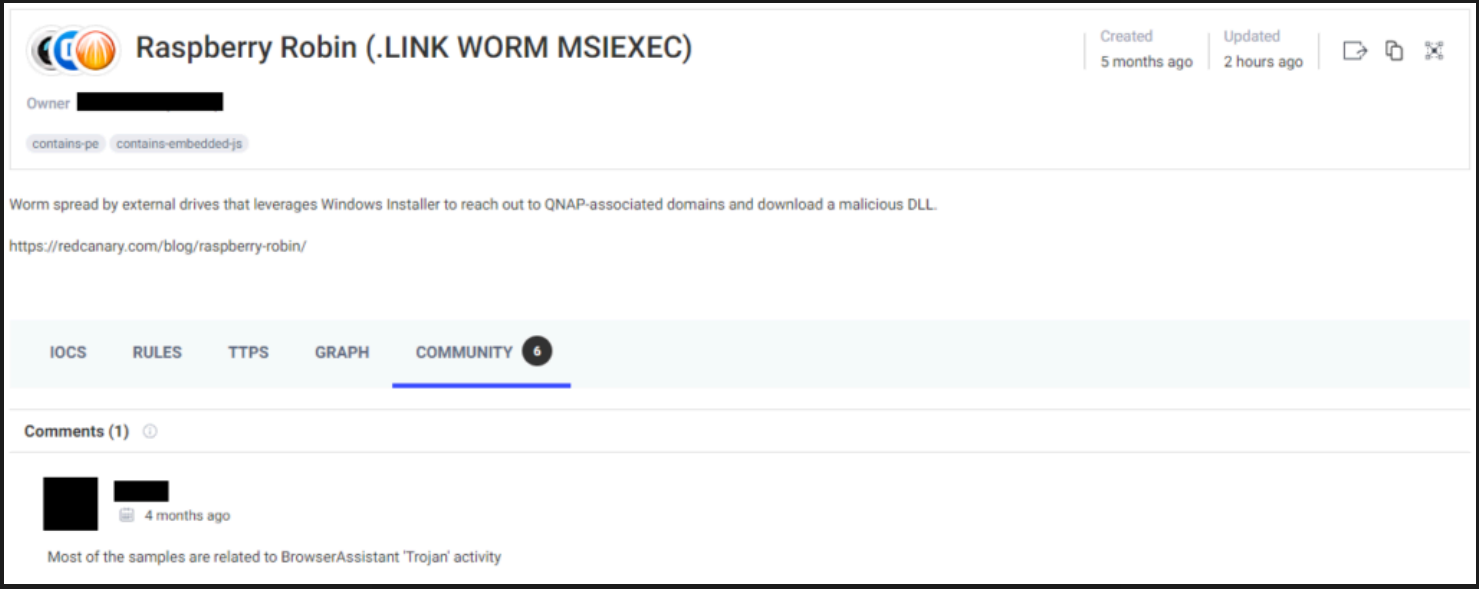
4. The most interesting case is when the malware reacts to a successful check by unpacking and loading a fake payload. This happens in the eighth layer, which is loaded with dozens of anti-analysis checks. The result of each of these checks is used to modify the value of a global variable. There are two payloads encrypted in the data section of the eighth layer: the real ninth layer and a fake payload. The real ninth layer will get decrypted only if the global variable matches the expected value after all the checks have been performed. If at least one check succeeded in detecting an analysis environment, the global variable's value will differ from the expected value, causing Roshtyak to unpack and execute the fake payload instead.



Roshtyak's obfuscation causes even relatively simple functions to grow into large proportions. This necessitates some custom deobfuscation tooling if one wants to reverse engineer it within a reasonable timeframe.

The fake payload is a BroAssist (a.k.a BrowserAssistant) adware sample. We believe this fake payload was intended to mislead malware analysts into thinking the sample is less interesting than it really is. When a reverse engineer focuses on quickly unpacking a sample, it might look like the whole sample is "just" an obfuscated piece of adware (and a very old one at that), which could cause the analyst to lose interest in digging deeper. And indeed, it turns out that these fake

payload shenanigans can be very effective. As can be seen on the screenshot below, it fooled at least one researcher, who misattributed the Raspberry Robin worm, because of the fake BrowserAssistant payload.



A security researcher misattributing Raspberry Robin because of the fake payload. This is not to pick on anyone, we just want to show how easy it is to make a mistake like this given Roshtyak's trickery and complexity.

The Bag of Tricks

For the sake of keeping this blog post (sort of) short and to the point, let's get straight into detailing some of the more interesting evasion techniques employed by Roshtyak.

Segment registers

Early in the execution, Roshtyak prefers to use checks that do not require calling any imported functions. If one of these checks is successful, the sample can quietly exit without generating any suspicious API calls. Below is an example where Roshtyak checks the behavior of the `gs` segment register. The check is designed to be stealthy and the surrounding garbage instructions make it easy to overlook.

```
lea     edx, [edx+eax]
sbb     edx, [esp+3ACh+var_3A0]
mov     edx, [esp+3ACh+var_3A8]
push    ecx
neg     cl
sets    ah
not     edx
setns   dl
mov     eax, edx
pop     gs
rcl     ecx, 1
sbb     eax, ecx
movzx   eax, al
neg     eax
inc     dh
neg     edx
not     ecx
rcr     eax, 0Dh
neg     ch
shl     eax, 13h
rcl     al, 5
movsx   ecx, ch
xchg    ch, dh
push    gs
inc     edx
xchg    ah, dl
imul    eax, edx
not     dl
lea     edx, [eax+edx*8]
pop     ecx
rcr     al, 7
```

A stealthy detection of single-stepping. Only the underscored instructions are useful.

The first idea behind this check is to detect single-stepping. Before the above snippet, the value of `cx` was initialized to `2`. After the `pop ecx` instruction, Roshtyak checks if `cx` is still equal to `2`. This would be the expected behavior because this value should propagate through the stack and the `gs` register under normal circumstances. However, a single step event would reset the value of the `gs` selector, which would result in a different value getting popped into `ecx` at the end.

But there is more to this check. As a side effect of the two push/pop pairs above, the value of `gs` is temporarily changed to `2`. After this check, Roshtyak enters a loop, counting the number of iterations until the value of `gs` is no longer `2`. The `gs` selector is also reset after a thread context switch, so the loop essentially counts the number of iterations until a context switch happens. Roshtyak repeats this procedure multiple times, averages out the result, and checks that it belongs to a sensible range for a bare metal execution environment. If the sample runs under a hypervisor

or in an emulator, the average number of iterations might fall outside of this range, which allows Roshtyak to detect undesirable execution environments.

Roshtyak also checks that the value of the `cs` segment register is either `0x1b` or `0x23`. Here, `0x1b` is the expected value when running on native x86 Windows, while `0x23` is the expected value under WoW64.

APC injection through a random ntdll gadget

Roshtyak performs some of its functionality from separate processes. For example, when it communicates with its C&C server, it spawns a new innocent-looking process like `regsvr32.exe`. Using shared sections, it injects its comms module into the address space of the new process. The injected module is executed via APC injection, using `NtQueueApcThreadEx`.

Interestingly, the `ApcRoutine` argument (which marks the target routine to be scheduled for execution) does not point to the entry point of the injected module. Instead, it points to a seemingly random address inside `ntdll`. Taking a closer look, we see this address was not chosen randomly but that Roshtyak scanned the code section of `ntdll` for `pop r32; ret` gadgets (excluding `pop esp`, because pivoting the stack would be undesirable) and picked one at random to use as the `ApcRoutine`.

```
ntdll.dll:7766150E __itoa_imported proc near
ntdll.dll:7766150E
ntdll.dll:7766150E arg_0= dword ptr  8
ntdll.dll:7766150E arg_4= dword ptr  0Ch
ntdll.dll:7766150E arg_8= dword ptr  10h
ntdll.dll:7766150E
ntdll.dll:7766150E ; FUNCTION CHUNK AT ntdll.dll:77692E9F SIZE 00000009 BYTES
ntdll.dll:7766150E
ntdll.dll:7766150E mov     edi, edi
ntdll.dll:77661510 push    ebp
ntdll.dll:77661511 mov     ebp, esp
ntdll.dll:77661513 cmp     [ebp+arg_8], 0Ah
ntdll.dll:77661517 mov     eax, [ebp+arg_0]
ntdll.dll:7766151A jnz     short loc_77661524
ntdll.dll:7766151C test    eax, eax
ntdll.dll:7766151E jl      loc_77692E9F
ntdll.dll:77661524
ntdll.dll:77661524 loc_77661524:                           ; CODE XREF: __itoa_imported+C↑j
ntdll.dll:77661524 push    0
ntdll.dll:77661526 push    [ebp+arg_8]
ntdll.dll:77661529
ntdll.dll:77661529 loc_77661529:                           ; CODE XREF: __itoa_imported+31995↓j
ntdll.dll:77661529 mov     ecx, [ebp+arg_4]
ntdll.dll:7766152C call    near ptr unk_776607A9
ntdll.dll:77661531 mov     eax, [ebp+arg_4]
ntdll.dll:77661534
ntdll.dll:77661534 loc_77661534:                           ; DATA XREF: Stack[000007A8]:00299B60↑o
ntdll.dll:77661534                                         ; Stack[000007A8]:00299F9C↑o
ntdll.dll:77661534 pop     ebp
ntdll.dll:77661535 retn
ntdll.dll:77661535 __itoa_imported endp ; sp-analysis failed
```

