BYOS – Bundle Your Own Stealer

By etal

Published: 2023-07-19 · Archived: 2026-05-07 02:08:50 UTC

Highlights:

- ***Check Point Research (CPR) provides an in-depth analysis of the new malware strain dubbed BundleBot spreading under the radar***.
- ***BundleBot is abusing the dotnet bundle (single-file), self-contained format that results in very low or no static detection at all***.
- ***Commonly distributed via Facebook Ads and compromised accounts leading to websites masquerading as regular program utilities, AI tools, and games***.
- ***CPR introduces several techniques that were approved to be effective for reverse engineering the dotnet bundle (single-file), self-contained format***.

Introduction

During the past few months, we have been monitoring a new unknown stealer/bot, we dubbed **BundleBot,** spreading under the radar and abusing dotnet **bundle** (single-file), self-contained format. This format of dotnet compilation has been supported for about four years, from .net core 3.0+ to dotnet8+, and there are already some known malware families abusing it (e.g., Ducktail).

The BundleBot using this specific dotnet format is different in the sense of its infection chain (more sophisticated), mostly abusing Facebook Ads and compromised accounts that lead to phishing websites masquerading as regular programs, AI tools, and games. Leveraging the dotnet bundle (single-file), self-contained format, multi-stage infection, and custom obfuscation resulted in an effective way to stay under the radar with very low or no static detection at all.

The dotnet bundle (single-file), self-contained format generally results in a very large binary bloated with the whole dotnet runtime. Furthermore, analyzing and debugging such a file could result in some problems, especially if such a file is affected by some obfuscation/protection.

The main subject of this research is an in-depth analysis of the BundleBot, its typical vector of infection, and an explanation of the dotnet bundle (single-file), self-contained format, focusing on general problems during the analysis of such file format.

Background & Key Findings

Since the release of .NET Core 3.0 (2019), it has been possible to deploy .NET assemblies as a single binary. These files are executables that do not contain a traditional .NET metadata header and run natively on the underlying operating system via a platform-specific application host bootstrapper.

Dotnet bundle (single-file), self-contained format is a compilation form that enables to produce a single executable binary that does not require to have a specific dotnet runtime version preinstalled on the OS. The executable is actually a native hosting binary that contains whole dotnet runtime, assemblies, and other dependencies in its overlay (so it is large in size – dozens of MBs). The native hosting binary is responsible for extracting (on-execute) all from overlay, loading the dotnet runtime and assemblies, preparing everything, and transferring execution to the Entry Point of a .NET module.

When it comes to extracting the assemblies from the overlay (on-execute), we can deal with different routines depending on the targeted dotnet version used to compile the specific application. The difference among dotnet versions is that before dotnet5+ (.NET Core 3.0+), by **default**, all assemblies were extracted to the **disk** (`temp` directory) and loaded into the process memory.

On the other hand, from the dotnet5+ version, all assemblies from the overlay are extracted and loaded directly into the process memory (no files dropped on disk – only native libraries if used and not deployed separately). From dotnet5+, the

extraction could be specified during the compilation, but the default setting is to extract directly into memory.

Despite the fact we are still dealing with dotnet-related applications, the above-mentioned description of this specific file format makes it clear that one would need to use a different toolset and techniques to analyze it properly.

We detected the BundleBot abusing the dotnet bundle (single-file), self-contained format as the last stage of infection that was related to several campaigns, very likely initiated by the same threat actor.

In all cases we spotted in the wild, the initial vector of infection was via Facebook Ads or compromised accounts that led to websites masquerading as regular program utilities, AI tools, and games (e.g., **Google AI**, **PDF Reader**, **Canva**, **Chaturbate**, **Smart Miner**, **Super Mario 3D World**). As one of the capabilities of the BundleBot is stealing Facebook account information, those campaigns could be considered self-feeding, where stolen information is further used to spread the malware via newly compromised accounts.

Vector of infection

As we mentioned earlier, the typical initial vector of infection points to Facebook Ads or compromised accounts leading to websites masquerading as regular program utilities. Still, we can not fully exclude other possible delivery methods as we could not obtain links of origin for all detected samples via their relevant tracking information.

Once the victim is tricked into downloading the fake program utility from the phishing website, the first stage downloader is delivered in the form of a "RAR" archive. Those downloader stages are usually on hosting services like **Dropbox** or **Google Drive**.

The downloaded "RAR" archive contains the first stage downloader in a self-contained dotnet bundle (single-file) format. Right upon execution of this first stage, the second stage is downloaded in the form of a password-protected "ZIP" archive, usually from a hosting service such as **Google Drive**. The password for the second stage is hardcoded in the downloader, usually in an encoded form.

The main part of the password-protected "ZIP" archive that gets extracted and executed is the BundleBot which abuses the dotnet bundle (single-file), self-contained format with a combination of custom obfuscation.

