Corelan Team

:: Knowledge is not an object, it's a flow ::

« ROP your way into B-Sides Las Vegas 2011
Installing Watobo on BackTrack 5 »

Please consider donating: https://www.corelan.be/index.php/donate/

mona.py – the manual

Published July 14, 2011 | By Corelan Team (corelanc0d3r)

Introduction

This document describes the various commands, functionality and behaviour of mona.py.

Released on june 16, this pycommand for Immunity Debugger replaces pvefindaddr, solving performance issues, offering numerous improvements and introducing tons of new features. pvefindaddr will still be available for download until all of its functionality has been ported over to mona.

Downloading mona.py

The mona project page is located here :
https://github.com/corelan/mona

You can download the latest version here,

Together with the release of this documentation, we are also proud to be able to release mona.py v1.1 .

Important : Mona only works on Immunity Debugger 1.83 and up, and WinDBG.

When you have downloaded mona.py, simply save the file into the PyCommands folder. In a typical installation, this folder is located here :

```
C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands
```

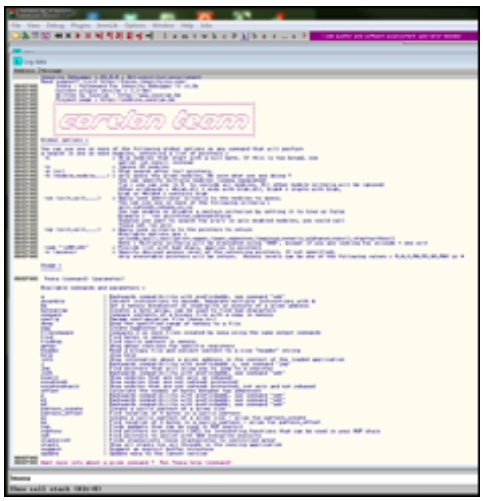
That's it. mona.py is now installed.

Basic usage

Open Immunity Debugger. At the bottom of the application you should see an input box (command bar)

Enter !mona and press return.

Open the log window (ALT-L) and you should get a full page of information about mona (options and commands)



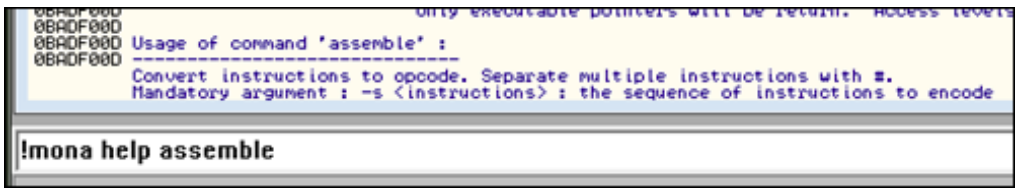
At the top, you can find the global options. The second half of the output contains all available commands.

If you want more information about the use of a certain command, you can simply run !mona help <command>.

Suppose you want more info about the use of the "assemble" command, run

```
!mona help assemble
```

output :



Keeping mona.py up-to-date

Before using mona, I strongly recommend making sure you are running the latest version. If you are using the stable release, you won't see updates that often (new version releases are often announced on twitter or Google+).

If you are using the trunk release of mona.py, you might want to update on a regular (daily) basis.

Updating is very simple. Make sure your device has access to redmine.corelan.be (https or http) and run

```
!mona update
```

By default, a connection is made to https://redmine.corelan.be. If, for whatever reason, you are unable to use https, you can use the -http switch to force the use of http:

```
!mona update -http
```

Starting from mona.py v1.1 you can switch between stable and trunk releases, by using the -t <version> option.

So, if you are running trunk and you want to change to release, run

```
!mona update -t release
```

if you want to switch to the latest trunk version, run

```
!mona update -t trunk
```

Install and update mona + basic usage

View the video online here

Setting up your debugger environment

working folder

One of the issues we had with pvefindaddr was the fact that all output was written into

the Immunity Debugger program folder. Since the output files for a given command, ran on different applications, carry the same name, you would basically end up overwriting the previous output.

Mona allows you to group the output and write then into application specific folders. In order to activate this feature, you will need to set a configuration parameter, for example :

```
!mona config -set workingfolder c:\logs\%p
```

This will tell mona to write the output to subfolders of c:\logs. The %p variable will be replaced with the process name currently being debugged. Note that, if you are not debugging an application, Immunity returns the name of the previously debugged application. In order to prevent mona from writing files into an application folder at that time, mona also checks for the process ID of the debugged app. If the pid is 0 (= not attached), then output will be written into a folder "_no_name"

If you want to further group output, you can even use the %i variable in the workingfolder parameter. This variable will get replaced with the process ID of the application being debugged.