A random `pop r32; ret` gadget used as the entry point for APC injection

Looking at the calling convention for the `ApcRoutine` reveals what's going on. The `pop` instruction makes the stack pointer point to the `SystemArgument1` parameter of `NtQueueApcThreadEx` and so the `ret` instruction effectively jumps to wherever `SystemArgument1` is pointing. This means that by abusing this gadget, Roshtyak can treat `SystemArgument1` as the entry point for the purpose of APC injection. This obfuscates the control flow and makes the `NtQueueApcThreadEx` call look more legitimate. If someone hooks this function and inspects the `ApcRoutine` argument, the fact that it is pointing into the `ntdll` code section might be enough to convince them that the call is not malicious.

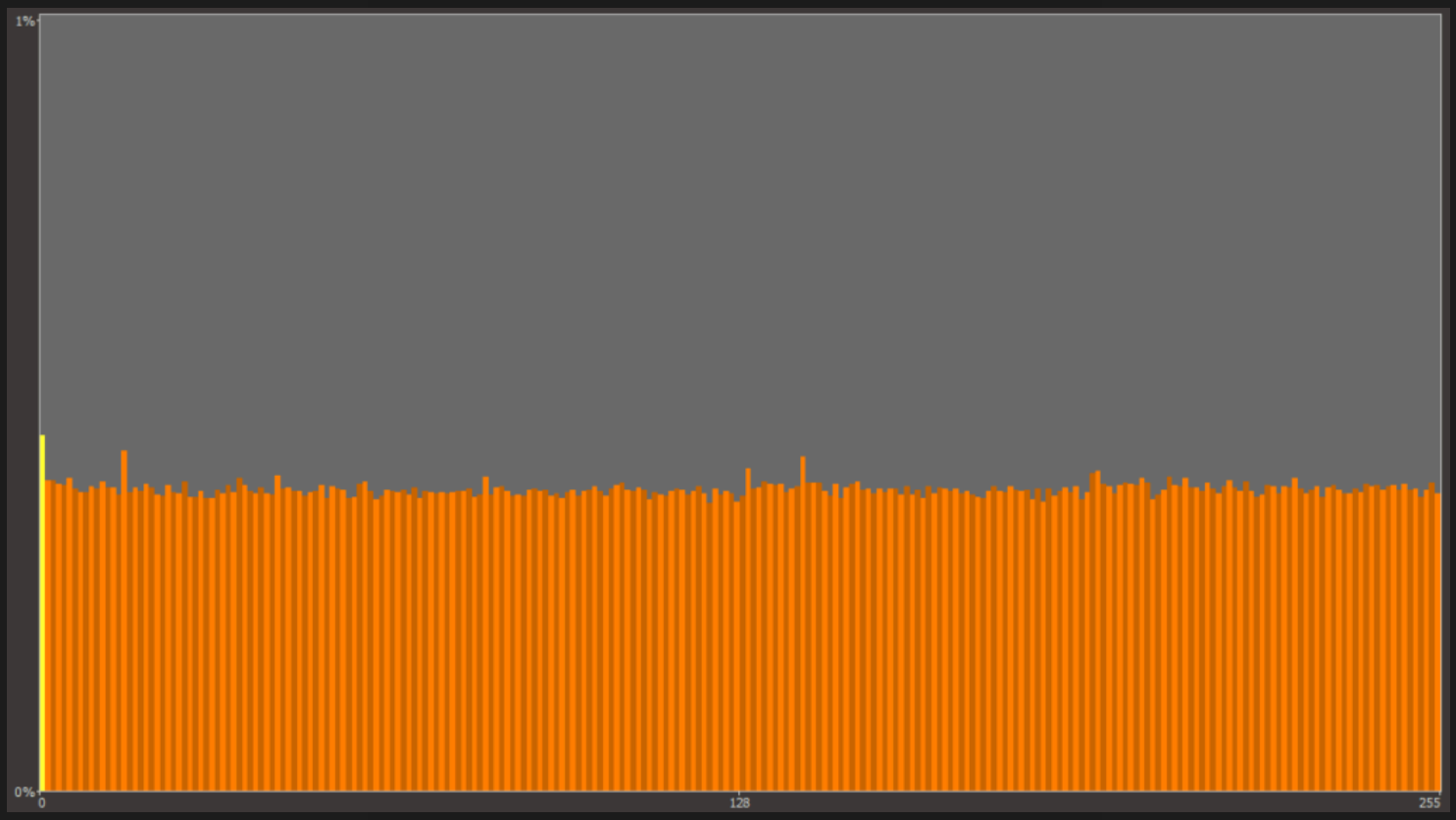
Checking read/write performance on write-combined memory

In this next check, Roshtyak allocates a large memory buffer with the PAGE_WRITECOMBINE flag. This flag is supposed to modify cache behavior to optimize sequential write performance (at the expense of read performance and possibly memory ordering). Roshtyak uses this to detect if it's

running on a physical machine. It conducts an experiment where it first writes to the allocated buffer and then reads from the allocated buffer, all while measuring the read/write performance using a separate thread as a counter. This experiment is repeated 32 times and the check is passed only if write performance was at least six times higher than read performance most of the times. If the check fails, Roshtyak intentionally selects a wrong RC4 key, which results in failing to properly decrypt the next layer.

Hiding shellcode from plain sight

The injected shellcode is interestingly hidden, too. When Roshtyak prepares for code injection, it first creates a large section and maps it into the current process as `PAGE_READWRITE`. Then, it fills the section with random data and places the shellcode at a random offset within the random data. Since the shellcode is just a relatively small loader followed by random-looking packed data, the whole section looks like random data.



A histogram of the bytes inside the shared section. Note that it looks almost random, the most suspicious sign is the slight overrepresentation of null bytes.

The section is then unmapped from the current process and mapped into the target process, where it is executed using the above-described APC injection technique. The random data was added in an attempt to conceal the existence of the shellcode. Judging only from the memory dump of the target process, it might look like the section is full of random data and does not contain any valid executable code. Even if one suspects actual valid code somewhere in the middle of the section, it will not be easy to find its exact location.

```
debug125:00D7AAB5 40                db  40h ; @
debug125:00D7AAB6 57                db  57h ; W
debug125:00D7AAB7 C7                db 0C7h
debug125:00D7AAB8 EA                db 0EAh
debug125:00D7AAB9 CF                db 0CFh
debug125:00D7AABA A0                db 0A0h
debug125:00D7AABB A5                db 0A5h
debug125:00D7AABC 72                db  72h ; r
debug125:00D7AABD 4A                db  4Ah ; J
debug125:00D7AABE A0                db 0A0h
debug125:00D7AABF 15                db  15h
debug125:00D7AAC0                   ; ----------------------
debug125:00D7AAC0
debug125:00D7AAC0                   shellcode_start:
debug125:00D7AAC0 0F A3 CB          bt      ebx, ecx
debug125:00D7AAC3 C0 D2 05          rcl     dl, 5
debug125:00D7AAC6 55                push    ebp
debug125:00D7AAC7 0F CD             bswap   ebp
debug125:00D7AAC9 8B EC             mov     ebp, esp
debug125:00D7AACB 0A CA             or      cl, dl
debug125:00D7AACD 83 E4 F0          and     esp, 0FFFFFFF0h
debug125:00D7AAD0 56                push    esi
debug125:00D7AAD1 0F C8             bswap   eax
debug125:00D7AAD3 57                push    edi
debug125:00D7AAD4 F6 D4             not     ah
```

The start of the shellcode within the shared section. It might be hard to pinpoint the exact start address because it unconventionally starts on an odd `bt` instruction.

