

KONNI evolves into stealthier RAT

By Mark Stockley

Published: 2022-01-25 · Archived: 2026-04-05 14:43:41 UTC

This blog post was authored by Roberto Santos

KONNI is a Remote Administration Tool that has been used for at least 8 years. The North Korean threat actor that is using this piece of malware has been identified [under the Kimsuky umbrella](#). This group has been very busy, attacking political institutions located in Russia and South Korea. The last known attack where KONNI Rat was used was described [here](#).

We found that KONNI Rat is being actively developed, and new samples are now including significant updates. In this blog post, we will cover some of the major changes and explain why the security community should keep a close eye on it.

Article continues below this ad.

Simplified Attack Chain

The attack usually starts leveraging a malicious Office document. When this document is opened by the victim, a multistage attack is started, involving various steps. But these steps are just the way that the attackers manage to accomplish tasks to elevate privileges, evade detection and deploy required files. As we described in a [previous blog post](#), the attack chain could be summarized in the following diagram:



The attack usually starts leveraging a malicious Office document. When this document is opened by the victim, a multistage attack is started, involving various steps. But these steps are just the way that the attackers manage to accomplish tasks to elevate privileges, evade detection and deploy required files.

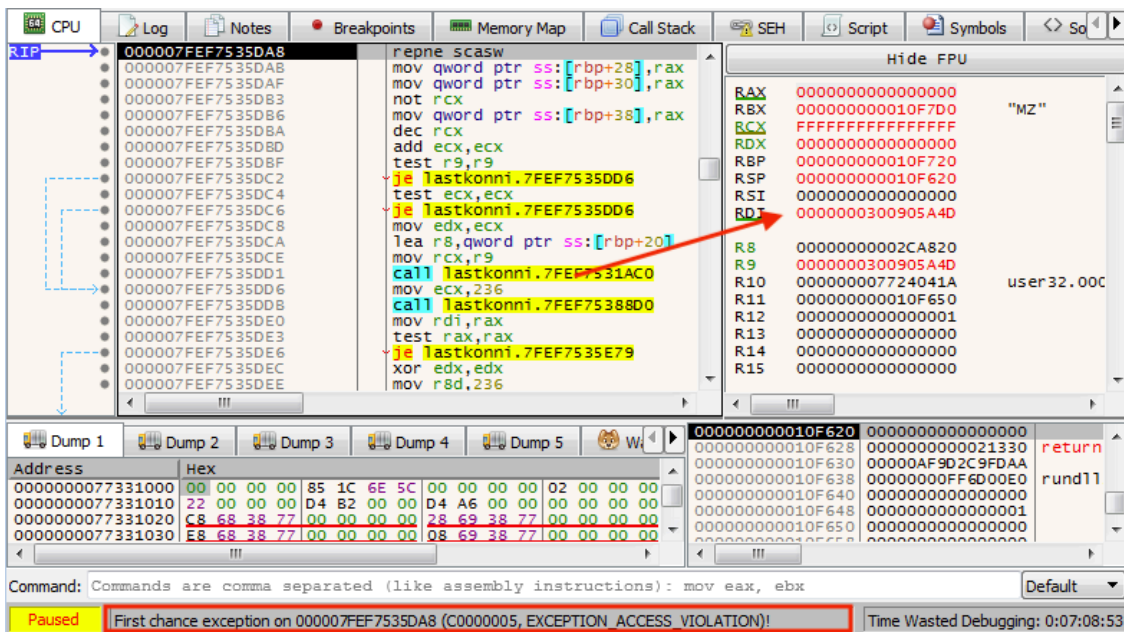
The final goal of the attack is installing what is called KONNI Rat, which is a .dll file supported by an .ini file. In a nutshell, the .dll file contains the functionality of the RAT, and the .ini file contains the address of the first C&C server. KONNI Rat’s general behavior remains almost the same as previous versions, but there are changes we will cover below.

Rundll no longer supported

In previous KONNI Rat samples there were two branches. One handles if the malware was launched using a Windows service, and the other handles the execution through rundll. The next image shows these two old branches, with the strings svchost.exe and rundll32.exe visible:

```
result = GetModuleFileNameW(0i64, &v4, 260i64);
if ( (_DWORD)result )
{
    if ( StrStrW(&v4, L"svchost.exe") )
    {
        qword_180006CD8 = RegisterServiceCtrlHandlerW(*serviceName, serviceCallbackFunction);
        if ( !(unsigned int)sub_180004318(2, 3000) && qword_180006CD8 )
            sub_180004318(1, 0);
        result = sub_1800043AC();
        if ( qword_180006CD8 )
            result = sub_180004318(1, 0);
    }
    else
    {
        result = StrStrW(&v4, L"rundll32.exe");
        if ( result )
            result = sub_18000424C();
    }
}
```

However, new samples will not show these strings. In fact, **rundll is no longer a valid way to execute the sample**. Instead, when an execution attempt occurs using rundll, an exception is thrown in the early stages.