This configuration parameter is written into a file called "mona.ini", inside the Immunity Debugger application folder.

exclude modules

Mona has a second configuration parameter, allowing you to always exclude certain modules from search operations. If you want to ignore modules loaded by shell extensions, virtual guest additions, antivirus, etc etc, you can simply tell mona to ignore them using the following config command.

```
!mona config -set excluded_modules "module1.dll,module2.dll"
!mona config -add excluded_modules "module3.dll"
```

If you want to remove a module from the list, simply look for mona.ini in the Immunity Debugger program folder and edit the file.

author

This variable will be used when producing metasploit compatible output. If you set this variable, it will be used in the author section of a metasploit module.

```
!mona config -set author corelanc0d3r
```

Global options

Global options allow you to influence and fine tune the results of any search command that produces a list of pointers.

option -n

If you use option -n, all modules that start with a null byte will be skipped. This will speed up the searches, but it might miss some results because the module might actually contain pointers that don't start with a null byte. If this behaviour is too broad, and if you only

want to exclude pointer with null bytes, then you should either use option -cp nonull or option -cpb '\x00' (see below)

option -o

This option will tell mona to ignore OS module from search operations. As you will see in the documentation of each command, this is often the default behaviour. You will be able to overrule the behaviour using the -cm option (see later)

option -p

This option takes one argument : a numeric value. Option -p allows you to limit the number of results to return. If you only want 5 pointers, you can use option -p 5

option -m

This option allows you to specify the modules to perform the search operation on. If you specify -m, it will ignore all other module criteria (set by option -cm or set as default behaviour for a given command). You can specify multiple modules by separating them with comma's.

Example : suppose you want to include all modules that start with "gtk", all modules that contains "win" and module "shell32.dll", then this is what the option should look like :

```
-m "gtk*,*win*,shell32.dll"
```

If you want to search all modules, you can simply use -m *

option -cm

This option allows you to set the criteria (c) a module (m) should comply with to get included in search operations.

The available criteria are :

- aslr
- rebase
- safeseh
- nx
- os

If you want to include aslr and rebase modules, but exclude safeseh modules, you can use

```
-cm aslr=true,rebase=true,safeseh=false
```

(note : this will also include the non aslr and non rebase modules). You can use the "true" value to override defaults used in a given command, you can use "false" to force certain modules from getting excluded).

If you want more granularity, you can use the -m "module1,module1" option (basically tell mona to search in those modules and only those modules)

option -cp

The cp option allows you to specify what criteria (c) a pointer (p) should match. pvefindaddr already marked pointers (in the output file) if they were unicode or ascii, or

contained a null byte, but mona is a lot more powerful. On top of marking pointers (which mona does as well), you can limit the returning pointers to just the ones that meet the given criteria.

The available criteria are :

- unicode (this will include unicode transforms as well)
- ascii
- asciiprint
- upper
- lower
- uppernum
- lowernum
- numeric
- alphanum
- nonull
- startswithnull

If you specify multiple criteria, the resulting pointers will meet ALL of the criteria. If you want to apply "OR" to the criteria, you'll have to run multiple searches.

We believe this is a very strong feature. Especially with ROP exploits (where large parts of the payload consist of pointers and not just shellcode), the ability to finetune the pointer criteria is very important (and an often missing feature from other search tools).

Example : only show pointers that contain ascii printable bytes

```
-cp asciiprint
```

Example : only show pointers that don't contain null bytes

```
-cp nonull
```

option -cpb

This option allows you to specify bad characters for pointers. This feature will basically skip pointers that contain any of the bad chars specified at the command line.

Suppose your exploit can't contain \x00, \x0a or \x0d, then you can use the following global option to skip pointers that contain those bytes :

```
-cpb '\x00\x0a\x0d'
```

option -x

This option allows you to set the desired access level of the resulting pointers. In most cases, pointers should part of an executable page, but in some cases, you may need to be able to look for pointers (or data) in non-executable area's as well.

The available values for -x are :

- *
- R
- RW

- RX
- RWX
- W
- WX
- X

In all cases the default setting will be X (which includes X, RX,WX and RWX)

Output files

Before looking at the various commands, let's take a look at what the output files will contain :

First of all, they contain a header, indicating the mona version, the OS version, the process being debugged, the pid, and a timestamp

```
================================================================================
  Output generated by mona.py v1.1-dev
  Corelan Team - http://www.corelan.be
================================================================================
  OS : xp, release 5.1.2600
  Process being debugged : GenBroker (pid 7012)
================================================================================
  2011-06-21 18:33:18
================================================================================
```