Ret2Kernel32

Roshtyak makes a point of cleaning up after itself. Whenever a certain string or piece of memory is no longer needed, Roshtyak wipes and/or frees it in an attempt to destroy as much evidence as possible. The same holds for Roshtyak's layers. Whenever one layer finishes its job, it frees itself before passing execution onto the next layer. However, the layer cannot just simply free itself directly. The whole process would crash if it called `VirtualFree` on the region of memory it's currently executing from.

Roshtyak, therefore, frees the layer through a ROP chain executed during layer transitions to avoid this problem. When a layer is about to exit, it constructs a ROP chain on the stack and returns into it. An example of such a ROP chain can be seen below. This chain starts by returning into `VirtualFree` and `UnmapViewOfFile` to release the previous layer's memory. Then, it returns into the next layer. The return address from the next layer is set to `RtlExitUserThread`, to safeguard execution.

```
dd offset kernelbase_VirtualFree
dd offset kernelbase_UnmapViewOfFile
dd offset loc_A90000                    ; lpAddress for VirtualFree
dd 0                                    ; dwSize for VirtualFree
dd MEM_RELEASE                          ; dwFreeType for VirtualFree
dd offset loc_906008                    ; next layer entrypoint
dd offset loc_A90000                    ; lpBaseAddress for UnmapViewOfFile
dd offset _RtlExitUserThread@4_imported
```

A simple ROP chain consisting of `VirtualFree` -> `UnmapViewOfFile` -> next layer -> `RtlExitUserThread`

MulDiv bug

MulDiv is a function exported by `kernel32.dll`, which takes three signed 32-bit integers as arguments. It multiplies the first two arguments, divides the multiplication result by the third argument, and returns the final result rounded to the nearest integer. While this might seem like a simple enough function, there's an ancient sign extension bug in Microsoft's implementation. This bug is sort of considered a feature now and might never get fixed.

Roshtyak is aware of the bug and tests for its presence by calling `MulDiv(1, 0x80000000, 0x80000000)`. On real Windows machines, this triggers the bug and `MulDiv` erroneously returns `2`, even though the correct return value should be `1`, because `(1 * -2147483648) / -2147483648 = 1`. This allows Roshtyak to detect emulators that do not replicate the bug. For example, this successfully detects Wine, which, funnily enough, contains a different bug, which makes the above call return `0`.

Tampering with return addresses stored on the stack

There are also tricks designed to obfuscate function calls. As shown in the previous section, Roshtyak likes to call functions using the `ret` instruction. This next trick is similar in that it also manipulates the stack so a `ret` instruction can be used to jump to the desired address.

To achieve this, Roshtyak scans the current thread's stack for pointers into the code section of one of the previous layers (unlike the other layers, this one was not freed using the ROP chain technique). It replaces all these pointers with the address it wants to call. Then it lets the code return multiple times until a `ret` instruction encounters one of the hijacked pointers, redirecting the execution to the desired address.

Exception-based checks

Additionally, Roshtyak contains checks that set up a custom vectored exception handler and intentionally trigger various exceptions to ensure they all get handled as expected.

Roshtyak sets up a vectored exception handler using `RtlAddVectoredExceptionHandler`. This handler contains custom handlers for selected exception codes. A top-level exception handler is also registered using `SetUnhandledExceptionFilter`. This handler should not be called in the targeted execution environments (none of the intentionally triggered exceptions should fall through the vectored exception handler). So this top-level handler just contains a single call to `TerminateProcess`. Interestingly, Roshtyak also uses `ZwSetInformationProcess` to set `SEM_FAILCRITICALERRORS` using the `ProcessDefaultHardErrorMode` class. This ensures that even if the exception somehow is passed all the way to the default exception handler, Windows would not show the standard error message box, which could alert the victim that something suspicious is going on.

When everything is set up, Roshtyak begins generating exceptions. The first exception is generated by a `popf` instruction, directly followed by a `cpuid` instruction (shown below). The value popped by the `popf` instruction was crafted to set the trap flag, which should, in turn, raise a single-step exception. On a physical machine, the exception would trigger right after the `cpuid` instruction. Then, the custom vectored exception handler would take over and move the instruction pointer away from the `C7 B2` opcodes, which mark an invalid instruction. However, under many hypervisors, the single-step exception would not be raised. This is because the `cpuid` instruction forces a VM exit, which might delay the effect of the trap flag. If that is the case, the processor will

raise an illegal instruction exception when trying to execute the invalid opcodes. If the vectored exception handler encounters such an exception, it knows that it is running under a hypervisor. A variation of this technique is described thoroughly in a blog post by Palo Alto Networks. Please refer to it for more details.

```
debug105:0F8F6733 D0 D1           rcl     cl, 1
debug105:0F8F6735 8B 4C 24 0C     mov     ecx, [esp+0Ch]
debug105:0F8F6739 0F 97 C6        setnbe  dh
debug105:0F8F673C FE CA           dec     dl
debug105:0F8F673E 13 D9           adc     ebx, ecx
debug105:0F8F6740 87 CA           xchg    ecx, edx
debug105:0F8F6742 0F 97 C1        setnbe  cl
debug105:0F8F6745 D0 D0           rcl     al, 1
debug105:0F8F6747 1A C3           sbb     al, bl
debug105:0F8F6749 9D              popf
debug105:0F8F674A 0F A2           cpuid
debug105:0F8F674A                 ; ------------------------
debug105:0F8F674C C7              db 0C7h
debug105:0F8F674D B2              db 0B2h
debug105:0F8F674E 85              db  85h
debug105:0F8F674F C1              db 0C1h
debug105:0F8F6750 35              db  35h ; 5
```

The exception-based check using `popf` and `cpuid` to detect hypervisors

Another exception is generated using the two-byte `int 3` instruction (`CD 03`). This instruction is followed by garbage opcodes. The `int 3` here raises a breakpoint exception, which is handled by the vectored exception handler. The vectored exception handler doesn't really do anything to handle the exception, which is interesting. This is because by default, when Windows handles the two-byte `int 3` instruction, it will leave the instruction pointer in between the two instruction bytes, pointing to the `03` byte. When disassembled from this `03` byte, the garbage opcodes suddenly start making sense. We believe this is a check against some overeager debuggers, which could "fix" the instruction pointer to point after the `03` byte.

Moreover, the vectored exception handler checks the thread's `CONTEXT` and makes sure that registers `Dr0` through `Dr3` are empty. If they are not, the process is being debugged using hardware breakpoints. While this check is relatively common in malware, the `CONTEXT` is usually

obtained using a call to a function like `GetThreadContext`. Here, the malware authors took advantage of `CONTEXT` being passed as an argument to the exception handler, so they did not need to call any additional API functions.

Large executable mappings

This next check is interesting mostly because we are not sure what it's really supposed to check (in other words, we'd be happy to hear your theories!). It starts with Roshtyak creating a large `PAGE_EXECUTE_READWRITE` mapping of size `0x386F000`. Then it maps this mapping nine times into its own address space. After this, it memsets the mapping to 0x42 (opcode for `inc edx`), except for the last six bytes, which are filled with four `inc ecx` instructions and `jmp dword ptr [ecx]` (see below). Next, it puts the nine base addresses of the mapped views into an array, followed by an address of a single `ret` instruction. Finally, it points `ecx` into this array and calls the first mapped view, which results in all the mapped views being called sequentially until the final `ret` instruction. After the return, Roshtyak validates that `edx` got incremented exactly `0x1FBE6FCA` times (`9 * (0x386F000 - 6)`).

```
debug072:0645EFEF 42                inc     edx
debug072:0645EFF0 42                inc     edx
debug072:0645EFF1 42                inc     edx
debug072:0645EFF2 42                inc     edx
debug072:0645EFF3 42                inc     edx
debug072:0645EFF4 42                inc     edx
debug072:0645EFF5 42                inc     edx
debug072:0645EFF6 42                inc     edx
debug072:0645EFF7 42                inc     edx
debug072:0645EFF8 42                inc     edx
debug072:0645EFF9 42                inc     edx
debug072:0645EFFA 41                inc     ecx
debug072:0645EFFB 41                inc     ecx
debug072:0645EFFC 41                inc     ecx
debug072:0645EFFD 41                inc     ecx
debug072:0645EFFE FF 21             jmp     dword ptr [ecx]
debug072:0645EFFE                   debug072 ends
```

The end of the large mapped section. The `jmp dword ptr [ecx]` instruction is supposed to jump to the start of the next mapped view.

