The tangle of WiryJMPer’s obfuscation

By Threat Research TeamThreat Research Team

Archived: 2026-04-05 20:55:34 UTC

The story of how we conquered WiryJMPer’s obfuscation begins with a simple binary file posing as an ABBC Coin wallet. We found the file suspicious, as the file size was three-times as big as it should be, and the strings in the file corresponded to other software WinBin2Iso (version 3.16) from SoftwareOK. ABBC Coin (originally Alibaba Coin, not affiliated with Alibaba Group) is an altcoin, one of many blockchain-based cryptocurrencies. WinBin2Iso, on the other hand, is software that converts various binary images of CD/DVD/Blu-ray media into the ISO format. Behavioral analysis revealed that the binary, posing as an ABBC Coin wallet, is a dropper, which we will, from now on, refer to as WiryJMPer. WiryJMPer hides a Netwire payload between two benign binaries.

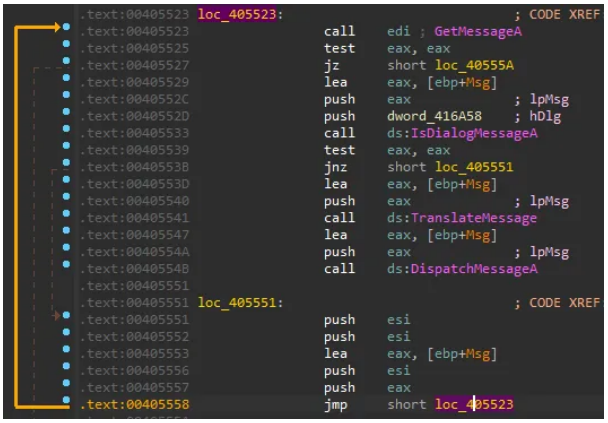
The first stage of the payload innocently appears as a regular WinBin2Iso binary with a suspiciously large *.rsrc* section. The JMP instruction, which is normally part of a loop handling window messages, jumps into the *.rsrc* section where a roller-coaster of control flow begins. This causes an unresponsive WinBin2Iso window to appear briefly before being replaced by a ABBC Coin wallet window. This window is always shown at startup and thus it is a good indicator of infection.

While this functionality isn’t novel in any sense and no sandbox evasion was utilized, the obfuscation was uncommon enough to gain our attention. The combination of control flow obfuscation and low level code abstraction made the analysis of the malware’s workflow rather tedious. This, in combination with the low detection rate on VirusTotal (6 out of 66 as of 7/8/2019), provided us a great excuse to rummage through our toolbox to perform the analysis. Moreover, during the analysis, we found that the obfuscated loader also utilises a (possibly) custom stack-based virtual machine during the RC4 key schedule, which aroused our interest even more.

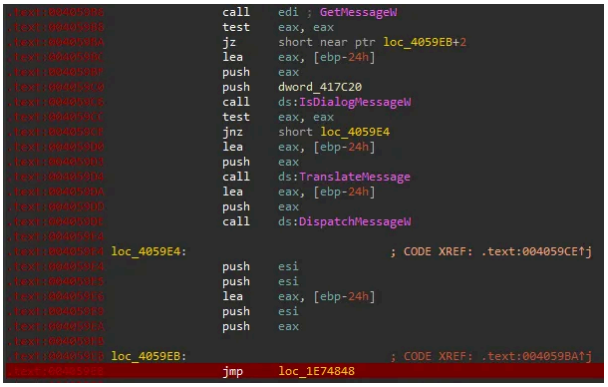
Due to the aforementioned reasons and low overall prevalence, this analysis will focus on the obfuscation itself. Resulting side-effects and the malware’s functionality, will be mostly mentioned as side-notes.

High-level overview

The malware starts in the *WinBin2Iso* binary that has a patched jump. This jump leads to the *.rsrc* section, where a loader is decrypted, loaded into memory and relocations are made.