An example of a detailed infection chain related to the fake utility "**Google AI**" that pretends to be a marketing tool using Google AI Bard could be seen below:

1. Facebook ads or Facebook posts from compromised accounts leading to `https://marketingaigg[.]com/`

Figure 1: Facebook post from compromised account leading to the phishing website

2. Phishing website `https://marketingaigg[.]com/` masquerading as a marketing tool using Google Bard AI leads to the download page `https://googlebardai[.]wiki/Googleai`

Figure 2: Phishing website that leads to the downloader stage

3. The URL `https://googlebardai[.]wiki/Googleai` is serving "RAR" archive `Google_AI.rar` (SHA-256: "dfa9f39ab29405475e3d110d9ac0cc21885760d07716595104db5e9e055c92a6") from the **Dropbox** hosting service
4. `Google_AI.rar` contains `GoogleAI.exe` (SHA-256: "5ac212ca8a5516e376e0af83788e2197690ba73c6b6bda3b646a22f0af94bf59"), dotnet bundle (single-file), self-contained application
5. `GoogleAI.exe` contains `GoogleAI.dll` dotnet module that serves as a downloader (downloads password-protected "ZIP" archive `ADSNEW-1.0.0.3.zip` from `https://drive.google[.]com/uc?id=1-mC5c7o_B1VuS6dbQeDAAqLuPbfAV58O&export=download&confirm=t`, password=**alex14206985alexjyjyjj)**
6. The extracted content of `ADSNEW-1.0.0.3.zip` (SHA-256: "303c6d0cea77ae6343dda76ceabaefdd03cc80bd6e041d2b931e7f6d59ca3ef6") contains `RiotClientServices.exe`, dotnet bundle (single-file), self-contained application

7. The `RiotClientServices.exe` served and executed as the last stage contains two malicious dotnet modules `RiotClientServices.dll` – BundleBot, `LiarySharing.dll` – C2 packet data serializer

Self-Contained Dotnet Bundle – analysis and debugging problems

When we need to analyze a self-contained dotnet bundle (single-file) binary, we can immediately encounter several problems.

The first one is that we need to extract somehow all binaries that are a part of the overlay of the bundle described earlier. This extraction will help us investigate each file statically, as we would do when dealing with ordinary dotnet assemblies. Despite the fact it is not so known, there are already existing solutions that understand the dotnet bundle format enough to help us with the extraction. We will mention both GUI-based tools and library to do it in a programmatic way. Notably, for now, the extraction of the dotnet bundle file is not supported in [dnSpy/dnSpyEx](https://research.checkpoint.com/2023/byos-bundle-your-own-stealer/).

Among the most reliable GUI-based tools that can help with the extractions are:

- ILSpy – open-source .NET assembly browser and decompiler
- dotPeek – free .NET decompiler and assembly Browser

Extraction of dotnet bundle in ILSpy:

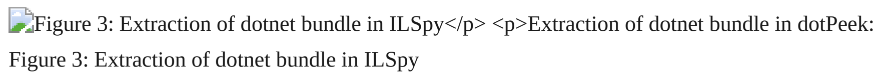

Figure 3: Extraction of dotnet bundle in ILSpy

Extraction of dotnet bundle in dotPeek:

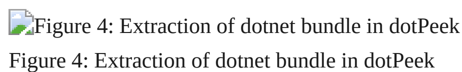

Figure 4: Extraction of dotnet bundle in dotPeek

As we already pointed out, the extraction of dotnet bundle files could also be done programmatically. Such a way could be very handy when we are processing a larger set of files.

One of the most suitable solutions for this purpose is to use AsmResolver. AsmResolver is a Portable Executable (PE) inspection library that is able to read, modify and write executable files. This includes .NET modules as well as native images. The library exposes high-level representations of the PE while still allowing the user to access low-level structures. What is even more crucial is that AsmResolver understands the bundle file format so we can use it to automate the extraction.

Such a code example extracting the bundle file content using AsmResolver and PowerShell can be seen below.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
[Reflection.Assembly]::LoadFrom("C:\AsmResolver\AsmResolver.DotNet.dll") | Out-Null

$extractionPath = "C:\Extracted\"

$manifest = [AsmResolver.DotNet.Bundles.BundleManifest]::FromFile("C:\RiotClientServices.exe")

foreach($file in $manifest.Files)

{

$fileInfo = [IO.FileInfo]::new($extractionPath + $file.RelativePath)

$fileInfo.Directory.Create()
```

[IO.File]::WriteAllBytes($fileInfo.FullName, $file.getData($true))

}

[Reflection.Assembly]::LoadFrom("C:\AsmResolver\AsmResolver.DotNet.dll") | Out-Null $extractionPath = "C:\Extracted\" $manifest = [AsmResolver.DotNet.Bundles.BundleManifest]::FromFile("C:\RiotClientServices.exe") foreach($file in $manifest.Files) { $fileInfo = [IO.FileInfo]::new($extractionPath + $file.RelativePath) $fileInfo.Directory.Create() [IO.File]::WriteAllBytes($fileInfo.FullName, $file.getData($true)) }