Our best guess is that this is yet another anti-emulator check. For example, in some emulators, mapped sections might not be fully implemented, so the instructions written into one instance of the mapped view might not propagate to the other eight instances. Another theory is the check could be done to request large amounts of memory that emulators might fail to provide. After all, the combined size of all the views is almost half of the standard 32-bit user mode address space.

Detecting process suspension

This trick abuses an undocumented thread creation flag in `NtCreateThreadEx` to detect when Roshtyak's main process gets externally suspended (which could mean that a debugger got attached). This flag essentially allows a thread to keep running even when `PsSuspendProcess` gets called. This is coupled with another trick abusing the fact that the thread suspend counter is a signed 8-bit value, which means that it maxes out at 127. Roshtyak spawns two threads, one of which keeps suspending the other one until the suspend counter limit is reached. After this, the first thread keeps periodically suspending the other one and checking if the call to `NtSuspendThread` keeps failing with `STATUS_SUSPEND_COUNT_EXCEEDED`. If it does not, the thread must have been externally suspended and resumed (which would leave the suspend counter at 126, so the next call to `NtSuspendThread` would succeed). Not getting this error code would be suspicious enough for Roshtyak to quit using `TerminateProcess`. This entire technique is described in more detail in a blog post by Secret Club. We believe that's where the authors of Roshtyak got this trick from. It's also worth mentioning Roshtyak uses this technique only on Windows builds 18323 (19H1) and later because the undocumented thread creation flag was not implemented on prior builds.

Indirect registry writes

Roshtyak performs many suspicious registry operations, for example, setting up the `RunOnce` key for persistence. Since modifications to such keys are likely to be monitored, Roshtyak attempts to circumvent the monitoring. It first generates a random registry key name and temporarily renames the `RunOnce` key to the random name using `ZwRenameKey`. Once renamed, Roshtyak adds a new persistence entry to the temporary key before finally renaming it back to `RunOnce`. This method of writing to the registry can be easily detected, but it might bypass some simple hooking-based monitoring methods.

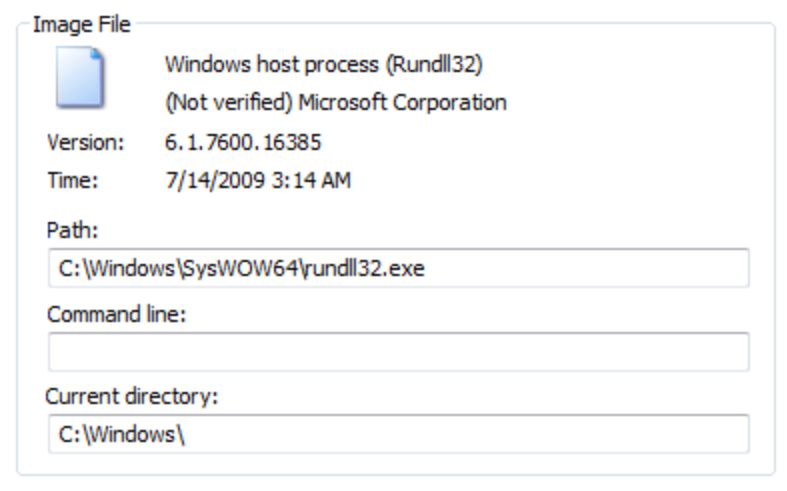
Similarly, there are multiple methods Roshtyak uses to delete files. Aside from the apparent call to `NtDeleteFile`, Roshtyak is able to effectively delete a file by setting `FileDispositionInformation` or `FileRenameInformation` in a call to `ZwSetInformationFile`. However, unlike the registry modification method, this doesn't seem to be implemented in order to evade detection. Instead, Roshtyak will try these alternative methods if the initial call to `NtDelete` file fails.

Checking VBAWarnings

The `VBAWarnings` registry value controls how Microsoft Office behaves when a user opens a document containing embedded VBA macros. If this value is `1` (meaning "Enable all macros"), macros are executed by default, even without the need for any user interaction. This is a common setting for sandboxes, which are designed to detonate maldocs automatically. On the other hand, this setting is uncommon for regular users, who generally don't go around changing random settings to make themselves more vulnerable (at least most of them don't). Roshtyak therefore uses this check to differentiate between sandboxes and regular users and refuses to run further if the value of `VBAWarnings` is `1`. Interestingly, this means that users, who for whatever reason have lowered their security this way, are immune to Roshtyak.

Command line wiping

Roshtyak's core is executed with very suspicious command lines, such as `RUNDLL32.EXE SHELL32.DLL,ShellExec_RunDLL REGSVR32.EXE -U /s "C:\Users\<REDACTED>\AppData\Local\Temp\dpcw.etl."`. These command lines don't look particularly legitimate, so Roshtyak attempts to hide them during execution. It does this by wiping command line information collected from various sources. It starts by calling `GetCommandLineA` and `GetCommandLineW` and wiping both of the returned strings. Then it attempts to wipe the string pointed to by `PEB->ProcessParameters->CommandLine` (even if this points to a string that has already been wiped). Since Roshtyak is often running under WoW64, it also calls `NtWow64QueryInformationProcess64` to obtain a pointer to `PEB64` to wipe `ProcessParameters->CommandLine` obtained by traversing this "second" PEB. While the wiping of the command lines was probably meant to make Roshtyak look more legitimate, the complete absence of any command line is also highly unusual. This was noticed by the Red Canary researchers in their blog post, where they proposed a detection method based on these suspiciously empty command lines.



Roshtyak's core process, as shown by Process Explorer. Note the suspiciously empty command line.

Additional tricks

Aside from the techniques described so far, Roshtyak uses many less sophisticated tricks that are commonly found in other malware as well. These include:

- Hiding threads using `ThreadHideFromDebugger` (and verifying that the threads really got hidden using `NtQueryInformationThread`)
- Patching `DbgBreakPoint` in `ntdll`
- Detecting user inactivity using `GetLastInputInfo`
- Checking fields from PEB (`BeingDebugged`, `NtGlobalFlag`)
- Checking fields from `KUSER_SHARED_DATA` (`KdDebuggerEnabled`, `ActiveProcessorCount`, `NumberOfPhysicalPages`)
- Checking the names of all running processes (some are compared by hash, some by patterns, and some by character distribution)
- Hashing the names of all loaded modules and checking them against a hardcoded blacklist
- Verifying the main process name is not too long and doesn't match known names used in sandboxes
- Using the `cpuid` instruction to check hypervisor information and the processor brand
- Using poorly documented COM interfaces
- Checking the username and computername against a hardcoded blacklist
- Checking for the presence of known sandbox decoy files

- Checking MAC addresses of own adapters against a hardcoded blacklist
- Checking MAC addresses from the ARP table (using `GetBestRoute` to populate it and `GetIpNetTable` to inspect it)
- Calling `ZwQueryInformationProcess` with `ProcessDebugObjectHandle`, `ProcessDebugFlags`, and `ProcessDebugPort`
- Checking `DeviceId` of display devices (using `EnumDisplayDevices`)
- Checking `ProductId` of `\\.\PhysicalDrive0` (using `IOCTL_STORAGE_QUERY_PROPERTY`)
- Checking for virtual hard disks (using `NtQuerySystemInformation` with `SystemVhdBootInformation`)
- Checking the raw SMBIOS firmware table (using `NtQuerySystemInformation` with `SystemFirmwareTableInformation`)
- Setting up Defender exclusions (both for paths and processes)
- Removing IFEO registry keys related to process names used by the malware

Obfuscation

We've shown many anti-analysis tricks that are designed to prevent Roshtyak from detonating in undesirable execution environments. These tricks alone would be easy to patch or bypass. What makes analyzing Roshtyak especially lethal is the combination of all these tricks with heavy obfuscation and multiple layers of packing. This makes it very difficult to study the anti-analysis tricks statically and figure out how to pass all the checks in order to get Roshtyak to unpack itself. Furthermore, even the main payload received the same obfuscation, which means that statically analyzing Roshtyak's core functionality also requires a great deal of deobfuscation.