In early stages of our analysis, we thought that they were using the classic process name check, or any other usual technique. The reality is far simpler and brilliant; the actual export just implements the SvcMain prototype so the program will break at some point when accessing one of the arguments.

In the previous image we see the state of the machine at the moment that this exception is thrown. RDI at that point should contain a pointer to the service name. The exception happens because the Service Main function meets one prototype and rundll32 will expect another different prototype:

```
VOID WINAPI SvcMain( DWORD dwArgc, LPTSTR *lpszArgv )
```

```
VOID WINAPI runnableExport(HWND hwnd, HINSTANCE hinst, LPSTR lpszCmdLine, int nCmdShow)
```

Basically, at some point of the execution, hinst will be treated as lpszArgv, causing the exception. But why did the attackers delete that functionality? There are multiple benefits.

First of all, we have not seen any recent attack that used rundll. In fact, the only way that the attackers launched KONNI Rat in recent campaigns involves registering a Windows service. So the rundll32 branch wasn't being used in real world attacks.

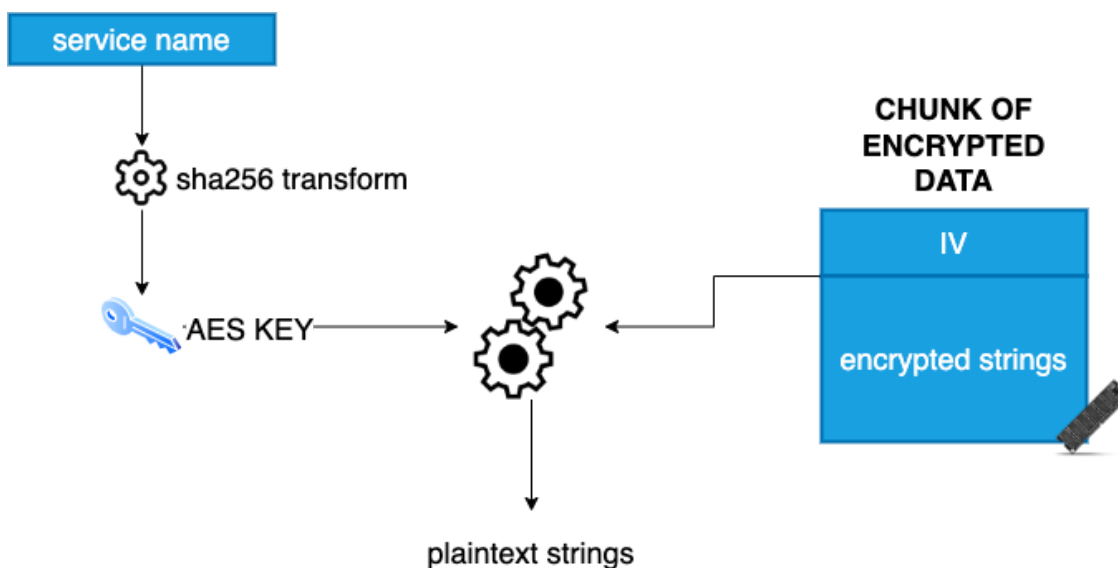
But there is another big reason in how sandboxes will fail in collecting the real behavior of the sample, as it just cannot execute that way.

Strings are now protected using AES

Multiple malware families protect their strings in order to defeat most basic string analysis. KONNI wasn't an exception, and also used this technique. Old samples were using base64 for obfuscation means. Also, they were using a custom alphabet. This custom alphabet was changed from time to time in order to make the decoding task more difficult:

```
; Segment permissions: Read/Write
_data      segment para public 'DATA' use64
          assume cs:_data
          ;org 180006000h
customB64Alphabet db 'TYXpzsuDoFSMmd70k%BGKHQb6Z5ygVcRiP8A04aWh1f2t1-ExvqrwUj93CL=JNIne'
                                     ; DATA XREF: posEncodeBuffer+ADf0
                                     ; decodeStr+27f0 ...
          db 0
          align 10h
; CHAR string1[]
string1    db '6jU0oXNAoSf%qYY%zkiBz1pHHvpyjCryjv4oXNjozdEZuHk6QV4oXNwosFs%UNzH'
                                     ; DATA XREF: sub_18000424C+19f0
                                     ; sub_18000424C+2Bf0
          db 'wNB%XTEZXTjdGTxmBTEZ8ee',0 ; cmd /c REG ADD HKCU\Console /v CodePage /t REG_DWORD /d 65001 /f
          align 10h
; CHAR string2[]
string2    db '5jHqyaHtmro-Zuvt',0 ; DATA XREF: posIATBuild+2Ff0
                                     ; posIATBuild+41f0
                                     ; kernel32.dll
          align 8
; CHAR string3[]
string3    db 'kQ%j6bYlmro-Zuvt',0 ; DATA XREF: posIATBuild+9E9f0
                                     ; posIATBuild+9FBf0
                                     ; Advapi32.dll
```