```
[Reflection.Assembly]::LoadFrom("C:\AsmResolver\AsmResolver.DotNet.dll") | Out-Null
$extractionPath = "C:\Extracted\"
$manifest = [AsmResolver.DotNet.Bundles.BundleManifest]::FromFile("C:\RiotClientServices.exe")

foreach($file in $manifest.Files)
{
    $fileInfo = [IO.FileInfo]::new($extractionPath + $file.RelativePath)
    $fileInfo.Directory.Create()
    [IO.File]::WriteAllBytes($fileInfo.FullName, $file.getData($true))
}
```

Now, when we are in a state we successfully extracted the whole content of the dotnet bundle file, we can use any tool that we would normally use to inspect the ordinary dotnet assemblies, like dnSpyEx. This will allow us to investigate each dotnet assembly statically.

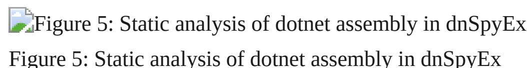
Figure 5: Static analysis of dotnet assembly in dnSpyEx

As dotnet assemblies, especially the malicious ones, are usually quite sophisticated and very often affected by some obfuscation or protection, most researchers prefer to combine both the static and dynamic analysis approach. Regarding the dynamic approach, we are getting closer to the second problem with a self-contained dotnet bundle (single-file) binary – **debugging**.

It is always welcome to debug dotnet assemblies in managed debuggers such as dnSpyEx. The debugging in dnSpyEx was not fully supported for self-contained dotnet bundle binary, and if one tried to debug such files, it could result in a similar-like exception shown below.

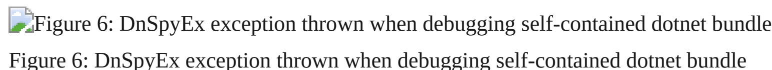
Figure 6: DnSpyEx exception thrown when debugging self-contained dotnet bundle

Fortunately, the recently released version of dnSpyEx (v6.4.0) improved the debugging of such files, so we should not get this kind of exception anymore, and debugging could proceed without any problems.

Despite the fact we can debug self-contained dotnet bundle files in the latest release of dnSpyEx (v6.4.0), it can not solve the problem of dealing with obfuscated dotnet assemblies that are a part of the dotnet bundle and would be very likely the main subject of our interest. In such cases, it is always good to be able to extract the whole content of the bundle file, deobfuscate assemblies and proceed with the below-mentioned guide to preserve the debugging.

When the dotnet binary is compiled as a self-contained bundle, it simply means that whole dependencies (especially dotnet runtime) are a part of the produced application, and such an application is configured to use them (via its configuration files). Those configuration files are the main problem affecting the debugging after extraction of the bundle and deobfuscation of each protected assembly.

To overcome this, we can actually convert the self-contained dotnet bundle file into a non-self-contained, non-single-file .NET program. This way converted program will be tricked into using dotnet runtime, which is a part of OS, so we must be sure to have it installed.

The conversion could be accomplished with the following steps:

- **Extracting** the content of the dotnet bundle file (as described earlier).

- **Finding out** the dotnet runtime version to be installed in OS and **installing it**. To quickly find out the information about specific version of dotnet runtime our .NET application depends on and we need to install, we can locate and check configuration files `*[appname].runtimeconfig.json*` and `*[appname].deps.json*` which should be inside the previously extracted content.

In the example below, we can clearly see that we need to install .NET Runtime 5.0.17, x86.

Figure 7: Configuration files

Figure 8: Required version of dotnet runtime to be installed (Microsoft)

- **Modifying** the content of configuration files `*[appname].runtimeconfig.json*` and `*[appname].deps.json*`. By modification of these files, we are converting the application into a non-self-contained, non-single-file .NET program, and we will force it to use the installed version of dotnet runtime.

Modification of `*[appname].runtimeconfig.json*` by changing the "**includedFrameworks**" string to "**frameworks**".

Figure 9: Modification of "[appname].runtimeconfig.json"

Modification of `*[appname].deps.json*` by removing "**runtimepack**" entries from "**libraries**".

Figure 10: Modification of "[appname].deps.json"

- **Running** and **Debugging**. The self-contained dotnet bundle application could have dependencies on native libraries that could be a part of the bundle (so we would have them already extracted from the content), or they can be provided separately alongside the bundle executable. We can quickly find out if the application has such dependencies (defined in `*[appname].deps.json*`) by checking the configuration file or by running it, as shown below.

Figure 11: Dependency-related error when running the extracted bundle application

To overcome this, simply copy all dependencies that should be alongside of the bundle application to the location of previously extracted content of the bundle. Now the debugging should work like for ordinary .NET applications using the dotnet runtime that is installed in OS.