In the rest of this section, we'll go through the main obfuscation techniques used by Roshtyak.

```
v55 = 9470521 * v54;
v54 *= 33147324;
v394 = ~((a8 ^ 0x2A0A7F1) & 0xE9D3E915) & 0x7F3C5EFF;
v446 = v55 - 1708702459;
_ECX = v311 + (v54 < (a8 ^ 0xDF3593FD)) + v394;
__asm { rcr     ecx, cl }
v375 = 1207367791 - (a8 ^ 0xDF916B44);
v421 = (a8 ^ 0x9C0520F2) + 385570526;
LOWORD(_ECX) = ((unsigned __int8)a8 ^ 1) & 7;
v395 = a8 ^ 0xCD418EB0;
__asm { rcl     ch, 3 }
v396 = (6198429 << _ECX) | 0x7A;
v349 = 33715423 * ((unsigned __int8)a8 ^ 0x45);
__asm { rcl     ah, 4 }
v346 = 1757652824 - (a8 ^ 0xC535F31B);
v324 = (a8 ^ 0xCF4D7899) & 0x40808003 | 0x81061010;
__asm { rcr     eax, 0Fh }
```

A random code snippet from Roshtyak. As can be seen, the obfuscation makes the raw output of the Hex-Rays decompiler practically incomprehensible.

Control flow flattening

Control flow flattening is one of the most noticeable obfuscation techniques employed by Roshtyak. It is implemented in an unusual way, giving the control flow graphs of Roshtyak's functions a unique look (see below). The goal of control flow flattening is to obscure control flow relations between individual code blocks.

Control flow is directed by a 32-bit control variable, which tracks the execution state, identifying the code block to be executed. This control variable is initialized at the start of each function to refer to the starting code block (which is frequently a `nop` block). The control variable is then modified at the end of each code block to identify the next code block that should be executed. The modification is performed using some arithmetic instructions, such as `add`, `sub`, or `xor`.

There is a dispatcher using the control variable to route execution into the correct code block. This dispatcher is made up of if/else blocks that are circularly linked into a loop. Each dispatcher block takes the control variable and masks it using arithmetic instructions to check if it should route execution into the code block that it is guarding. What's interesting here is there are multiple points of entry from the code blocks into the dispatcher loop, giving the control flow graphs the jagged "sawblade" look in IDA.

Branching is performed using a special code block containing an `imul` instruction. It relies on the previous block to compute a branch flag. This branch flag is multiplied using the `imul` instruction with a random constant, and the result is added, subbed, or xored to the new control variable. This means that after the branch block, the control variable will identify one of the two possible succeeding code blocks, depending on the value that was computed for the branch flag.

Control flow graph of a function obfuscated using control flow flattening

Function activation keys

Roshtyak's obfuscated functions expect an extra argument, which we call an *activation key*. This activation key is used to decrypt all local constants, strings, variables, etc. If a function is called with a wrong activation key, the decryption results in garbage plaintext, which will most likely cause Roshtyak to get stuck in an infinite loop inside the control flow dispatcher. This is because all constants used by the dispatcher (the initial value of the control variable, the masks used by the

dispatcher guards, and the constants used to jump to the next code block) are encrypted with the activation key. Without the correct activation key, the dispatcher simply does not know how to dispatch.

Reverse engineering a function is practically impossible without knowing the correct activation key. All strings, buffers, and local variables/constants remain encrypted, all cross-references are lost, and worse, there is no control flow information. Only individual code blocks remain, with no way to know how they relate to each other.

Each obfuscated function has to be called from somewhere, which means the code calling the function has to supply the correct activation key. However, obtaining the activation key is not that easy. First, call targets are also encrypted with activation keys, so it's impossible to find where a function is called from without knowing the right activation keys. Second, even the supplied activation key is encrypted with the activation key of the calling function. And that activation key got encrypted with the activation key of the next calling function. And so on, recursively, all the way until the entry point function.

This brings us to how to deobfuscate the mess. The activation key of the entry point function must be there in plaintext. Using this activation key, it is possible to decrypt the call targets and activation keys of functions that are called directly from this entry point function. Applying this method recursively allows us to reconstruct the full call graph along with the activation keys of all the functions. The only exceptions would be functions that were never called and were left in by the compiler. These functions will probably remain a mystery, but since the sample does not use them, they are not that important from a malware analyst's point of view.

Variable masking

Some variables are not stored in plaintext form but are masked using one or more arithmetic instructions. This means that if Roshtyak is not actively using a variable, it keeps the variable's value in an obfuscated form. Whenever Roshtyak needs to use the variable, it has to first unmask it before it can use it. Conversely, after Roshtyak uses the variable, it converts it back into the masked form. This masking-based obfuscation method slightly complicates tracking variables during debugging and makes it harder to search memory for a known variable value.

Loop transformations

Roshtyak is creative with some loop conditions. Instead of writing a loop like `for (int i = 0; i < 1690; i++)`, it transforms the loop into e.g. `for (int32_t i = 0x06AB91EE; i != 0x70826068; i = i * -0x509FFFF + 0xEC891BB1)`. While both loops will execute exactly 1690 times, the second one is much harder to read. At first glance, it is not clear how many iterations the second loop executes (and if it even terminates). Tracking the number of loop iterations during debugging is also much harder in the second case.

Packing

As mentioned, Roshtyak's core is hidden behind multiple layers of packing. While all the layers look like they were originally compiled into PE files, all but the strictly necessary data (entry point, sections, imports, and relocations) were stripped away. Furthermore, Roshtyak supports two custom formats for storing the stripped PE file information, and the layers take turns on what format they use. Additionally, parts of the custom formats are encrypted, sometimes using keys generated based on the results of various anti-analysis checks.

This makes it difficult to unpack Roshtyak's layers statically into a standalone PE file. First, one would have to reverse engineer the custom formats and figure out how to decrypt the encrypted parts. Then, one would have to reconstruct the PE header, the sections, the section headers, and the import table (the relocation table doesn't need to be reconstructed since relocations can just be turned off). While this is all perfectly doable (and can be simplified using libraries like LIEF), it might take a significant amount of time. Adding to this that the layers are sometimes interdependent, it might be easier to just analyze Roshtyak dynamically in memory.

```
.data:200070A0 first_section   dd 4B40h                ; raw_size
.data:200070A0                                         ; DATA XREF: loader+2945↑o
.data:200070A0                 dd 4E0h                 ; virtual_padding_size
.data:200070A0                 dd 1B906AA2h            ; encryption_key
.data:200070A0                 db 29h, 43h, 0DCh, 0E3h, 0E5h, 4Eh, 12h, 6Fh, 2, 9Eh, 6Eh; encrypted_section
.data:200070A0                 db 88h, 49h, 31h, 5Eh, 37h, 6Bh, 0C2h, 0BFh, 5Fh, 33h; encrypted_section
.data:200070A0                 db 2Ah, 6Ch, 0E8h, 37h, 0E2h, 89h, 0EDh, 0B2h, 5Dh, 96h; encrypted_section
.data:200070A0                 db 0B5h, 3Eh, 98h, 97h, 68h, 4Dh, 0AAh, 0C0h, 1Dh, 93h; encrypted_section
.data:200070A0                 db 0D5h, 0ADh, 0F4h, 9Eh, 32h, 0C4h, 0A3h, 56h, 9, 7, 0B3h; encrypted_section
.data:200070A0                 db 5Ah, 88h, 6Fh, 0Bh, 0E5h, 98h, 64h, 0Ch, 61h, 0E3h; encrypted_section
.data:200070A0                 db 0A1h, 0D8h, 5Fh, 0A1h, 0E7h, 0DEh, 81h, 0AEh, 43h, 0BFh; encrypted_section
.data:200070A0                 db 83h, 38h, 51h, 1Eh, 0C5h, 85h, 0A3h, 95h, 0EBh, 9Eh; encrypted_section
.data:200070A0                 db 3, 5Ah, 0B9h, 34h, 58h, 10h, 0AEh, 0C9h, 0C1h, 4Bh; encrypted_section
.data:200070A0                 db 35h, 74h, 92h, 9Ch, 0BAh, 69h, 0EAh, 0F8h, 0DCh, 0BBh; encrypted_section
.data:200070A0                 db 0B6h, 0C4h, 12h, 47h, 85h, 0EDh, 6Fh, 0FEh, 0ECh, 65h; encrypted_section
.data:200070A0                 db 8Dh, 0FAh, 8, 33h, 46h, 95h, 0CBh, 9Ch, 0D8h, 8Bh, 0A7h; encrypted_section
```

A section header in one of the custom PE-like file formats: `raw_size` corresponds to `SizeOfRawData`, `raw_size + virtual_padding_size` is effectively `VirtualSize`. There is no `VirtualAddress` or `PointerToRawData` equivalent because the sections are loaded sequentially.