Now, the attackers made a major change in that regard by protecting the strings using AES encryption. The algorithm followed by new Konni RAT samples could be represented as follows:

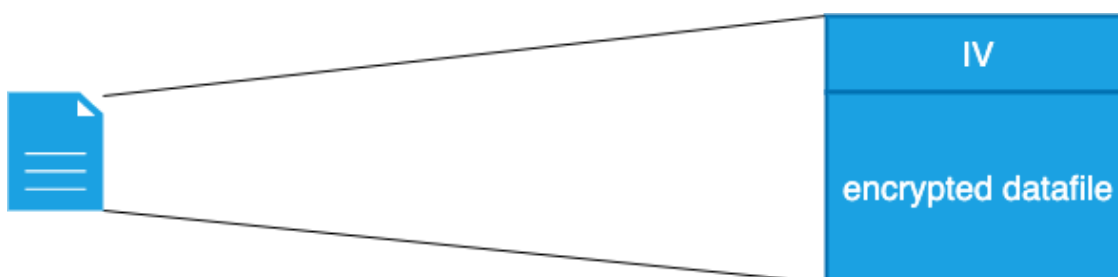


The reason behind that change is clear. As the key used for decryption is the service name, samples run by different service names will not work properly. Moreover, having only the sample **without knowing the service name becomes useless**, as these strings contain core information about the sample behavior.

Files are also protected using AES

KONNI Rat makes use of various support files when it is executed. One of these files is the .ini file, which contains the primary C&C server, but there are others like the .dat file that is supposed to be dropped eventually, and temporal files that are used to send some basic information about the computer.

Our tests reveal that all of these files are dropped and protected using AES. Cleverly, they reused the algorithm used for string protection, making the file layout identical to the protected strings layout, as they appear in raw memory:



As can be seen from the diagram, the file itself contains the IV and the encrypted data. The key used is extracted from its original filename. In some cases, the names match with the service name, so the keys used in the .ini and the .dat files are the result of applying a SHA256 to the service name as well.

Also, files sent to the C&C server are protected using AES. The IV is generated using a QueryPerformanceCounter API CALL. Filenames are generated concatenating 2 letters that represent the data with the current timestamp, followed by the extension. Furthermore, they will use this newly generated name as AES key, so they send this name through the request to the C&C server.

```
debug088:000000000256620 a7e4512a60722Co_3 db '-----7e4512a60722',0Dh,0Ah
debug088:000000000256620 db 'Content-Disposition: form-data; name="fileToUpload"; filename="ff'
debug088:000000000256620 db ' 01-13 17-08-38.txt"',0Dh,0Ah
debug088:000000000256620 db 'Content-Type: application/octet-stream',0Dh,0Ah
debug088:000000000256620 db 0Dh,0Ah,0
```

In that regard, as the filename is generated automatically using the timestamp, identical files will produce different request contents, as they were encrypted using that filename. **Network signatures could also fail** to detect the malicious activity, due to that.

Other obfuscation techniques

As we found some samples that were protected *just* by the means that we described before, we also have found others that were making use of an unidentified packer (*UPDATE: There is a strong correlation between this packer and VMProtect v3, as you would see in the following paragraphs. Later deeper analysis tasks suggest that this is probably the packer that was used in this occasion*). We would like to share some of our notes regarding that packer, as others could find it useful in identification and attribution tasks.