Figure 12: Debugging converted non-self-contained, non-single-file .NET app in dnSpyEx

The above-mentioned guide is not necessary in cases where we are not dealing with obfuscated/protected dotnet assemblies that are a part of the dotnet bundle, as using the latest release of dnSpyEx (v6.4.0) can debug them directly. Still, the workaround above is needed when we deal with obfuscated assemblies and prefer to debug them in deobfuscated form.

Noteworthy is that we described a general approach to converting a self-contained dotnet bundle file to ordinary dotnet assembly, which depends on the presence of an appropriate version of dotnet runtime being preinstalled on the targeted OS. This approach should work across different OS platforms (Windows, Linux, macOS)

Encouraged with the knowledge of how to extract the content of a self-contained dotnet bundle file and how to debug it, we can finally move forward to analysis.

Technical Analysis: Highlights

- Self-contained dotnet bundle format to harden the analysis and static AV detection
- Affected by simple but effective custom-made obfuscation
- Abusing password-protected archives to deliver the last stage
- The last stage is a new stealer/bot – BundleBot

- Custom homebrew packet data serialization for C2 communication

Technical Analysis – Downloader

For the analysis of the downloader stage, sample `GoogleAI.exe`, SHA-256: "5ac212ca8a5516e376e0af83788e2197690ba73c6b6bda3b646a22f0af94bf59" was used.

This sample is a 32-bit self-contained dotnet bundle application (.NET Core 3.0.3), originally a part of the RAR archive. After extraction of this bundle, the main module `GoogleAI.dll` is a downloader, affected by simple custom obfuscation – only strings and names (non-meaningful Thai text).

Figure 13: Downloader affected by simple custom obfuscation

PDB path of the downloader: `D:\BOT\RAT\Rat Ver 4.0\HashCode\Bot ADS-Server 4\ClientDowload-FB\ClientDowload\obj\Debug\netcoreapp3.0\win-x86\GoogleAI.pdb`.

After deobfuscation, the main functionality resides in the function named as `ProcessMain`.

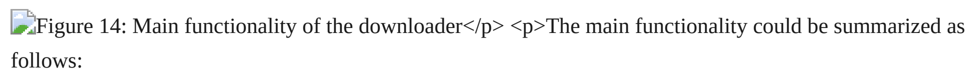
Figure 14: Main functionality of the downloader</p> <p>The main functionality could be summarized as follows:
Figure 14: Main functionality of the downloader.

The main functionality could be summarized as follows:

- Single instance check
- Downloading password-protected ZIP archive saved with a random name and ".rar" extension
- Archive downloaded from: `https://drive.google[.]com/uc?id=1-mC5c7o_B1VuS6dbQeDAAqLuPbfAV58O&export=download&confirm=t`
- Setting the file attribute of the downloaded archive as "Hidden"
- Extracting the content of the downloaded archive to a newly created folder `C:\Users\User\Documents\{random}`, password: `alex14206985alexjyjyjj`
- Setting the attribute of the newly created folder and all ".exe" files inside as "Hidden"
- Trying to execute all ".exe" files
- Deleting the downloaded archive

The BundleBot, in the form of a self-contained dotnet bundle file, is the main part of the downloaded password-protected archive and gets executed by the downloader. Noteworthy, all analyzed downloaders contained the same hardcoded password `alex14206985alexjyjyjj` (either in clear-text or base64 encoded) to extract the next stage.

Technical Analysis – BundleBot

For the analysis of the BundleBot stage, sample `RiotClientServices.exe`, SHA-256: "6552a05a4ea87494e80d0654f872f980cf19e46b4a99d5084f9ec3938a20db91" was used.

This sample is a 32-bit self-contained dotnet bundle application (.NET 5.0.17), originally a part of the password-protected ZIP archive. After extraction of this bundle, its main malicious components are the main module `RiotClientServices.dll` and a library `LirarySharing.dll`.

The assembly `RiotClientServices.dll` is a custom, new stealer/bot that uses the library `LirarySharing.dll` to process and serialize the packet data that are being sent to C2 as a part of the bot communication.

These binaries are affected by similar custom-made obfuscation that mainly focuses on name obfuscation and bloating those dotnet modules with a lot of junk code. Such an obfuscation will result in an overwhelming number of methods and classes that will make the analysis much harder and require creating a custom deobfuscator to simplify the analysis process.

Before the deobfuscation, the size of the `RiotClientServices.dll` is ≈ 11MB containing 26742 methods and 902 classes. In the case of `LirarySharing.dll`, the obfuscation resulted in a binary size ≈ 10MB with 32462 methods and 9473 classes.

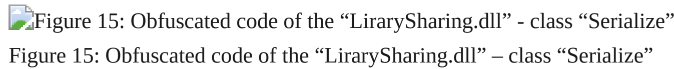
Figure 15: Obfuscated code of the "LirarySharing.dll" – class "Serialize"

One could get easily lost in such a mess. Because of that, we quickly put together a simple deobfuscator that works for all binaries that are affected by similar-based custom obfuscation. This deobfuscator uses AsmResolver and PowerShell to mainly clean the junk code and still preserves the debugging opportunity.

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