Next, there will be a table with the module information for all loaded modules :

```
Module info :
--------------------------------------------------------------------------------
 Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll
--------------------------------------------------------------------------------
 0x002d0000 | 0x00326000 | 0x00056000 | True   | True    | False |  False   | True
 0x77b20000 | 0x77b32000 | 0x00012000 | False  | True    | False |  False   | True
 0x74980000 | 0x74aa3000 | 0x00123000 | False  | True    | False |  False   | True
 0x77a80000 | 0x77b15000 | 0x00095000 | False  | True    | False |  False   | True
 0x00c10000 | 0x00ed5000 | 0x002c5000 | True   | True    | False |  False   | True
 0x7c800000 | 0x7c8f6000 | 0x000f6000 | False  | True    | False |  False   | True
 0x76780000 | 0x76789000 | 0x00009000 | False  | True    | False |  False   | True
 0x7c3a0000 | 0x7c41b000 | 0x0007b000 | False  | True    | False |  False   | True
 0x7c900000 | 0x7c9b2000 | 0x000b2000 | False  | True    | False |  False   | True
 0x015b0000 | 0x015b8000 | 0x00008000 | True   | False   | False |  False   | False
 0x71a90000 | 0x71a98000 | 0x00008000 | False  | True    | False |  False   | True
 0x00270000 | 0x0029f000 | 0x0002f000 | True   | False   | False |  False   | True
 0x77fe0000 | 0x77ff1000 | 0x00011000 | False  | True    | False |  False   | True
 0x71ad0000 | 0x71ad9000 | 0x00009000 | False  | True    | False |  False   | True
 0x71aa0000 | 0x71aa8000 | 0x00008000 | False  | True    | False |  False   | True
 0x774e0000 | 0x7761e000 | 0x0013e000 | False  | True    | False |  False   | True
 0x77f60000 | 0x77fd6000 | 0x00076000 | False  | True    | False |  False   | True
 0x7e410000 | 0x7e4a1000 | 0x00091000 | False  | True    | False |  False   | True
 0x010e0000 | 0x010ef000 | 0x0000f000 | True   | True    | False |  False   | True
 0x71b20000 | 0x71b32000 | 0x00012000 | False  | True    | False |  False   | True
```

The fields listed are

- Base
- Top
- Size
- Rebase
- SafeSEH

- ASLR
- NX
- OS dll ?
- Module version
- Module name
- Path

Finally, the results of the search operation are listed (one line per pointer). Each line will contain a pointer, information about the pointer, and information about the module the pointer belongs to.

If possible, if there was already a file with the same file name in the output folder, the previous file will be renamed to .old first. If that was not possible, it will get overwritten.

Commands

Before looking at the various commands, it's important to remember that most of the commands have default behaviour in terms of pointer and/or module criteria.

We will clearly document what those values are. You can override these default by using one or more of the global options listed above.

a

The "a" command is an alias for "seh". In pvefindaddr, this command initiated a search for add esp+8 / ret instructions, but in mona, all searches have been combined into "seh".

If you run "a", it will kick off "seh" and perform a full search for pointers that should bring you back to nseh (in a seh overwrite exploit). For more info, see "seh" below.

assemble

The "assemble" command allows you to convert one or more assembly instructions into opcode.

Mandatory argument :

- -s <instruction>

You can specify multiple instructions by separating them with a #

Example : find opcodes for pop eax, inc ebx and ret :

```
!mona assemble -s "pop eax#inc ebx#ret"
```

Output :

```
Log data
Address    Message
0BADF00D   Opcode results :
0BADF00D   ----------------
0BADF00D    pop eax = \x58
0BADF00D    inc ebx = \x43
0BADF00D    ret = \xc3
0BADF00D    Full opcode : \x58\x43\xc3
0BADF00D   Action took 0:00:00.018000
```

bf

The "bf" command allows you to set breakpoint on exported or imported function(s) of selected module(s).

Mandatory argument :

- -t <type> : where <type> is ADD or DEL

Optional arguments :

- -f <function type> : set to import or export to read IAT or EAT. Default value : export
- -s <func,func,func> : specify the functions you want to set a breakpoint on (or remove breakpoint from). You can use wildcards *func, *func*, func*. If you want to set a bp on all functions, use -s *

Example : set breakpoint on all imported functions in myapp.exe, that contain "mem"

```
!mona bf -t ADD -f import -s *mem* -m myapp.exe
```

bp

The "bp" command allows you to set a breakpoint on a given address, which will be triggered if the address is read from or written to.

Mandatory arguments :

- -a <address>
- -t <type> : where <type> is either "READ" or "WRITE"

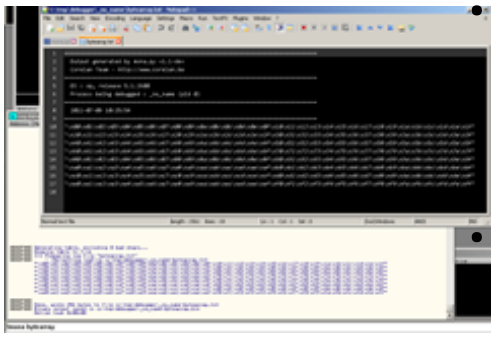
Note : the address should exist when setting the breakpoint. If not, you'll get an error.