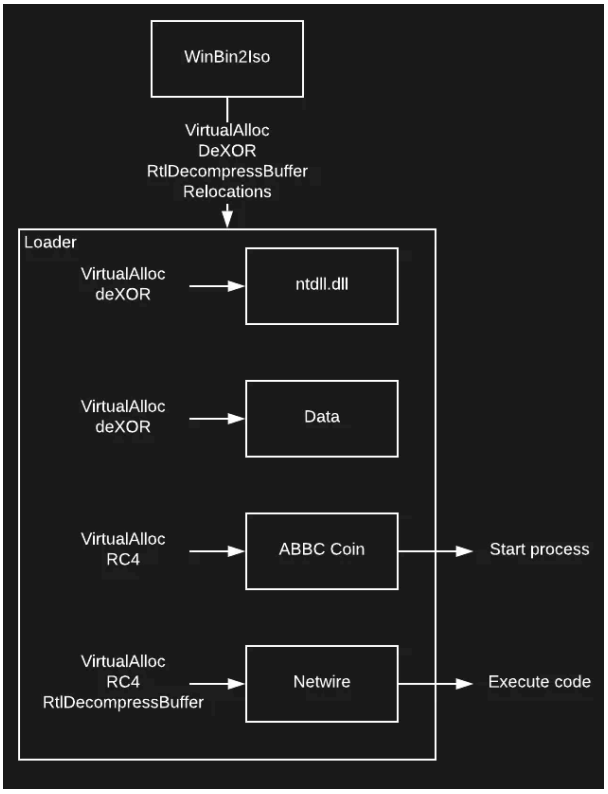


Original WinBin2Iso binary, note that the jump leads back to GetMessageA



Patched WinBin2Iso binary, notice that the jump leads to totally different address range

This loader then handles everything until the control flow is redirected to Netwire. It loads *ntdll.dll* into the memory, decrypts some auxiliary data such as LNK filename or RC4 decryption password. Afterwards, it decrypts Netwire, which is also loaded into the memory, and the “decoy” binary (ABBC Coin wallet in this case), which is saved onto the disk.



High-level overview of WiryJMPer's workflow

Subsequently, the control flow is redirected into Netwire. Netwire is a pretty much standard remote access tool, no significant modifications were made. The Netwire C&C lies at `46.166.160[.]158`, this address was unfortunately unresponsive at the time we performed our analysis.

The loader also tries to achieve persistence by copying the original binary to `%APPDATA%\abbcdriver.exe` and by creating an LNK file, leading to this binary, in the startup folder.

Accessing the loader

The code following the patched jump consists of many small code blocks connected by a network of jumps. This makes the binary rather hard to (statically) analyze without any preprocessing. With the help of an emulator, we are able to reconstruct the call graph and concatenate some of these blocks. While this approach still yields rather unpleasant results, it allowed us to get rid of some dummy instructions and simplify the control flow. Note that this obfuscation also utilises so called opaque predicates, i.e. conditional jumps with one branch that will never be reached. If we wanted to keep the analysis static, we would have to employ e.g. symbolic execution or other heuristics to resolve these jumps. However, for the purpose of our analysis, a simple heuristic, where the code following the conditional branch should lead to "reasonable" instructions, was good enough.

```
01e67c66 jmp 0x1e67c6d
01e67c6d mov edx, 0xcb8ff565
01e67c72 xor ebx, ebx
01e67c74 jmp 0x1e67c79
01e67c79 jmp 0x1e67c80
01e67c80 shr dl, 8
01e67c83 cmp edx, 0xe11b0377
01e67c89 je 0x1e67c9c
01e67c8b jmp 0x1e67c94
01e67c94 not ebx
```

Several concatenated consecutive blocks, note the conditional jump that has predetermined result.

Brief inspection of the sandbox behavioral log reveals two calls of `RtlDecompressBuffer`, the second one decompressing our Netwire payload. Unfortunately, addresses of the compressed buffers lead to addresses allocated during the execution, meaning that we cannot access them directly during static analysis, and thus we will have to dive deeper in order to retrieve the payload.

```
01e670b2 sub edi, 0xfd41
01e670b8 mov esi, 0x79000
01e7279d add esi, dword ptr [esp] ; encrypted buffer
01e727a0 mov ecx, 0xfd41
01e727a5 mov ebx, 0xca81c398
01e727aa push ecx
01e6d44e cld
01e6d44f rep movsb byte ptr es:[edi], byte ptr [esi] ; copy into allocated memory
01e6d451 pop ecx
01e75ad1 mov edx, ecx
01e75ad3 sub edi, ecx
01e75ad5 push edi
```

Finding content of ESI is easy, we just have to dig under ESP. Note that superfluous jumps were removed.

The first *RtlDecompressBuffer* call takes data loaded by rep movsb at `0x01e6d44f`. This instruction loads the data from the binary itself into an allocated memory (*VirtualAlloc*). Since user-space debuggers were probably detected by the sample, we decided to use a kernel debugger that got us to the offset that's under ESI (`0x004f9000`), which leads to data in the *.rsrc* section. A brief check whether this data matches the input to *RtlDecompressBuffer* reveals that the data is encrypted. Fortunately, the decryption loop is located right after the block with `rep movsb`.

```
01e75ad6 xor dword ptr [edi], ebx
01e75ad8 add edi, 4
01e6c97b sub ecx, 4
01e6c97e cmp ecx, 0
01e65549 jg 0x1e75ad6
```

Decryption routine.