```
[Reflection.Assembly]::LoadFrom("C:\AsmResolver\AsmResolver.DotNet.dll") | Out-Null

$obfuscated = "C:\RiotClientServices.dll"

$moduleDef = [AsmResolver.DotNet.ModuleDefinition]::FromFile($obfuscated)

# Removing junk methods

foreach($type in $moduleDef.GetAllTypes())

{

foreach($method in [array]$type.Methods.Where{$_.HasMethodBody})

{

if(($method.MethodBody.Instructions.Where{$_.Opcode.Mnemonic -like "call" -and

$_.Operand.FullName -like "*System.Console::WriteLine*"}).count -eq 5)

{

$type.Methods.Remove($method) | Out-Null

}

}

}

# Removing junk NestedTypes

foreach($type in [array]$moduleDef.GetAllTypes().Where{$_.IsNested -and $_.Methods.Count -eq 1 -and

$_.Methods[0].IsConstructor -and $_.Methods[0].MethodBody.Instructions.Count -eq 4})

{

foreach($topType in $moduleDef.TopLevelTypes.Where{$type -in $_.NestedTypes})

{

$topType.NestedTypes.Remove($type) | Out-Null

}

}

# Removing junk TopLevelTypes
```

foreach($topType in [array]$moduleDef.TopLevelTypes.Where{$_.BaseType.FullName -like "System.Object" -and

$_.Methods.Count -eq 1 -and $_.Methods[0].IsConstructor -and

$_.Methods[0].MethodBody.Instructions.Count -eq 4 -and $_.CustomAttributes.Count -eq 0})

{

$moduleDef.TopLevelTypes.Remove($topType) | Out-Null

}

Originally compiled as R2R binary and we don´t care about the native precompiled code, ILOnly == True

$moduleDef.IsILOnly = $true

$moduleDef.Write($obfuscated + "-cleaned.dll")

[Reflection.Assembly]::LoadFrom("C:\AsmResolver\AsmResolver.DotNet.dll") | Out-Null $obfuscated = "C:\RiotClientServices.dll" $moduleDef = [AsmResolver.DotNet.ModuleDefinition]::FromFile($obfuscated) # Removing junk methods foreach($type in $moduleDef.GetAllTypes()) { foreach($method in [array]$type.Methods.Where{$_.HasMethodBody}) { if(($method.MethodBody.Instructions.Where{$_.Opcode.Mnemonic -like "call" -and $_.Operand.FullName -like "*System.Console::WriteLine*"}).count -eq 5) { $type.Methods.Remove($method) | Out-Null } } } # Removing junk NestedTypes foreach($type in [array]$moduleDef.GetAllTypes().Where{$_.IsNested -and $_.Methods.Count -eq 1 -and $_.Methods[0].IsConstructor -and $_.Methods[0].MethodBody.Instructions.Count -eq 4}) { foreach($topType in $moduleDef.TopLevelTypes.Where{$type -in $_.NestedTypes}) { $topType.NestedTypes.Remove($type) | Out-Null } } # Removing junk TopLevelTypes foreach($topType in [array]$moduleDef.TopLevelTypes.Where{$_.BaseType.FullName -like "System.Object" -and $_.Methods.Count -eq 1 -and $_.Methods[0].IsConstructor -and $_.Methods[0].MethodBody.Instructions.Count -eq 4 -and $_.CustomAttributes.Count -eq 0}) { $moduleDef.TopLevelTypes.Remove($topType) | Out-Null } # Originally compiled as R2R binary and we don´t care about the native precompiled code, ILOnly == True $moduleDef.IsILOnly = $true $moduleDef.Write($obfuscated + "-cleaned.dll")

```
[Reflection.Assembly]::LoadFrom("C:\AsmResolver\AsmResolver.DotNet.dll") | Out-Null
$obfuscated = "C:\RiotClientServices.dll"
$moduleDef = [AsmResolver.DotNet.ModuleDefinition]::FromFile($obfuscated)

# Removing junk methods
foreach($type in $moduleDef.GetAllTypes())
{
    foreach($method in [array]$type.Methods.Where{$_.HasMethodBody})
    {
        if(($method.MethodBody.Instructions.Where{$_.Opcode.Mnemonic -like "call" -and
            $_.Operand.FullName -like "*System.Console::WriteLine*"}).count -eq  5)
        {
            $type.Methods.Remove($method) | Out-Null
        }
    }
}
# Removing junk NestedTypes
foreach($type in [array]$moduleDef.GetAllTypes().Where{$_.IsNested -and $_.Methods.Count -eq 1 -and
        $_.Methods[0].IsConstructor -and $_.Methods[0].MethodBody.Instructions.Count -eq 4})
{
    foreach($topType in $moduleDef.TopLevelTypes.Where{$type -in $_.NestedTypes})
    {
        $topType.NestedTypes.Remove($type) | Out-Null
    }
}
# Removing junk TopLevelTypes
foreach($topType in [array]$moduleDef.TopLevelTypes.Where{$_.BaseType.FullName -like "System.Object" -and
```

```
        $_.Methods.Count -eq 1 -and $_.Methods[0].IsConstructor -and
        $_.Methods[0].MethodBody.Instructions.Count -eq 4 -and $_.CustomAttributes.Count -eq 0})
{
    $moduleDef.TopLevelTypes.Remove($topType) | Out-Null
}
# Originally compiled as R2R binary and we don´t care about the native precompiled code, ILOnly == True
$moduleDef.IsILOnly = $true
$moduleDef.Write($obfuscated + "-cleaned.dll")
```

The deobfuscation reduced the size, count of methods and classes to:

- `RiotClientServices.dll` size ≈ 124KB, 158 methods, 35 classes
- `LiırarySharing.dll` size ≈ 30KB, 220 methods, 28 classes

Figure 16: Deobfuscated code of the "LirarySharing.dll" – class "Serialize"

Further deobfuscation of the names (methods, classes, etc.) could be processed using the de4dot tool. Still, we should not forget to supply all binaries together (to preserve the debugging – the main module is referencing and using the library).

Plain text

Copy to clipboard

Open code in new window

EnlighterJS 3 Syntax Highlighter

.\de4dot.exe "C:\RiotClientServices.dll" "C:\LirarySharing.dll"

.\de4dot.exe "C:\RiotClientServices.dll" "C:\LirarySharing.dll"