Other obfuscation techniques

In addition to the above-described techniques, Roshtyak also uses other obfuscation techniques, including:

- Junk instruction insertion
- Import hashing
- Frequent memory wiping
- Mixed boolean-arithmetic obfuscation
- Redundant threading
- Heavy polymorphism

Core Functionality

Now that we've described how Roshtyak protects itself, it might be interesting to also go over what it actually does. Roshtyak's DLL is relatively large, over a megabyte, but its functionality is surprisingly simple once you eliminate all the obfuscation. Its main purpose is to download further payloads to execute. In addition, it does the usual evil malware stuff, namely establishing persistence, escalating privileges, lateral movement, and exfiltrating information about the victim.

Persistence

Roshtyak first generates a random file name in `%SystemRoot%\Temp` and moves its DLL image there. The generated file name consists of two to eight random lowercase characters concatenated with a random extension chosen from a hardcoded list. The PRNG used to generate this file name is seeded with the volume serial number of `C:\`. The sample we analyzed hardcoded seven extensions (`.log`, `.tmp`, `.loc`, `.dmp`, `.out`, `.ttf`, and `.etl`). We observed other extensions being used in other samples, suggesting this list is somewhat dynamic. With a small probability, Roshtyak will also use a randomly generated extension. Once fully constructed, the full path to the Roshtyak DLL might look like e.g. `C:\Windows\Temp\wcdp.etl`.

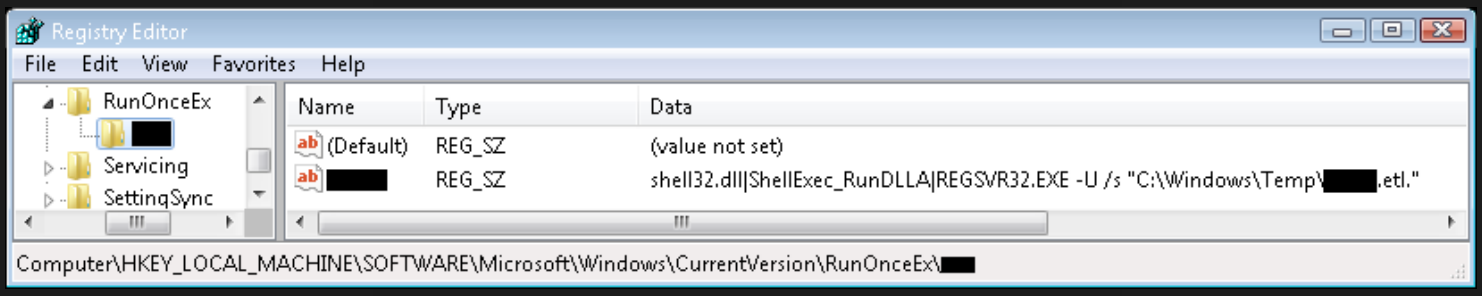
After the DLL image is moved to the new filesystem path, Roshtyak stomps its `Modified` timestamp to the current system time. It then proceeds to set up a `RunOnce(Ex)` registry key to actually establish persistence. The registry entry is created using the previously described indirect registry write technique. The command inserted into the key might look as follows:

```
RUNDLL32.EXE SHELL32.DLL,ShellExec_RunDLL REGSVR32.EXE -U /s "C:\Windows\Temp\wcdp.etl."
```

There are a couple of things to note here. First, `regsvr32` doesn't care about the extensions of the DLLs it loads, allowing Roshtyak to hide under an innocent-looking extension such as `.log`. Second, the `/s` parameter puts `regsvr32` into silent mode. Without it, `regsvr32` would complain that it did not find an export named `DllUnregisterServer`. Finally, notice the trailing period character at the end of the path. This period is removed during path normalization, so it practically has no effect on the command. We are not exactly sure what the author's original intention behind including this period character is. It looks like it could have been designed to trick some anti-malware software into not being able to connect the persistence entry with the payload on the filesystem.

By default, Roshtyak uses the `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce` key for persistence. However, under some circumstances (such as when it detects that Kaspersky is running by checking for a process named `avp.exe`) the key `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnceEx` will be used instead. The

`RunOnceEx` key is capable of loading a DLL, so when using this key, Roshtyak specifies `shell32.dll` directly, omitting the use `rundll32`.



A `RunOnceEx` persistence entry established by Roshtyak

Privilege escalation

Roshtyak uses both UAC bypasses and regular EoP exploits in an attempt to elevate its privileges. Unlike many other pieces of malware, which just blindly execute whatever UAC bypasses/exploits the authors could find, Roshtyak makes efforts to figure out if the privilege escalation method is even likely to be successful. This was probably implemented to lower the chances of detection due to the unnecessary usage of incompatible bypasses/exploits. For UAC bypasses, this involves checking the `ConsentPromptBehaviorAdmin` and `ConsentPromptBehaviorUser` registry keys. For EoP exploits, this is about checking the Windows build number and patch level.

Besides checking the `ConsentPromptBehavior(Admin|User)` keys, Roshtyak performs other sanity checks to ensure that it should proceed with the UAC bypass. Namely, it checks for admin privileges using `CheckTokenMembership` with the SID `S-1-5-32-544` (`DOMAIN_ALIAS_RID_ADMINS`). It also inspects the value of the `DbgElevationEnabled` flag in `KUSER_SHARED_DATA.SharedDataFlags`. This is an undocumented flag that is set if UAC is enabled. Finally, there are AV checks for BitDefender (detected by the module `atcuf32.dll`), Kaspersky (process `avp.exe`), and our own Avast/AVG (module `aswhook.dll`). If one of these AVs is detected, Roshtyak avoids selected UAC bypass techniques, presumably the ones that might result in detection.

As for the actual UAC bypasses, there are two main methods implemented. The first is an implementation of the aptly named ucmDccwCOM method from UACMe. Interestingly when this method is executed, Roshtyak temporarily masquerades its process as `explorer.exe` by

overwriting `FullDllName` and `BaseDllName` in the _LDR_MODULE structure corresponding to the main executable module. The payload launched by this method is a randomly named LNK file, dropped into `%TEMP%` using the IShellLink COM interface. This LNK file is designed to relaunch the Roshtyak DLL, through LOLBins such as `advpack` or `register-cimprovider`.

The second method is more of a UAC bypass framework than a specific bypass method, because multiple UAC bypass methods follow the same simple pattern: first registering some specific shell open command and then executing an autoelevating Windows binary (which internally triggers the shell open command). For instance, a UAC bypass might be accomplished by writing a payload command to `HKCU\Software\Classes\ms-settings\shell\open\command` and then executing `fodhelper.exe` from `%windir%\system32`. Basically, the same bypass can be achieved by substituting the pair `ms-settings`/`fodhelper.exe` with other pairs, such as `mscfile`/`eventvwr.exe`. Roshtyak uses the following six pairs to bypass UAC:

Class	Executable
`mscfile`	`eventvwr.exe`
`mscfile`	`compmgmtlauncher.exe`
`ms-settings`	`fodhelper.exe`
`ms-settings`	`computerdefaults.exe`
`Folder`	`sdclt.exe`
`Launcher.SystemSettings`	`slui.exe`

Let's now look at the kernel exploits (CVE-2020-1054 and CVE-2021-1732) Roshtyak uses to escalate privileges. As is often the case in Roshtyak, these exploits are stored encrypted and are only decrypted on demand. Interestingly, once decrypted, the exploits turn out to be regular PE files with completely valid headers (unlike the other layers in Roshtyak, which are either in shellcode form or stored in a custom stripped PE format). Moreover, the exploits lack the obfuscation given to the rest of Roshtyak, so their code is immediately decompilable, and only some basic string encryption is used. We don't know why the attackers left these exploits so

exposed, but it might be due to the difference in bitness. While Roshtyak itself is x86 code (most of the time running under WoW64), the exploits are x64 (which makes sense considering they exploit vulnerabilities in 64-bit code). It could be that the obfuscation tools used by Roshtyak's authors were designed to work on x86 and are not portable to x64.

```
IsMenu = GetProcAddress(HandleUser32, ProcName);
v12 = IsMenu;
if ( IsMenu )
{
  for ( i = 0; ; ++i )
  {
    v14 = i + 1;
    if ( *(_BYTE *)IsMenu == 0xE8 )
      break;
    IsMenu = (FARPROC)((char *)IsMenu + 1);
    if ( (unsigned int)v14 >= 0x20 )
      return v0;
  }
  if ( i != -1 )
    return (__int64)v9 + (_DWORD)v12 + *(_DWORD *)((char *)v12 + v14) - (_DWORD)v9 + i + 5;
}
```

Snippet from Roshtyak's exploit for CVE-2020-1054, scanning through `IsMenu` to find the offset to HMValidateHandle.