Example : set a breakpoint when the application reads from 0012C431 :

```
!mona bp -a 0x0012C431 -t READ
```

bytearray

The "bytearray" option was implemented to assist exploit developers when finding bad chars. It will produce an array with all bytes between \x00 and \xff (except for the ones that you excluded), and writes the array to 2 files :



- a text file containing the array in ascii format (bytearray.txt)
- a binary file containing the same array (bytearray.bin)

Optional arguments :

-b <bytes> : exclude these bytes from the array
-r : show the array in reverse (starting at \xff, end at \x00)

We believe finding bad chars is a mandatory step in the exploit writing process. Over the last few weeks and months, a lot of people have asked us how to perform this task, so here are the steps to finding bad chars :

Let's say your exploit structure (classic saved return pointer overwrite) looks like this :

```
[ 280 bytes of junk ] [ EIP ] [ 500 bytes (shellcode) ]
```

We should try to make the process as reliable and smooth as possible, so we'll have to make sure we're not messing with the essential part of the payload, which is the part that should allow us to keep control over EIP.

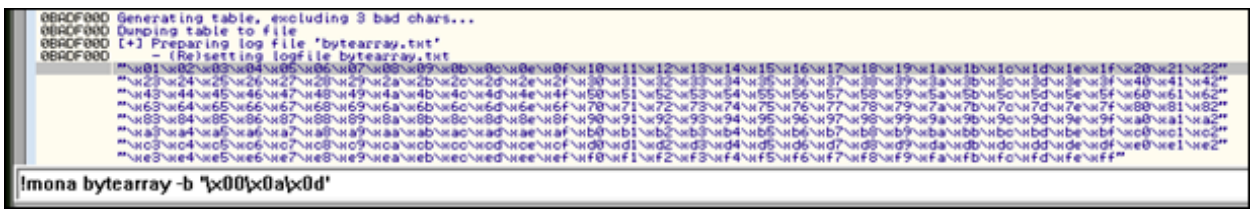
We'll keep the first 280 bytes and the 4 bytes (saved return pointer) intact. If we would change those and the first 280 bytes contain bad chars, then we might not even be able to control EIP anymore, making it very hard to debug and find bad chars.

We have a number of bytes available after overwriting saved return pointer. So that is the perfect location to put our array. It will make sure that we can make the application crash in a reliable way, allowing us to set a breakpoint on EIP and then compare the array of bytes in memory with the original array of bytes.

Before creating & using the byte array, you should ask yourself "what kind of bytes are very likely going to be breaking the payload or altering the behaviour of the application". If, for example, the buffer overflow gets triggered because of a string copy function, it's very likely that a null byte will break the payload. A carriage return/line feed (\x0a\x0d) are common bytes that would change the way the payload is read & processed by the application as well, so in case of a string based overflow, you'll probably discover that those are bad chars too.

So let's start with excluding those 3 bytes from the array. (Of course, this is all based on assumptions. If you want to be sure, don't exclude anything from the array yet).

```
!mona bytearray -b '\x00\x0a\x0d'
```



Open "bytearray.txt" and copy the array. Open your exploit and paste the array after overwriting EIP (so instead of the 500 bytes, put the array).

Then set the necessary breakpoints (at the pointer you will use to control EIP with in this case) and trigger the crash. Your breakpoint should be hit, and the byte array (or a portion of it) should be in memory somewhere.

The process of finding the bad chars is very simple. We can use another mona feature to read the original byte array from the binary file that was created, and compare those bytes with the array that is in memory at crash time.

Mona will then read the binary file, takes the first 8 bytes from the file, locates all instances of those 8 bytes in memory, and compare each of the bytes of the array with the bytes in memory, listing the ones that were not altered, and listing the ones that were changed/corrupted.

```
!mona compare -f bytearray.bin
```

At this point, there are 2 things you should check in the output of this command :

- Find the location (probably on the stack in this example) where EIP would jump to. If you are building an exploit for a saved return pointer overwrite, you will most likely find your payload at ESP. Look at ESP (or the location where your payload is) and jot down that address. Open compare.txt, look for that address and you will get the list of bytes that were corrupted. Corruption can result in 2 types of behaviour : either the byte gets changed (other bytes are still fine), or all bytes after the bad char are different. Either way, unless you know what you are doing, only write down the first byte that was changed and add that byte to the list of bad chars. Then repeat the entire process (create bytearray, paste array into exploit, trigger crash, compare). So, if you discovery that for example \x59 is a bad char too, simply run the bytearray command again :

  ```
  !mona bytearray -b '\x00\x0a\x0d\x5c'
  ```

  Repeat the entire process and do another compare. If the resulting array was found at the right location, unmodified, then you have determined all bad chars. If not, just find the next bad char, exclude it from the array and try again. The process might be time consuming, and there may even be better ways to do this, but this technique works just fine.

- Look at other locations where your array was found. Maybe one of those locations have a non-corrupted copy of the array, providing you with an alternative location to place your shellcode without having to bother about bad chars. All you would need to do is write some jumpcode (or egghunter) to jump to that location.