```
.\de4dot.exe "C:\RiotClientServices.dll" "C:\LirarySharing.dll"
```

Figure 17: "LirarySharing.dll" – deobfuscated vs. obfuscated

After deobfuscation, the main logic of the BundleBot could be seen in the module `RiotClientServices.dll`:

Figure 18: Main logic of the BundleBot in the "RiotClientServices.dll" module

The main functionality could be summarized as follows:

- `Sleep` patch detection (Anti-Sandbox)
- Installing/Uninstalling persistence via registry path `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, value `ApplicationName`
- Information Stealing
- C2 BOT communication (TCP socket communication to C2 IP `51.79.180.158`, port `5505`)

Stolen and exfiltrated data:

- Telegram data
- Discord token
- Computer information – HWID (first 10 bytes of computed MD5 from CPU count, UserName, MachineName, OSVersion, and TotalSize of OS Drive), Windows version, UserName, Windows region – country, IP info (IP address, country, region, city, timezone, ISP) – retrieved via services `http://icanhazip[.]com` and `http://ip-api[.]com/json/`
- Web Browsers data (Chrome, Edge, Opera, Brave, Coccoc, Firefox) – profile name, decrypted key, credential cookies, passwords, bookmarks, extensions, credit cards

- Facebook account – uid, name, cookie, access_token, pages, ad account info, business info, browser name, browser profile name
- Captured screenshot

All stolen data and C2 communication are processed, serialized, and compressed with the help of the `LiarySharing.dll` library. This library defines the main capabilities of bot communication as follows:

- Bot Identification – start TCP socket, send stolen data to C2 (serialized, GZip compressed)
- Bot Update (complete reinstall of the bot with a new version)
- Bot Off (only exit the process)
- Bot Kill (remove persistence and exit the process)
- Bot Re-update Identification (steal data and send to C2)
- Bot Get Browser Data (collect web browser data and send to C2)

Example of dissected network traffic related to Bot Identification:

Figure 19: Dissection of network traffic related to Bot Identification

Noteworthy, we encountered a slightly different version of the final BundleBot that shares all code, but in addition, it has the functionality to exfiltrate stolen data to C2 via HTTPS. All such samples we analyzed were configured to avoid using the bot communication via TCP and were just configured to exfiltrate the stolen data to the C2 web server with URL `https://cp.bemilcoin[.]io/api/cookiePc?cookie`. The added code can be seen below.

Figure 20: Code related to data exfiltration to C2 web server

The stolen data are exfiltrated to the C2 web server in the content of the ZIP archive.

Figure 21: Stolen data exfiltrated to the C2 web server as a ZIP archive

As we were able to get the original pdb files of these specific versions and could confirm that a dependency resulting from this new capability was presented, this functionality was not caused by manual alternation but was more likely added as a recent feature.

Conclusion

By monitoring the BundleBot for a few months, we got a deeper insight into its infection vectors and how it abuses the dotnet bundle (single-file), self-contained format that results in very low or no static detection at all. The combination of this specific file format and multi-stage infection spread this threat silently under the radar for several months.

The delivering method via Facebook Ads and compromised accounts is something that has been abused by threat actors for a while, still combining it with one of the capabilities of the revealed malware (to steal a victim´s Facebook account information) could serve as a tricky self-feeding routine.

In this research, we pointed out how the attacker abused the dotnet bundle (single-file), self-contained format and hardened the analysis even more by applying a custom-made obfuscation. We introduced and detailed several tools and techniques to properly analyze similar threats and to avoid any problem that could occur during reverse engineering.

Check Point customers remain protected from the threats described in this research.