Looking at the XORing at `0x1e75ad6` we see that our key is hidden in EBX which is coincidentally loaded just before our `rep movsb` at `0x01e727a5`. Furthermore, we can see that the key is static and thus the whole buffer is XORed by `0xca81c398`. This buffer is then decompressed by the aforementioned *RtlDecompressedBuffer*.

This resolved our first question: where is the compressed loader located? Now two questions remain unresolved: where can we find Netwire payload and where is the "decoy" (ABBC Coin wallet) binary that is launched at the end?

Loader relocation

Before the execution flow moves into the loader from the *.rsrc* section, the relocation has to be made due to the undeterministic nature of *VirtualAlloc* (and the presence of absolute jumps in the loader). The relocation is implemented in the same way as standard relocations in PE files. The relocation table has the following structure:

```
struct reloc_row {
    uint32_t reloc_high;
    uint32_t row_size;
    uint16_t reloc_low[];
}
```

The offset of the instruction to be patched is calculated in the following way:
`reloc_high + reloc_low[i] & 0xfff + allocated_memory_base`

Unsurprisingly, the difference between the original base (`0x10000000`) and the base of the VirtualAlloc-ed memory is added to the residing 32-bit value. The list of `reloc_low` is iterated until the iterator points to the next row of the relocation table.

Accessing the payloads

Since Netwire payload is unpacked using *RtlDecompressBuffer*, it should be easily trackable using the very same tricks we used before. However, since there is other stuff being dropped or extracted, we used *VirtualAlloc* for our breakpoints instead.

The first *VirtualAlloc*, coming after the unpacked loader, prepares space for *ntdll.dll* that is loaded with *NtReadFile*. This is becoming a rather common anti-debugging trick as debuggers mostly do not recognize calls into this manually copied DLL.

Now this is where it gets more interesting. The second *VirtualAlloc* is made for some internal configuration (LNK file name, RC4 key that will be used later on) that is located in the *.rsrc* section (`0x01e632ac`). Again, this binary blob is encrypted by a simple XOR cipher with a key `0x98c381ca` . From now on, we'll start with an assumption that everything is encrypted by the XOR cipher with a hardcoded key, this may simplify the analysis as it is straightforward to recover the key from the plaintext-ciphertext pair.

Our assumption failed immediately on the next payload (`0x004899B6`), its decryption loop contained the following instructions (note that we have removed superfluous jumps):

```
inc esi
dec edi
inc bl
mov dl, byte ptr [ebx + 0x6125015]
add al, dl
push bp
lea esp, [esp + 2]
mov cl, byte ptr [eax + 0x9458248]
mov byte ptr [eax + 0x9458248], cl
sub esp, 4
mov word ptr [esp], bp
lea esp, [esp + 4]
mov byte ptr [eax + 0x9458248], dl
add cl, dl
mov cl, byte ptr [eax + 0x9458248]
xor byte ptr [esi], cl
```

This is a keystream generation loop of the RC4 cipher. This brings us to another problem – we would like to find the key that was used to instantiate the table at `0x09458248` . Unfortunately, the key schedule for the RC4 cipher is obfuscated by a stack-based virtual machine and thus we started to debug again. To illustrate the following process, we will recall the RC4 key schedule algorithm (see Python implementation below).

```
def rc4_key_schedule(S: list, key: bytes):
    for i in range(256):
        S[i] = i
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[j], S[i] = S[i], S[j]
```

RC4 key schedule in Python

This time, we have set a breakpoint on the virtual machine's instruction that is used to write to the table which allows us to recover the state of registers and thus access addresses of `S[i]` and `S[j]`. By subtracting the RC4 table's base address, we obtained the respective indices `i` and `j`. Now, it's trivial to step through the RC4 key schedule and recover bytes of the key from the index `j`. More specifically, we obtain a sequence:

`105, 120, 89, 105, 77, 70, 82, 88, 56, 83, 78, 70, 68, 74, 112, 72, 104, 85, 82, 121, 105, 120, 89, 105, 77, 70, 82, 88, 56, 83, 78, 70, 68, 74, 112, 72, 104, 85, 82, 121, 105, 120, 89, 105, 77, 70, 82, 88, 56, 83, 78, 70, 68, 74, 112, 72, 104, ...`

This sequence obviously has repetitions (highlighted in bold) and these values seem to fall into the printable ASCII range. Using these observations, we recover the RC4 key: `ixYiMFRX8SNFDJpHhURy`. This key was loaded from the buffer, located in the second *VirtualAlloc*-ed memory (containing the internal configuration), into the loader. The decryption of this blob yields the ABBC Coin wallet binary.

