

# Forum:[Hack Forums](#)

Thread: [\[GUIDE\] Simple-yet effective AV EVASION Methods](#)

**r00d** - April 20, 2017, 9:01 pm

## Preface

I am not responsible for anything you do with this, as it is for educational purposes only. Do not infect people with malware, it is illegal, only learn from it.

By no means am I an expert on this, but I have been researching it recently and thought it might be useful for some of you to have.

This guide contains a mixture of copy pasta ready code and fragments of code. With these methods and some thought, you should be able to create an UD stub, possibly FUD if you put your mind to it.

This guide will very loosely cover encryption and decryption methods, as those are straight forward. It also will not cover runPE or other code injection methods in depth, as those are honestly too complex for me to explain, and I still can't completely understand them...

Lastly, I do not take credit for discovering these methods. If anything, regard this as an index of popular and well-working methods for bypassing AVs, with some more explanation and some lesser known things, too.

## Section 1 - AV Detection Types

There are three main reasons a piece of malicious code gets detected:

### 1. Signature-Based analysis

The most advanced anti-virus programs today all have huge databases of malicious code signatures. The most powerful anti-viruses update these databases constantly, always trying to keep their users protected from new malware. Signatures are simply ways of blacklisting common malware.

### 2. Heuristic analysis

Heuristic analysis (often called HEUR) is a way of detecting malware by looking at its behavior and the types of things the malware is trying to do. For example, a program that tries to inject something into the memory of another process -- normal, user-friendly programs don't normally do this.

### 3. Dynamic analysis (VM/Runtime)

Dynamic analysis has to do with the anti-virus actually allocating system resources into a virtual machine to test out an executable, and see if it is safe for the user. If in the VM nothing bad happens that raises any flags, the file is passed along as safe. This will be one of the biggest points this guide touches on, evading the AV's virtual machine, and detecting the environment the program is running in.

## Section 2 - Logic

The basic idea of what the majority of the code will help you achieve:

Main is invoked

(Pretend to do something useful)

Detect if the code is being analysed

If being analysed, pretend to do something useful then return

Else, run shellcode

A more classic approach to this would be to end the process immediately if it were being analysed. I will explain why it needs to pretend to do something useful later.

### Section 3 - Is this a VM?

Along with the benefits of detecting if the program is in a VM described above, there also is the benefit of making your code look "less evil" when people who are suspicious launch it in something like sandboxie.

So, let's get right into it...

#### 1. Junk loops

This is possibly the oldest and most famous method for evading AVs' VMs. It used to be back in the day that whenever an anti-virus launched a virtual machine, it had a very short time limit before it were to close down. Now, since computers are much faster and have much more processing power, this isn't as necessary, yet some AVs still have the time limit. Anti-viruses will attempt to skip over `Thread.Sleep()` statements, something that will be touched on later, so the more effective way to do such a thing is a for loop that increments a variable for no apparent reason.

```
Code int x = 133700000;
int y = 0;
for(int i = 0; i < x; i++)
{
    y++;
}
if(y == x)
{
    shellcodeIsAgo();
}else
    PretendToDoSomethingUseful(); //explanation further down...
```

This forces the AV to waste its time on incrementing some stupid variables. If it times out the VM, great. If it doesn't... more protection!!

#### 2. Are you warping through the space time continuum? (Time distortion)

As stated above, some AVs' VMs (to save time) will skip over `Thread.Sleep(x)`, but not adjust the VM's time accordingly. Let's exploit this...

```
Code long time = System.DateTime.UtcNow.Ticks;
```

```
Thread.Sleep(1551);
long timeAfter = System.DateTime.UtcNow.Ticks;
```

```
if(timeAfter - time > 1551)
{
    shellcodeIsAgo();
}
else
```

```
    PretendToDoSomethingUseful(); //explanation further down...
```

The code above measures the current system's time in ticks. Then the thread sleeps for 1,551 milliseconds, then measures the system's time again in ticks. Because nothing is perfect, it actually takes a little more than 1,551 milliseconds for the next piece of code to run. Due to randomness, it is not viable to check within a certain realm of error what the difference was, so it is fine just to leave it as a greater than statement. Also, do not set it to greater than or equals, as that picks up a few more detections. Some AVs make the mistake of not actually simulating a real system, and instead move the time forward exactly 1,551 ticks. I was able to get 2 less detections with that small trick... Credit goes out to some random guy on an Indian forum I found somewhere, as well as Emeric Nasi!