Video on the use of “bytearray” and “suggest”

finding badchars

View the video online here

compare

We already discussed the basic functionality of "compare" in the documentation of previous command.

Mandatory argument :

- -f <filename> : <filename> points to a binary file containing the bytes to locate and compare in memory

You can use this command to find bad chars (as explained earlier), but you can also use it to see if your shellcode got corrupted in memory.

You could, for example, write tag+tag+shellcode (egghunter egg) to a binary file and use the compare command to find the eggs and show if they got corrupted or not. If some of them do, others don't, you'll know if and how you have to tweak the start location of the hunter, or just use a checksum routine.

> The compare routine will also try to find unicode versions of the binary file
> There's no need to write the shellcode in unicode expanded format into the binary.

config

We have discussed all features of the config command at the top of this document.

dump

The dump command allows you to dump a block of bytes from memory to a file.

Mandatory arguments :

- -s <address> : the start address
- -f <filename> : the name of the file to write the bytes to

Optional argument :

- -n : size (nr of bytes to read & write to file) *or*
- -e <address> : the end address

Either the end address or the size of buffer needs to be specified.

Example : write all bytes from 0012F100 to 0012F200 to file c:\tmp\test.bin

```
!mona dump -s 0x0012F100 -e 0x0012F200 -f "c:\tmp\test.bin"
```

egg

This command will create an egghunter routine.

Optional arguments :

- -t : tag (ex: w00t). Default value is w00t
- -c : enable checksum routine. Only works in conjunction with parameter -f
- -f <filename> : file containing the shellcode
- -depmethod <method> : method can be "virtualprotect", "copy" or "copy_size"
- -depreg <reg> : sets the register that contains a pointer to the API function to bypass DEP. By default this register is set to ESI
- -depsize <value> : sets the size for the dep bypass routine
- -depdest <reg> : this register points to the location of the egghunter itself. When bypassing DEP, the egghunter is already marked as executable. So when using the copy or copy_size methods, the DEP bypass in the egghunter would do a "copy 2 self". In order to be able to do so, it needs a register where it can copy the shellcode to. If you leave this empty, the code will contain a GetPC routine.

If you don't specify a file containing shellcode, it will simply produce a regular egghunter routine. If you specify a filename, you'll have the ability to use option -c, which will insert a checksum routine for the shellcode in the file. The checksum routine will only work for that particular shellcode (or any other shellcode with the same size, producing the same checksum value :))

Using the various -dep* parameters, you can tell the egghunter routine to create code that will bypass DEP on the found egg, before running it. You can find more information about this technique here.

filecompare

The filecompare command allows you to compare the output of a certain mona command with the output of the same mona command from another system, looking for matching pointers, that point to the same instruction. If you are looking for pointers that match across OS versions for example, this may be the command you need.

Mandatory argument :

- -f "file1,file2" : specify all files to compare, separated with comma's. The first file in the list will be used as the reference

Optional arguments :

- -contains "blah" : only look at lines that contain a certain string
- -nostrict : this will also list pointers even if the instruction associated with the pointer is different

Let's say "seh_xpsp3.txt" was created on XP SP3, and "seh_win2k3.txt" was created on a 2003 server, then this is the command that will indicate the pointers that exist in both files :

```
!mona filecompare -f "c:\temp\seh_xpsp3.txt,c:\temp\seh_win2k3.txt"
```

The filecompare command will produce 2 files :

- filecompare.txt (containing the pointers that exist in all files)
- filecompare_not.txt (containing the pointers that don't exist in all files)

Note : the more files (and the bigger the files), the longer this process will take. (nested iterations)

find

The "find" command has improved a lot since its initial implementation in pvefindaddr. It's faster, but above all it's more flexible and has a shiney new "recursive" search feature.

Mandatory argument :

- -s <search> : search for a string, a series of bytes, an instruction, or a filename containing lines that start with a pointer (mona generated files) (-> use in conjunction with type "file")

By default, mona will attempt to guess the type of the search by looking at the search pattern. It is recommended, however, to use option -type <type>

Optional arguments :

- -type <type> : set the type of the search pattern. The type can be "asc", "bin", "ptr", "instr" and "file"
- -b <address> : the bottom of the search range
- -t <address> : the top of the search range
- -c : skip consecutive pointers, try to get the length of the total string instead
- -p2p : find pointers to pointers to your search pattern (this setting is an alias for setting -level to 1)
- -r <number> : use in conjunction with p2p : allows you to find "close" pointers to pointers. The number indicates the nr of bytes the resulting pointer can be above the actual location
- -level <number> : do recursive search (pointers to pointers). If you set level to 1, it will look for pointers to pointers to the search pattern. You can set level to a higher value to perform deeper search (ptr to ptr to prt to …). This is without a doubt one of the most powerful features of this find command.
- -offset <value> : this value will be subtracted from the pointer at a given level (see offsetlevel). Use in combination with -level and -offsetlevel
- -offsetlevel <value> : this is the level where the value will be subtracted from the pointer to search

By default, "find" will look for all locations (accesslevel set to *). Furthermore, by default, it will search the entire memory space (including the memory space of modules that were excluded).