Check Point's Threat Emulation provides comprehensive coverage of attack tactics, file types, and operating systems and has developed and deployed a signature to detect and protect customers against threats described in this research.

Check Point's Harmony Endpoint provides comprehensive endpoint protection at the highest security level, crucial to avoid security breaches and data compromise. Behavioral Guard protections were developed and deployed to protect customers against threats described in this research.

Threat Emulation:

- InfoStealer.Wins.BYOSDownloader.A

Harmony Endpoint:

- InfoStealer.Win.FakeGoogleAI.A
- InfoStealer.Win.FakeGoogleAI.B
- InfoStealer.Win.FakeGoogleAI.C
- InfoStealer.Win.FakeGoogleAI.D
- InfoStealer.Win.FakeGoogleAI.E

IOCs

Files

Name	Hash SHA-256	Description
Google_AI.rar	dfa9f39ab29405475e3d110d9ac0cc21885760d07716595104db5e9e055c92a6	RAR Archive contai Downloader stage
ADSNEW-1.0.0.3.zip	303c6d0cea77ae6343dda76ceabaefdd03cc80bd6e041d2b931e7f6d59ca3ef6	Pass-protected ZIP a containing BundleBo (pass:alex14206985a
Bot_Server6_1.0.0.3.zip	90b37f26d7574a23437a2f0ad75d3cce5ecf3928efb58beacedde289fd3568bf	Pass-protected ZIP a containing BundleBo (pass:alex14206985a
ADS_1.0.0.3.zip	af92d0545ce01e5dcbe228a43babe6281a1631836e5631286908c7f0aa225f3d	Pass-protected ZIP a containing BundleBo (pass:alex14206985a
FB_1.0.0.3.zip	25c0f65acb3ecfe435a39bed3f5013eadd85eca1e78a0dc754cb4b82389ee4bb	Pass-protected ZIP a containing BundleBo (pass:alex14206985a
COIN_1.0.0.4.zip	a99dbc0cb0a051ec68bd89c468fd589b201380f47330bdedbb69f9b076099711	Pass-protected ZIP a containing BundleBo (pass:alex14206985a
Coin_1.0.0.0.zip	b47ac379cc23a059e1aaaba351f528c5a955fd56da35928c0bc0043c4ab8b38a	Pass-protected ZIP a containing BundleBo (pass:alex14206985a
RiotClientServices.zip	3198a613574a8ab84637bf80ebe5f6a56c851aa292973515c5de856f1e958d6d	Pass-protected ZIP a containing BundleBo (pass:alex14206985a
SubwaySub.dll	a1389d02c0b7892ffeae60b7869f3a761c2326629bd1c304839a1e8b7400744e	Downloader stage – main module
GoogleAI.dll	22bb60b0ea0d5bb57e105287843867880f336ddafa1545332e2de16d412cde12	Downloader stage – main module
PDF Reader.dll	4b4f69b01edd2c96db6374a9d0d980f5023383d440914831301f19d1d29ae4d9	Downloader stage – main module
PDF.dll	bc1fceb2d6c5dc7bedfdf1790ac0f06ccf0a9777c79d831d037dff10ae4ace8f	Downloader stage – main module
PDF.dll	d0146a3bbed91d5680c9b44c0f0e69deabe4d6c0f114e50d9fdee9cd202242fc	Downloader stage – main module
PDF Reader.dll	1c27a31830946ca806be10d07dc67b185d3f1e2bbc76cd5365719055966600fb	Downloader stage – main module
Smart Miner.dll	20b833c028322139b81e220cc165513ec2d4a490cfbd84e88e985a84d3173025	Downloader stage – main module
Chaturbate.dll	0e2bb46c9cb2baa0263824f4a6725a2e4db2541eafd392f25bd9a4921a2e04f3	Downloader stage – main module
Mario.dll	4c39df6e78b110e4912f3cb543130297b9b3cc3d33daa2d613999a1b991ba763	Downloader stage – main module

Name	Hash SHA-256	Description
Super Mario 3D World.dll	9b4c6dcee2848e2c23cffe1b8925ebc37d4d98a441fe6b0ff82dc788595a68be	Downloader stage – main module
Canva.dll	601f888abbb545b003ed37e3835237de7915874893f22ee5bb6ebc9f5db618b5	Downloader stage – main module
PDF.dll	2038aa28b4e23806030f945aadcf5dbbfa2e3f7ae2b924bd987fda62f87773fc	Downloader stage – main module
PDF.dll	cd1c00427973b7ff7bac1803d35c071ff0fdeb975c4eb5a54829bedf12c4d136	Downloader stage – main module
GoogleAI.exe	5ac212ca8a5516e376e0af83788e2197690ba73c6b6bda3b646a22f0af94bf59	Downloader stage – file
PDF.exe	67f24b507fe2f6dc06a294b85486cfa1dba6af188e59c51a74adc3b3f9ed29d8	Downloader stage – file
Chaturbate.exe	97f777abfeada170c1caa625ffbf12b8d097ae5331f3f4c5b57dad4fc0c4f8c1	Downloader stage – file
Super Mario 3D World.exe	8d1aa8ca616afc7fdf3cd6552e94fb486196d67e062adf5c97ada05b7b176985	Downloader stage – file
PDF.exe	9e6175a02a129fe559f108f6dced7fb6bf66c468cfb3ca276f3621ab8c312e91	Downloader stage – file
PDF Reader.exe	953e1b59b2163ddafaafe7872033ae6351a46500b575a717c853b6393d2c7ef6	Downloader stage – file
RiotClientServices.exe	230e5844ac0c767baf4d5d660f9ebcd0a9dd7f5a5ec5869387f53fa3eb902aa3	BundleBot stage – bu
RiotClientServices.exe	26d0853adcec8b273346924e97170226abd7b800b5ee51f6768c58ac45f59d20	BundleBot stage – bu