Contiguous instruction obfuscation

The flow of the obfuscated program will make use of series of push-call pairs of instructions, where the pushed values will indicate the actions that the program will take. An image can better explain that:

<pre>68 A9 F1 7C 01 E8 38 62 F3 FF 97 68 33 59 34 A9 E8 C9 3A FE FF 6A 68 AA 79 B6 33 E8 BC 91 F8 FF 12 95 49 88 28 4E 68 50 78 8A 69 E8 EE 90 F7 FF 7E 68 57 58 BC B3 E8 A1 91 F8 FF 86</pre>	<pre>push 17CF1A9h call loc_1801D6F84 xchg eax, edi push 0FFFFFFFA9345933h call loc_180284820 ; ----- ; db 6Ah ; j ; ----- ; push 33B679AAh call loc_180229F1E adc dl, [rbp+4E288849h] ; DATA XREF: .qwdfr0:00000001802A41DC↓ ; .qwdfr0:00000001802A41E8↓ push 698A7850h call loc_180219E60 ; ----- ; db 7Eh ; ~ ; ----- ; push 0FFFFFFFB3BC5857h call loc_180229F1E ; ----- ; db 86h ; †</pre>
--	---

In particular, we find it interesting that the attackers have placed random bytes between these pairs. This silly trick causes wrong code interpretation for decompilers that will assume that bytes after the push instruction are part of the next instruction. The image below shows how IDA fails in analyzing the code:

Obfuscated program flow

The used packer will obfuscate the original program flow. This is accomplished in various steps. The first required step is to find the Image Base value, placed in a fixed location and the RIP (Instruction Pointer) value.

<pre>00000001801E9166 48:B8 0000000000000000 mov rax,0 00000001801E9170 80D6 6D adc dh,6D 00000001801E9173 D2F3 shl bl,cl 00000001801E9175 4C:2BD8 sub r11,rax 00000001801E9178 F6D8 neg bl 00000001801E917A 48:0E9F00 movsx rdx,ax 38: 00000001801E917E 48:8D1D F9FFFFFF lea rbx,qword ptr ds:[1801E917E] 00000001801E9185 0FB7D0 movzx edx,ax 00000001801E9188 49:81EA 04000000 sub r10,4 31: 00000001801E918F 80DA A3 sbb dl,A3</pre>	<pre>RAX 0000000000000000 RBX 00000001801E917E RCX 00007937D5930000 RDX 0000000000000000 RBP 0000000000298C60 RSP 00000000004EF920 RSI 0000000000298500 RDI 00000000004EFAA8</pre>
--	--

Once the packer knows these two values, it will start jumping from one place to another, making analysis harder. For that, it will store in some register value of the next address to jump in registers. The value of these registers is calculated right after the jmp instruction, using structures like POP [reg] – JMP [reg] or ADD [reg1, reg2] – JMP [reg1]. Note that decompilers will fail in displaying the real flow, as the jumping address is determined by a somehow undefined register.



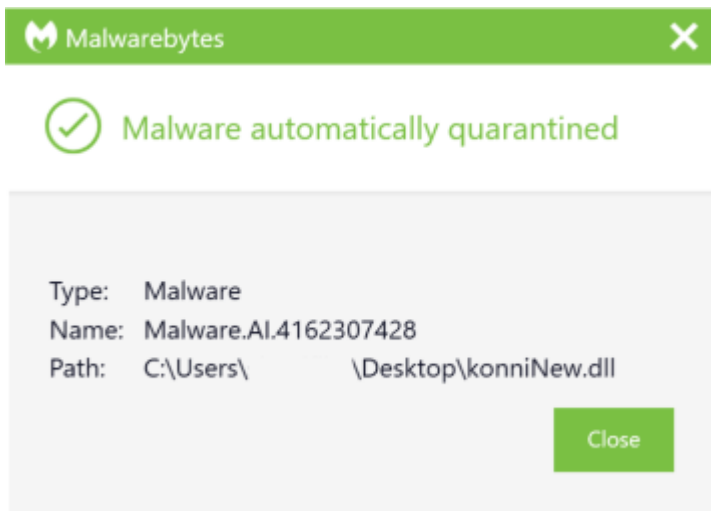
The combination of these simple techniques ends in the packer being now in control of the flow, but statically the decompiler cannot represent the path that the code will follow. Finally, the packer will execute a big amount of junk instructions and eventually will execute the real interesting code. For instance, the original code will take no more than 20 instructions between GetProcAddress calls in IAT building tasks. but the packed code executes more than 30,000 instructions.

According to our threat intel data, most recent attacks are not making use of that packer anymore.

Conclusion

As we have seen, KONNI Rat is far from being abandoned. The authors are constantly making code improvements. In our point of view, their efforts are aimed at breaking the typical flow recorded by sandboxes and making detection harder, especially via regular signatures as critical parts of the executable are now encrypted.

Malwarebytes users are protected against this attack.



IOCs

A3CD08AFD7317D1619FBA83C109F268B4B60429B4EB7C97FC274F92FF4FE17A2
F702DFDDBC5B4F1D5A5A9DB0A2C013900D30515E69A09420A7C3F6EAAAC901B12

Source: <https://blog.malwarebytes.com/threat-intelligence/2022/01/konni-evolves-into-stealthier-rat/>