### 3. Who has 2 cores nowadays?

System have unusually low specs? Yes, probably a VM. This may not suit everyone depending on their clients' setups, but it's useful for some.

```
Code performanceCounter ramCounter;
ramCounter = new PerformanceCounter("Memory", "Available MBytes");
```

```
if(Environment.ProcessorCount > 2 && ramCounter.NextValue() > 4000)
{ // if system has more than two cores and more than 4GB (standard amount of ram)
    shellcodeIsAgo();
}
else
```

```
    PretendToDoSomethingUseful(); //explanation further down...
```

It is quite simple, really. 4GB is basically the standard amount of RAM every computer should have nowadays. Most anti-viruses' VMs won't allocate that much horsepower to a dynamic analysis, it usually isn't worth it and may disrupt the smoothness of operation for the user for a short period. Simple, yet effective.

#### \*. Pretend to do something useful!

As anti-viruses and their developers got more advanced, they realized that a program that shuts down after doing nothing but asking for system requirements and such may be somewhat suspicious... so let's pretend to do something!

Honestly, this can be setup however you prefer. I would recommend you fill it with junk code and randomness, and make sure it asks for an object. You can pass it null, it doesn't really matter. Here's mine!

```
Code public static void PretendToDoSomethingUseful(object useful)
{
    useful = 1;
    try
    {
        Console.Write(useful);
    }
    catch (Exception) { }
```

```

Thread.Sleep(1);
byte xD = Encoding.ASCII.GetBytes("800815")[69];

for (int i = 0; i > -100; i--)
{
    byte y = xD;
    i++;
    i--;
}
}

```

As I said, complete randomness. In a beautiful way.

Alright, so that's it for bypassing sandbox/AV detections. There are still many other ways to get detections, more specifically generic detections! -- Just do some googling. Onwards and upwards... :)

.

## Section 4 - I swear, I'm a legit program.

You more than likely have seen people suggest making their programs look more legit to bypass anti-viruses... and yes, this is a completely viable and important part in getting a file UD. Some detections are simply based off of how legitimate the executable looks. This is (kind of) the idea with SmartScreen, requiring a license for a program to be executed, except there are no public exploit for bypassing it as of now, so it will be disregarded.

### 1. Packers / Packing

Packers "pack" executable files among other things to make them take up more space. Packers work many different yet basically the same ways, so we will only look into manual packing. The easiest way to do this is to include an embedded resource that takes up a lot of space when compiling your stub. If you don't know how to add an embedded resource, here's the MSDN on it: [support.microsoft.com/en-us/help/319292/how-to-embed-and-access-resources-by-using-visual-c](https://support.microsoft.com/en-us/help/319292/how-to-embed-and-access-resources-by-using-visual-c)

As far as what type of file to embed, I would recommend writing a program that simply writes a byte[] of whatever amount of bytes you want to back to a random file name, and then use that. Keep in mind, 1,024 bytes = 1kb, 1,024kb = 1mb, 1,024mb = 1gb, etc... so 1mb = 1,000,000 bytes or so.

### 2. Assembly Info

An extremely simple yet effective way of avoiding generic detections is to just add a version number, company name, etc. when you compile your stub. In case you do not know how to do this, navigate to your project's properties tab > Application > Click "Assembly Information..." and make it look legit!

### 3. Assembly Info

Also, back at the application page you may choose an icon. I recommend IconArchive (google it) for high quality icons.

## Section 5 - Encryption. Duh

I intentionally left this to be last on the guide. [i]Every single stub ever should feature encrypted

shellcode that is decrypted and then run. It is safe to say that you never want to write the decrypted shellcode to the disc as you risk getting a detection, but instead decrypt the shellcode and run it entirely in memory. There is an elegant way to do this in C#:

```
Code Assembly exeAssembly = Assembly.Load(bytecode); // Load the decrypted shellcode in a byte[]
```

```
object[] parameters = new object[1];
try
{
    exeAssembly.EntryPoint.Invoke(null, parameters); // Invoke entry point
}
catch(Exception)
{
    exeAssembly.EntryPoint.Invoke(null, null);
}
```

This only works for other .NET executable shellcode, but is very short and sweet. Of course this is all done in memory, and does not write anything to the disc. There are better, more sophisticated ways to do it, but there isn't much of a difference that it is worthwhile (unless you are talking code-injection, but that's something for a later date).

Thank you for reading! Hopefully some people learned from this. If I was misleading at all or made any mistakes, please let me know! :)

**Decim** - April 21, 2017, 12:47 am

This looks like an interesting read, will check back later.

Thanks for taking the time to write it.

**OrbitalData** - April 21, 2017, 1:29 am

New here on HF thanks for such a detailed guide! Very helpful for learning; bookmarking this guide.

**r00d** - April 21, 2017, 5:02 am

(Apr 21, 2017 12:47 AM)Arcane. Wrote: [\[CITE\]](#) This looks like an interesting read, will check back later.

Thanks for taking the time to write it.

You're welcome! ?

(Apr 21, 2017 01:29 AM)OrbitalData Wrote: [\[CITE\]](#) New here on HF thanks for such a detailed guide! Very helpful for learning; bookmarking this guide.

Hope it's helpful! Once I'm able to PM, I'll send you some more stuff if you're interested ?