Example : Find all locations where the string "w00t" can be found :

```
!mona find -type asc -s "w00t"
```

Example : Find all executable locations that have a pointer to "jmp ecx"

```
!mona find -type instr -s "jmp ecx" -p2p -x X
```

Example : Find ALL locations that have a pointer to any of the pointers in file c:\temp\seh.txt

```
!mona find -type file -s "c:\temp\seh.txt" -p2p -x *
```

Example : let's say you control ECX, and your payload is at ESP+4

This is the code that will give you control over EIP :

- MOV EAX,DWORD PTR DS:[ECX]
- CALL [EAX+58]

The idea is to retun to the payload at ESP+4 to initiate a rop chain.

Think about it. What kind of pointer would we need to put in ECX ?

Answer : We need a pointer to a pointer(2), where pointer(2) – 0x58 points to a pointer to ADD ESP,4/RET.

For sure you find one manually. If you have time.

In order to instruct mona to find all pointers, you'll need 2 commands :

- One to find all stackpivots of 4
- One to find the ptr to ptr-0x58 to ptr to one of the stackpivots

Solution :

```
# first, get all stackpivots, save output to c:\logs\stackpivot.txt
!mona config -set workingfolder c:\logs
!mona stackpivot -distance 4,4

# read the file and find ptr to ptr-58 to each of the pointers
!mona find -type file -s "c:\logs\stackpivot.txt" -x *
      -offset 88 -level 2 -offsetlevel 2
```

So basically, we need to apply the offset to level 2. (which is "look for ptr to ptr to ptr" in this case, because we already read ptr to the instructions into stackpivot)

Before looking for a pointer to a pointer in that level, the offset will be subtracted from the target pointer first.

Hint : in order to debug this routine, you'll have to set a breakpoint on reading [ECX]. Suppose you put 7a10b473 in ECX, then – prior to triggering the bug – set a breakpoint on reading that location :

```
!mona bp -t READ -a 0x7a10b473
```

findmsp

The findmsp command will find all instances or certain references to a cyclic pattern (a.k.a. "Metasploit pattern") in memory, registers, etc

This command has an important requirement : you have to use a cyclic pattern to trigger a crash in your target application.

At crash time, simply run findmsp and you will get the following information :

- Locations where the cyclic pattern can be found (looks for the first bytes of a pattern) and how long that pattern is
- Registers that are overwritten with 4 byte of a cyclic pattern and the offset in the pattern to overwrite the register
- Registers that point into a cyclic pattern, the offset, and the remaining size of the pattern

- SEH records overwritten with 4 bytes of a cyclic, offset, and size
- Pointers on the current thread stack, into a cyclic pattern (offset + size)
- Parts of a cyclic pattern on the stack, the offset from the begin of the pattern and the size of the pattern

In all cases, findmsp will search for normal pattern, uppercase,lowercase, and unicode versions of the cyclic pattern.

Optional argument :

- -distance <value> : This value will set the distance from ESP (both + and -) to search for pointers to cyclic pattern, and parts of a cyclic pattern. If you don't specify distance, findmsp will search the entire stack (which might take a little while to complete)

Other optional arguments in terms of patters to use : see pattern_create

findwild

The findwild command allows you to perform "wildcard" searches in memory. It takes the following mandatory argument :

- -s <chain of instructions>

Optional arguments :

- -depth <nr> : the number of instructions to search forward. Default value is 8. This value indicates the maximum length of the entire instruction chain.
- -b <address> : the base address for the search
- -t <address> : the top address for the search
- -all : indicate that you also want to return the instructions chains that might contains a possibly "bad" instruction (one that might break the chain)

Instructions are separated with #. You can use * to indicate that you want to allow any instruction(s). You can also use r32 to indicate that you want to allow any register.

Example : Search for a push (any register), later followed by pop eax, directly followed by inc eax, ending the chain with a retn :

```
!mona findwild -s "push r32#*#pop eax#inc eax#*#retn"
```

You can, of course, use any of the global options to finetune/limit the search.

Output will be written to findwild.txt

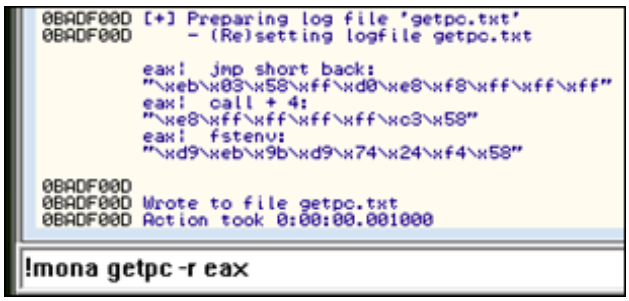
getpc

GetPC will output 3 different GetPC routines that put the getpc value into a register. This can come handy when you are writing some custom shellcode.