To execute the exploits, Roshtyak spawns (the AMD64 version of) `winver.exe` and gets the exploit code to run there using the KernelCallbackTable injection method. Roshtyak's implementation of this injection method essentially matches a public PoC, with the biggest difference being the usage of slightly different API functions due to the need for cross-subsystem injection (e.g. `NtWow64QueryInformationProcess64` instead of `NtQueryInformationProcess` or `NtWow64ReadVirtualMemory64` instead of `ReadProcessMemory`). The code injected into `winver.exe` is not the exploit PE itself but rather a slightly obfuscated shellcode, designed to load the exploit PE into memory.

The kernel exploits target certain unpatched versions of Windows. Specifically, CVE-2020-1054 is only used on Windows 7 systems where the revision number is not higher than `24552`. On the other hand, the exploit for CVE-2021-1732 runs on Windows 10, with the targeted build number range being from `16353` to `19042`. Before exploiting CVE-2021-1732, Roshtyak also scans through installed update packages to see if a patch for the vulnerability is installed. It does this by enumerating the registry keys under

`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Component Based Servicing\Packages` and checking if the package for KB4601319 (or higher) is present.

Lateral movement

When it comes to lateral movement, Roshtyak simply uses the tried and tested PsExec tool. Before executing PsExec, Roshtyak ensures it makes sense to run it by checking for a SID matching the "well-known" `WinAccountDomainAdminsSid` group. If domain admin rights are not detected, Roshtyak skips its lateral movement phase entirely.

Roshtyak attempts to get around detection by setting Defender exclusions, as PsExec is often flagged as a hacktool (for good reasons). It sets a path exclusion for `%TEMP%` (where it will drop PsExec and other files used for lateral movement). Later, it sets up a process exclusion for the exact path from which PsExec will be executed.

While we would expect PsExec to be bundled inside Roshtyak, it turns out Roshtyak downloads it on demand from `https://download.sysinternals[.]com/files/PSTools.zip`. The downloaded zip archive is dropped into `%TEMP%` under a random name with the `.zip` extension. PsExec is then unzipped from this archive using the Windows Shell COM interface (`IShellDispatch`) into a randomly named `.exe` file in `%TEMP%`.

The payload to be executed by PsExec is a self-extracting package created by a tool called IExpress. This is an archaic installer that's part of Windows, which is probably why it's used, since Roshtyak can rely on it already being on the victim machine. The installer generation is configured by a text file using the Self Extraction Directive (SED) syntax.

```
[OPtIONS]
poSTInstaLLCMD=<None>
APplAuNChed=RegSvr32 -U -S "%2"
targeTnAMe="%1"
USeLonGfilenAmE=1
ShOwiNStaLLProGRAMWiNDOW=1
sOUrCEfIles=WfyAlLqdNN
hIDeExTRActAnIMATIon=1
rEboOtmOdE=0
[wFYaLLQdNn]
RjjEhnUSMDpxmcpA=
[VErSion]
cLasS=IEXprEsS
seDVERSion=3
[RJjEhNusmdpXMCPA]
%3=
```

Roshtyak's IExpress configuration template

Roshtyak uses a SED configuration template with three placeholders (`%1`, `%2`, and `%3`) that it substitutes with real values at runtime. As seen above, the configuration template was written in mixed-case, which is frequently used in Raspberry Robin in general. Once the SED configuration is prepared, it is written into a randomly named `.txt` file in `%TEMP%`. Then, `iexpress` is invoked to generate the payload using a command such as `C:\Windows\iexpress.exe /n /q <path_to_sed_config>`. The generated payload is dumped into a randomly named `.exe` file in `%TEMP%`, as configured by the `TargetName` directive (placeholder `%1`).

Once the payload is generated, Roshtyak proceeds to actually run PsExec. There are two ways Roshtyak can execute PsExec. The first one uses the command `<path_to_psexec> \\* -accepteula -c -d -s <path_to_payload>`. Here, the `\\*` wildcard instructs PsExec to run the payload on all computers in the current domain. Alternatively, Roshtyak might run the command `<path_to_psexec> @<path_to_target_file> -accepteula -c -d -s <path_to_payload>`. Here, the `target_file` is a text file containing a specific list of computers to run the payload on. Roshtyak builds this list by enumerating Active Directory objects using API functions exported from `activeds.dll`.

Profiling the victim

USB worms tend to have a life of their own. Since their worming behavior is usually completely automated, the threat actor who initially deployed the worm doesn't necessarily have full control over where it spreads. This is why it's important for threat actors to have the worm beacon back to their C&C servers. With a beaconing mechanism in place, the threat actor can be informed about all the machines under their control and can use this knowledge to manage the worm as a whole.

The outgoing beaconing messages typically contain some information about the infected machine. This helps financially-motivated cybercriminals decide on how to best monetize the infection. Roshtyak is no exception to this, and it collects a lot of information about each infected victim. Roshtyak concatenates all the collected information into a large string, using semicolons as delimiters. This large string is then exfiltrated to one of Roshtyak's C&C servers. The exfiltrated pieces of information are listed below, in order of concatenation.

- External IP address (obtained during a Tor connectivity check)
- A string hardcoded into Roshtyak's code, e.g. `AFF123` (we can't be sure what's the meaning behind this, but it looks like an affiliate ID)
- A 16-bit hash of the DLL's PE header (with some fields zeroed out) xored with the lower 16 bits of its `TimeDateStamp`. The `TimeDateStamp` appears to be specially crafted so that the xor results in a known value. This could function as a tamper check or a watermark.
- Creation timestamp of the `System Volume Information` folder on the system drive
- The volume serial number of the system drive
- Processor count (`GetActiveProcessorCount`)
- IsWow64Process (`_PROCESS_EXTENDED_BASIC_INFORMATION.Flags & 2`)
- Windows version (`KUSER_SHARED_DATA.Nt(Major|Minor)Version`)
- Windows product type (`KUSER_SHARED_DATA.NtProductType`)
- Windows build number (`PEB.OSBuildNumber`)
- Local administrative privileges (`ZwQueryInformationToken(TokenGroups)`/`CheckTokenMembership`, check for `DOMAIN_ALIAS_RID_ADMINS`)
- Domain administrative privileges (check for `WinAccountDomainAdminsSid`/`WinAccountDomainUsersSid`)
- System time (`KUSER_SHARED_DATA.SystemTime`)

- Time zone (`KUSER_SHARED_DATA.TimeZoneBias`)
- System locale (`NtQueryDefaultLocale(0)`)
- User locale (`NtQueryDefaultLocale(1)`)
- Environment variables (`username`, `computername`, `userdomain`, `userdnsdomain`, and `logonserver`)
- Java version (`GetFileVersionInfo("javaw.exe")` -> `VerQueryValue`)
- Processor information (`cpuid` to obtain the `Processor Brand String`)
- Path to the image of the main executable module (`NtQueryVirtualMemory(MemorySectionName)`)
- Product ID and serial number of the main physical drive (`DeviceIoControl(IOCTL_STORAGE_QUERY_PROPERTY, StorageDeviceProperty)`)
- MAC address of the default gateway (`GetBestRoute` -> `GetIpNetTable`)
- MAC addresses of all network adapters (`GetAdaptersInfo`)
- Installed antivirus software (`root\securitycenter2` -> `SELECT * FROM AntiVirusProduct`)
- Display device information (`DeviceId`, `DeviceString`, `dmPelsWidth`, `dmPelsHeight`, `dmDisplayFrequency`) (`EnumDisplayDevices` -> `EnumDisplaySettings`)
- Active processes (`NtQuerySystemInformation(SystemProcessInformation)`)
- Screenshot encoded in base64 (`gdi32` method)

Beaconing

Once collected, Roshtyak sends the victim profile to one of its C&C servers. The profile is sent over the Tor network, using a custom comms module Roshtyak injects into a newly spawned process. The C&C server processes the exfiltrated profile and might respond with a shellcode payload for the core module to execute.