Interestingly, the extracted binary matches the real ABBC Coin wallet (version 3.9.1). Later on, this binary will be extracted into the temporary directory under a name matching the following regular expression: `[A-Za-z]{5}\.exe`, and executed right away. We suppose that this is intended to mask the real purpose of the original binary. This payload will not be executed if the original binary is already in the `%APPDATA%` directory. Incidentally, this is the location where the original binary is copied to and thus it won't launch the ABBC Coin wallet if ran at startup (through the aforementioned LNK file).

While we will discuss the virtual machine itself in the Virtual Machine section, we will have a brief look at its part, which is called a dispatcher (actually WiryJMPer has three more similar dispatchers). You may notice one detail:

```
0034b939 movzx edx, byte ptr [esi]
0034b886 mov al, dl
0034b888 inc esi
0034b889 mov eax, edx
0034b7a1 shl eax, 2
0034af66 add eax, 0x85af772e
0034b607 mov eax, dword ptr [eax + 0x7a85300e]
0034af8c add eax, 0x340000
0034b5d2 mov esp, edi
0034b5d4 jmp 0x34b5bc
0034b5bc push eax
0034b5bd ret
```

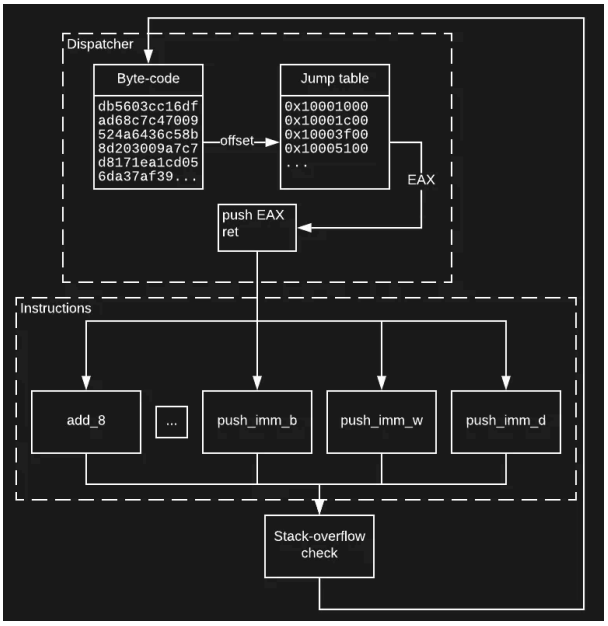
One of the four VM dispatchers that are used during RC4 key-schedule. The correspondence between highlighted parts is not accidental.

Recall that the memory, where the loader resides, had been allocated by *VirtualAlloc* before calling the *RtlDecompressBuffer* and the virtual machine is in the loader. Therefore, bytes at the address `0x0034AF8C` couldn't have been set with this offset from the beginning and had to be patched during runtime as the address range for the loader is not known beforehand. This also indicates the need for relocations.

Now, we just need to find Netwire in the sample. We used the same approach to find *VirtualAlloc* that allocates memory for Netwire, we set a breakpoint on the write and then found a location from which Netwire is copied (offset `0x01e585b6` in *.rsrc* section). Luckily for us, it was encrypted using RC4 with the same key. The decryption yields a UPX packed Netwire, thus concluding the payload extraction and confirming its presence hinted by the sandbox behavioral log.

Virtual machine

During the analysis, we discovered that the RC4 key schedule was implemented in a custom stack-based virtual machine.



Schematic of common stack-based virtual machine

The first part of every virtual machine is a dispatcher. WiryJMPer's virtual machine uses four distinct *switch dispatchers.* In general, *switch dispatchers* jump to the code corresponding to the desired instruction via switch statements or similar constructions. The instruction is translated into a specific address or offset via a jump table, pushed onto the stack and the instruction `ret` is called, although other constructions are also possible. The typical setting is shown on the diagram below, but note that WiryJMPer's virtual machine is more complex as it has more dispatchers and the stack-overflow check does not follow all instructions.

```
0034b939 movzx edx, byte ptr [esi]
0034b886 mov al, dl
0034b888 inc esi
0034b889 mov eax, edx
0034b7a1 shl eax, 2
0034af66 add eax, 0x85af772e
0034b607 mov eax, dword ptr [eax + 0x7a85300e]
0034af8c add eax, 0x340000
0034b5d2 mov esp, edi
0034b5d4 jmp 0x34b5bc
0034b5bc push eax
0034b5bd ret
```

Another dispatcher of the virtual machine.