RiotClientServices.exe	37a06e2e28d16096c45bfd3ef2679fe8dc722810b6f6119b7dc5f1483e66ec01	BundleBot stage – bu
RiotClientServices.exe	50b7447d83715b8b7b36a15d0e7c7b8ae881a56dc0f39eb1aa22604e00f97d17	BundleBot stage – bu
RiotClientServices.exe	6552a05a4ea87494e80d0654f872f980cf19e46b4a99d5084f9ec3938a20db91	BundleBot stage – bu
RiotClientServices.exe	6834be1cbde6718d153a729f2e68e3f3b21bcbcb51a9f381e98f78b7a414969f	BundleBot stage – bu
RiotClientServices.exe	bfa7b12cc68b9cd26022a4c611ceaa473c84ffe36bb8008c67c1692b968b88d8	BundleBot stage – bu
RiotClientServices.dll	0ba224ecc2546d0a5ccc13bc8f929ec0035ca884fce44c8aebcfec185550169c	BundleBot stage – ex main module
RiotClientServices.dll	0c5ef531c2d5be15ef2a031c381a9531db22e030b14a1c2de311c68da23fef48	BundleBot stage – ex main module
RiotClientServices.dll	2e0492507ed4127b25e523444b205c58312902fa0bf2f5697c184049af5e4e18	BundleBot stage – ex main module
RiotClientServices.dll	41c884718ce264195d75695252b22021680c6d5470a303f999f3f333a5eef9c9	BundleBot stage – ex main module
RiotClientServices.dll	5beb1ce875166ec47ee7fbcd9e48c891fe0b27ccec04edf3da82bf8b3b2ea04b	BundleBot stage – ex main module
RiotClientServices.dll	84319f401994ca83d2659aef8fa5810224f4a0fef2d3ed6883a5a265d3a8c291	BundleBot stage – ex main module

Name	Hash SHA-256	Description
RiotClientServices.dll	9b0a6fdc188de6d80117f9f0894c456e9f541f19ba5b4ed8cfd03e86d8fb8af9	BundleBot stage – e main module
LiarySharing.dll	386189e521d431428157cf37b4653444f8c2116ee0a5229313012c43e5839edd	BundleBot stage – e data serialization libr
LiarySharing.dll	4856cdb407d67ee82d44e1cd606e382cde7b6bcaf9127dd7924e2d604c1cad38	BundleBot stage – e data serialization libr
LiarySharing.dll	6632c655875279ed1c19937805416a716d9994db71c8e30d2c8b4a3a3c3f9620	BundleBot stage – e data serialization libr
LiarySharing.dll	7a0cd3cc214b312cda20a54f7e0e93509fbcf5f6e6d9f41fd95d6dfa3bb5bcdc	BundleBot stage – e data serialization libr
LiarySharing.dll	a47d68411f64887300800cbe471f3cb24047e2e352bff74b810ad1940cfff85c	BundleBot stage – e data serialization libr
LiarySharing.dll	fca477e3e0fe31dfc14a4bade9828da267b6f234c343f9fb654e6921ba71bd08	BundleBot stage – e data serialization libr

Network

URL	IP Address	Description
https://drive.google[.]com/uc?id=1obRjbjOkXO3aCKKVa6BHKYqsROXRVmzL&export=download&confirm=t	–	URL to download BundleBot stage (embedded in downloader)
https://drive.google[.]com/uc?id=1-mC5c7o_B1VuS6dbQeDAAqLuPbfAV58O&export=download&confirm=t	–	URL to download BundleBot stage (embedded in downloader)
https://drive.google[.]com/uc?id=1f6QEiRPXZ1GKKtu-G_d_iQ448xYPGfMC&export=download&confirm=t	–	URL to download BundleBot stage (embedded in downloader)
https://drive.google[.]com/uc?id=1ypYJpu5pgaFRnXx64ZnCCfoGaUMYBt5E&export=download&confirm=t	–	URL to download BundleBot stage (embedded in downloader)

URL	IP Address	Description
https://drive.google[.]com/uc?id=1S2G8OmhMREHS8l24hG-BmGKINxEL_DD5&export=download&confirm=t	–	URL to download BundleBot stage (embedded in downloader)
https://drive.google[.]com/uc?id=1Uvyx_Fj7wF9cVnq3IwIAm5-i2IROsi0R&export=download&confirm=t	–	URL to download BundleBot stage (embedded in downloader)
https://drive.google[.]com/uc?id=1teMU5O6VYsRjH9GVQf1V7h5ya-3Ssbkn&export=download&confirm=t	–	URL to download BundleBot stage (embedded in downloader)
–	51.79.180[.]158:5505	C2 – BundleBot TCP connection
–	85.239.242[.]27:5505	C2 – BundleBot TCP connection
–	139.99.80[.]193:5505	C2 – BundleBot TCP connection
–	139.99.38[.]193:5505	C2 – BundleBot TCP connection
https://cp.bemilcoin[.]io/api/cookiePc?cookie	–	C2 – BundleBot HTTPS (exfil)

References

1. Single-file deployment: https://learn.microsoft.com/en-us/dotnet/core/deploying/single-file/
2. Runtime configuration: https://learn.microsoft.com/en-us/dotnet/core/runtime-config/
3. Runtime configuration: https://github.com/dotnet/sdk/blob/main/documentation/specs/runtime-configuration-file.md
4. DnSpyEx “latest” releases: https://github.com/dnSpyEx/dnSpy/releases

5. DnSpyEx issue, related to dotnet bundle: https://github.com/dnSpyEx/dnSpy/issues/48
6. AsmResolver: https://github.com/Washi1337/AsmResolver
7. De4dot: https://github.com/de4dot/de4dot

Source: https://research.checkpoint.com/2023/byos-bundle-your-own-stealer/