Mandatory argument :

- -r <reg> : where reg is a valid register

Example :



getiat

The getiat command allows you to dump the contents of the IAT of selected modules.

Optional argument:

- -s <keyword,keyword> : only show IAT entries that contain one of the keywords. You can use *keyword, *keyword*, or keyword* to be more specific in what output you want to get

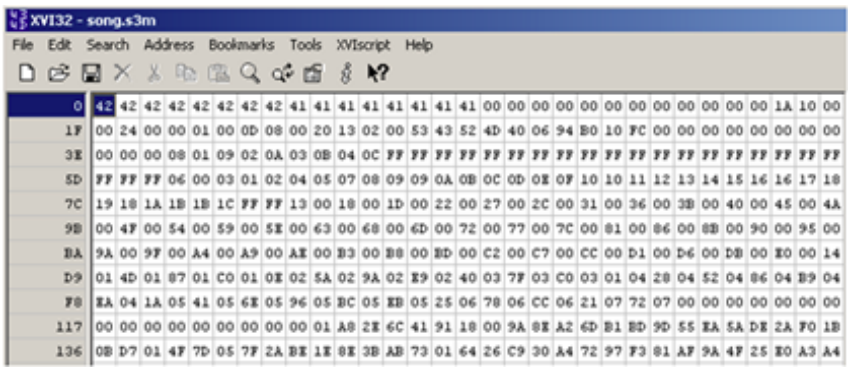
header

The header command will read a file and output the contents of the file in such a way it can be used in an exploit (as a "nice" header).

Mandatory argument :

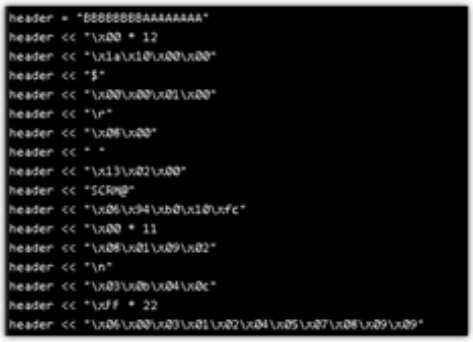
- -f <filename> : the file to read

Example : Let's say you are building a fileformat exploit, and the header of the file (in a hex editor) looks like this :




```
!mona header -f c:\temp\song.s3m
```

The "header" command will return this :



(Note : the output is in ruby format)

heap

The heap command allows you to dump lookaside list and freelists entries on XP (or both). Without arguments, it shows the list of heaps available in the process.

Optional arguments:

- -a <address> : base address of one of the heaps. If you omit this argument, information will be shown for all heaps.
- -t <type> : where type can be lal (for lookaside lists), freelist (for freelist) or all (for both)

help

The help command provides you with more information about a given command.

Mandatory argument :

- <command> : the name of the command

Example : suppose you want to get more information about the "findmsp" command :

```
!mona help findmsp
```

info

This command will show some information about a given pointer

Mandatory argument :

- -a <address>

The output will contain

- information about the pointer
- information about the module it belongs to (or indicate if this is a stack pointer)
- the instruction at that address
- the access level of the page the pointer belongs to

j

This command is just an alias for the "jmp" command. "J" was part of pvefindaddr, so we decided to keep it for backwards compatibility.

jmp

This command will search for pointers that will lead to execute the code located at the address pointed by a given register.

Mandatory argument :

- -r <reg> : where <reg> is a valid register

Default module criteria : skip aslr and rebase modules. The search will include OS modules by default, but you can overrule that behaviour by using the -cm os=false global option.

The command will search for pointers to the following instructions (where reg is the register to jump to, and r32 any register) :

- jmp reg
- call reg
- push reg + ret (+ offsets)
- push reg + pop r32 + jmp r32
- push reg + pop r32 + call r32
- push reg + pop r32 + push r32 + ret (+ offset)
- xchg reg,r32 + jmp r32
- xchg reg,r32 + call r32
- xchg reg,r32 + push r32 + ret (+ offset)
- xchg r32,reg + jmp r32
- xchg r32,reg + call r32
- xchg r32,reg + push r32 + ret (+ offset)
- mov r32,reg + jmp r32
- mov r32,reg + call r32
- mov r32,reg + push r32 + ret (+offset)

jop

This command will search for gadgets that can be used in a JOP (Jump Oriented Programming) payload. The current implementation is very basic, more work will be done in this area over the next few weeks and months.