Since every virtual instruction has to deterministically reach either the dispatcher or exit the virtual machine, we tried to reach these instructions by tracking references leading to these dispatchers. As this virtual machine is stack-based, arguments are passed through stack instead of registers. General purpose registers are mostly used locally, although some registers, such ESI (instruction pointer), ESP (stack frame base) or EBP (stack top), have a global effect on the virtual machine.

Instructions are rather similar to, for instance, the WProtect virtual machine – arithmetic operations, jumps, memory/stack writes and reads, etc. We assumed a typical setting where registers are put onto specific positions in the stack during the initialization and then loaded back from these positions on exit. Due to the amount of instructions (and some duplicities), we will only provide a few examples of these instructions.

```
0034b628 mov ecx, dword ptr [ebp]
0034b62b jmp 0x34b612
0034b612 mov edx, dword ptr ss:[ecx]
0034b615 jmp 0x34b643
0034b643 mov dword ptr [ebp], edx
0034b646 jmp 0x34a1b3
```

Read DWORD from stack

```
0034adfc mov esi, dword ptr [ebp]
0034af17 add ebp, 4
0034af1a jmp 0x34a1dd
0034a1dd mov ebx, esi
0034a1df jmp 0x34a181
0034a181 add esi, dword ptr [ebx]
0034a184 jmp 0x34a1b3
```

Jump

```
0034d461 mov edx, dword ptr [ebp]
0034d464 mov ecx, dword ptr [ebp + 4]
0034d467 jmp 0x34d429
0034d429 add ebp, 8
0034d42c mov dword ptr [edx], ecx
0034d42e jmp 0x34a1b3
```

Write DWORD to memory

Similar files

We found files utilising the same scheme – WinBin2Iso binary patched to unpack Netwire and another binary. For example, the decoy payload led to a different, yet legitimate, installer of Bitcoin Core (version 0.18.0). Others led to the Yoroi wallet, Neon wallet, ZecWallet, DigiByte Core, OWallet, Verge core wallet and others. The common denominator seems to be cryptocurrency wallets.

Conclusion

While the malware's functionality isn't very innovative, it has managed to pass under the radar for some time, probably due to obfuscation and rather low prevalence. The utilised obfuscation was easily overcome by behavioral analysis, nevertheless it served well in obfuscating details of the malware's operation. Rather slow setup of the decoy showing multiple windows with unrelated titles may be suspicious enough for power-users, on the other hand, providing the "decoy" binary might be comforting enough for ordinary users.

Indicators of Compromise (IoC)

- Repository: https://github.com/avast/ioc/tree/master/WiryJMPer
- List of SHA-256: https://github.com/avast/ioc/blob/master/WiryJMPer/samples.sha256

Analyzed sample

File	SHA-256
WiryJMPer	f1963b44a9c887f02f6e9574aea863974be57a033600047b8e0911f9dbcb9914
ABBC Coin Wallet	7477159797a7f06e3c153662bfef624d056e64b552f455fe53e80f0afb0a1860
Netwire payload	6daa1ff03fdbbb58b1f41d2f7dc550ee97fc5b957252b7f1703c81c50b3d406f

Analyzed sample

Netwire payload C&C: `46.166.160[.]158`

Similar samples

SHA-256
6e1cfde5278d03c6df204d845d165673df89cfd047f4eda97816ee351115a652
4b7bd8581b85bb33d4748aaeda6a3e5ec8f930751688ffb6854522411f3ad275
81740ad6a3f0e5c1698132524e0d4b23b4f4773761bca68fdaef33748ef299e3
880de7e64c0678a38ef6964b6ff2f48e426449426b58a516556285421c223374
125cf6b01deb86df16e0961021a57b28177b8efedc6bf4f617bef940cf4b9d74
04a92a7e171b583c40cee9d2760b20fa8324e45f3938f7d41f48065829103ebd
4a3d3e85d09074ed1e1de5e48c97c4e42fbcb3cfb44b213c0224ffb191dcd1c2
0631ace562e077814c7788b9fe10c865579a29cf180654658f30ab38387a13e3
d1457c238b99ca8904693551f92310acae561c68c20a8caafe3391d927d7618e
ea855c2b53419dcd81e677520d4e55d41cb5ce2933f550edd6520cce15da93fc

Similar samples

Threat Research Team

Threat Research Team

A group of elite researchers who like to stay under the radar.

Source: https://decoded.avast.io/adolfstreda/the-tangle-of-wiryjmpers-obfuscation/