Let's now take a closer look at this whole process. It's worth mentioning that before generating any malicious traffic, Roshtyak first performs a Tor connectivity check. This is done by contacting 28 legitimate and well-known `.onion` addresses in random order and checking if at least one of them responds. If none of them respond, Roshtyak doesn't even attempt to contact its C&C, as it would most likely not get through to it anyway.

As for the actual C&C communication, Roshtyak contains 35 hardcoded V2 onion addresses (e.g. `ip2djbz3xidmkmkw:53148`, see our IoC repository for the full list). Like during the connectivity check, Roshtyak iterates through them in random order and attempts to contact each of them until one responds. Note that while V2 onion addresses are officially deprecated in favor of V3 addresses (and the Tor Browser no longer supports them in its latest version) they still appear to be functional enough for Roshtyak's nefarious purposes.

```
0026AE50 dd offset aIp2djbz3xidmkm      ; "ip2djbz3xidmkmkw:53148"
0026AE54 dd offset aTq2srsgevhutzw      ; "tq2srsgevhutzw42:43477"
0026AE58 dd offset aKrq2qyjhfwh4tr      ; "krq2qyjhfwh4trww:51499"
0026AE5C dd offset aXph6exfmdo7b4t      ; "xph6exfmdo7b4tkw:38607"
0026AE60 dd offset aP2dw3umgw6qhrl      ; "p2dw3umgw6qhrld3:25947"
0026AE64 dd offset a2q3n7ycm7vxe73      ; "2q3n7ycm7vxe73g6:30656"
0026AE68 dd offset aKzzuxfvchn5kb7      ; "kzzuxfvchn5kb73c:21646"
0026AE6C dd offset a42xgf6qae5wjbc      ; "42xgf6qae5wjbcva:45252"
0026AE70 dd offset aMiwia5zo4oxcj7      ; "miwia5zo4oxcj7n6:11472"
0026AE74 dd offset aGk7jrmr5v3nw3u      ; "gk7jrmr5v3nw3u7m:40090"
0026AE78 dd offset aNwogcq7cmhth7e      ; "nwogcq7cmhth7e4x:15588"
0026AE7C dd offset aLbwgagk54ww5c3      ; "lbwgagk54ww5c3nj:32284"
0026AE80 dd offset aRe5sb73yb75nbk      ; "re5sb73yb75nbkrm:33033"
0026AE84 dd offset a5fajnveyn2bd4n      ; "5fajnveyn2bd4nm7:5990"
0026AE88 dd offset aSv2fubnuttyzvf      ; "sv2fubnuttyzvfgl:39828"
0026AE8C dd offset aXh6pciiw6yeqz3      ; "xh6pciiw6yeqz3bs:19956"
0026AE90 dd offset aXup6y7cxgjorez      ; "xup6y7cxgjorezif:51516"
```

Roshtyak's hardcoded C&C addresses

The victim profile is sent in the URL path, appended to the V2 onion address, along with the `/` character. As the raw profile might contain characters forbidden for use in URLs, the profile is wrapped in a custom structure and encoded using Base64. The very first 0x10 bytes of the custom structure serve as an encryption key, with the rest of the structure being encrypted. The custom structure also contains a 64-bit hash of the victim profile, which presumably serves as an integrity check. Interestingly, the custom structure might get its end padded with random bytes. Note that the full path could be pretty large, as it contains a doubly Base64-encoded screenshot. The authors of Roshtyak were probably aware that the URL path is not suitable for sending large amounts of data and decided to cap the length of the victim profile at 0x20000 bytes. If the screenshot makes the exfiltrated profile larger than this limit, it isn't included.

When the full onion URL is constructed, Roshtyak goes ahead to launch its Tor comms module. It first spawns a dummy process to host the comms module. This dummy process is randomly chosen and can be one of `dllhost.exe`, `regsvr32.exe`, or `rundll32.exe`. The comms module is injected into the newly spawned process using a shared section, obfuscated through the previously described shellcode hiding technique. The comms module is then executed via `NtQueueApcThreadEx`, using the already discussed ntdll gadget trick. The injected comms module is a custom build of an open-source Tor library packed in three additional protective shellcode layers.

The core module communicates with the comms module using shared sections as an IPC mechanism. Both modules synchronously use the same PRNG with the same seed (`KUSER_SHARED_DATA.Cookie`) to generate the same section name. Both then map this named section into their respective address spaces and communicate with each other by reading/writing to it. The data read/written into the section is encrypted with RC4 (the key also generated using the synchronized PRNGs).

The communication between the core module and the comms module follows a simple request/response pattern. The core module writes an encrypted onion URL (including the URL path to exfiltrate) into the shared section. The comms module then decrypts the URL and makes an HTTP request over Tor to it. The core module waits for the comms module to write the encrypted HTTP response back to the shared section. Once it's there, the core module decrypts it and unwraps it from a custom format (which includes decrypting it yet again and computing a hash to check the payload's integrity). The decrypted payload might include a shellcode for the core module to execute. If the shellcode is present, the core module allocates a huge chunk of memory, hides the shellcode there using the shellcode hiding technique, and executes it in a new thread. This new thread is hidden using the `NtSetInformationThread -> ThreadHideFromDebugger` technique (including a follow-up anti-hooking check using `NtGetInformationThread` to confirm that the `NtSetInformationThread` call did indeed succeed).

Conclusion

In this blog post, we took a technical deep dive into Roshtyak, the backdoor payload associated with Raspberry Robin. The main focus was to describe how to deal with Roshtyak's protection mechanisms. We showed some never-before-seen anti-debugger/anti-sandbox/anti-VM tricks and

discussed Roshtyak's heavy obfuscation. We also described Roshtyak's core functionality. Specifically, we detailed how it establishes persistence, escalates privileges, moves laterally, and uses Tor to download further payloads.

We have to admit that reverse engineering Roshtyak was certainly no easy task. The combination of heavy obfuscation and numerous advanced anti-analysis tricks made it a considerable challenge. Nick Harbour, if you're looking for something to repurpose for next year's final Flare-On challenge, this might be it.

Indicators of Compromise (IoCs)

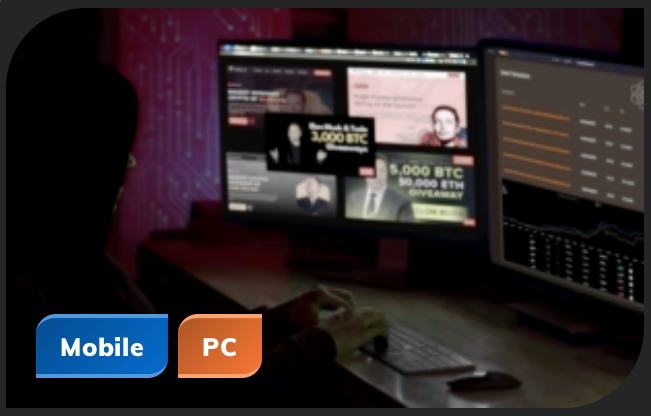
IoCs are available at https://github.com/avast/ioc/tree/master/RaspberryRobin.

Tagged as

CVE-2020-1054 **CVE-2021-1732** **Rapsberry Robin** **Roshtyak**

Share:

Further reading



CryptoCore: Unmasking the Sophisticated Cryptocurrency Scam Operations

August 13, 2024 - by **Martin Chlumecký**

As digital currencies have grown, so have cryptocurrency scams, posing significant user risks. The rise of AI and deepfake technology has intensified scams exploiting famous personalities and events by creating realistic fake videos. Platforms like X and YouTube have been especially targeted, with...

Decrypted: DoNex Ransomware and its Predecessors

July 8, 2024 - by **Threat Research Team**

Researchers from Avast have discovered a flaw in the cryptographic schema of the DoNex ransomware and its predecessors. In cooperation with law enforcement organizations, we have been silently providing the decryptor to DoNex ransomware victims since March 2024. The cryptographic weakness was...

New Diamorphine rootkit variant seen undetected in the wild

June 18, 2024 - by **David Álvarez**

Introduction Code reuse is very frequent in malware, especially for those parts of the sample that are complex to develop or hard to write with an essentially different alternative code. By tracking both source code and object code, we efficiently detect new malware and track the evolution of...