The code will search for gadgets ending with the following instructions :

- jmp r32
- jmp [esp]
- jmp [esp+offset]
- jmp [r32]
- jmp [r32+offset]

Output will be written to jop.txt

jseh

This command is an alias for "seh". In pvefindaddr, this command would search for safeseh bypass pointers in memory outside of the loaded modules. Mona does all of that using the "seh" command.

modules

The modules command shows information about loaded modules. Without parameters, it will show all loaded modules (but you can use the global options to filter the modules to display)

noaslr

This command is a wrapper around the "modules" command. It will show all modules that don't have aslr enabled by calling the modules function with global option -cm aslr=false

nosafeseh

This command is a wrapper around the "modules" command. It will show all modules that don't have safeseh enabled by calling the modules function with global option -cm safeseh=false

nosafesehaslr

This command is a wrapper around the "modules" command. It will show all modules that don't have safeseh and aslr enabled by calling the modules function with global option -cm safeseh=false,aslr=false

offset

This command will show the distance between 2 addresses, and will show the offset in hex which can be used to make a jump forward or backwards.

Mandatory arguments :

- -a1 : the start address or register
- -a2 : the end address or register

p

This command is an alias for the "seh" command. By default, the "p" command will search in non safeseh and non rebase modules.

p1

This command is an alias for the "seh" command. By default, the "p1" command will search in non safeseh, non aslr and non rebase modules.

p2

This command is an alias for the "seh" command. By default, the "p" command will search in all loaded modules.

pattern_create

This command will produce a cyclic pattern (a.k.a. metasploit pattern) of a given size and write it to pattern.txt

Mandatory argument :

- <number> : where <number> is the desired length of the pattern

Example : if you want a 5000 byte pattern, run this :

```
!mona pattern_create 5000
```

By default, patterns are created using the following 3 character sets :

1. uppercase characters
2. lowercase characters
3. numeric

You can use the following optional arguments to change this default behaviour :

- extended : will add additional chars to the 3rd (numeric) charset : ,.;+=-_!&()#@'{}[]%
- c1 <string> : set your own characters, replacing the first charset used
- c2 <string> : set your own characters, replacing the second charset used
- c3 <string> : set your own characters, replacing the third charset used

If you use a custom pattern charset, don't forget to the same charset options for all related commands (pattern_offset, findmsp, suggest)

pattern_offset

This command will locate the given 4 bytes in a cyclic pattern and return the position (offset) of those 4 bytes in the pattern.

Mandatory argument :

- <search>, where <search> are the 4 bytes to look for. You can specify the 4 bytes as a string, bytes or pointer

pattern_offset will look for the 4 bytes in a normal, uppercase and lowercase pattern, and will also try to find the reversed search bytes in those patterns (in case you reversed the parameter)

Optional arguments : see pattern_create

pc

Alias for pattern_create

po

Alias for pattern_offset

rop

The rop command was written make your life easier when building ROP based exploits. It will do a couple of things :

- search rop gadgets (rop.txt)
- classify the rop gadgets and build a list of suggested gadgets (rop_suggestions.txt)
- find stackpivots (rop_stackpivots.txt)
- attempt to produce 4 entire rop chains (based on VirtualProtect , VirtualAlloc, NtSetInformationProcess and SetProcessDEPPolicy) (output written into rop_chains.txt) (Yes, ROP automation ftw !)

By default, it will look for gadgets in non aslr, non rebase and non OS modules. Again, you can tweak/overrule these criteria using one or more global options.

Optional arguments :

- -offset <value> : set the maximum offset value for RETN instruction (at the end of the gadget). Default value is 40
- -distance <value> : set the minimum value / distance for stack pivots. If you want to set a minimum and maximum value, use -distance min,max (basically use 2 values, separate by comma)
- -depth : set the maximum number of instructions to look back from the ending instruction. Default is 6
- -split : don't save all gadgets into a single rop.txt file, but split up the gadgets in a file per module
- -fast : skip non-interesting gadgets. This might speed up the process, but might make you miss some important gadgets as well
- -end <instruction> : specify a custom ending instruction. By default, the ending instruction is RETN
- -f "file1,file2,file3" : instead of doing an in memory search, read gadgets from one or more rop.txt files created earlier with mona. If you want to use multiple files, make sure to separate them with comma's. It is advised to still attach the debugger to the application when using this option, so mona can locate all information needed to build a rop chain.
- -rva : without this parameter, the rop chain will use absolute addresses (even if the source modules are rebased/aslr enabled). With this option, all addresses will be module base address + rva

Example : generate rop gadgets and automatically attempts to produce ROP chains using 4 particular modules, and list the stackpivots with a distance between 800 and 1200 bytes.

```
!mona rop -m "libglib,libatk,libgdk-win32,libgtk-win32" -distance 800,1200
```

Result :

Note : output is, again, ruby (and thus very much Metasploit friendly)

Based on the number of modules to query, the rop routine can be fast or very slow. The more precise your instructions to mona, the faster and more accurate the results will be.

As an example, take a look at this post : https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/

I used mona to produce a rop chain for one module (17 seconds). The module was not complete, so I had to find 3 more gadgets, which can be found in rop_suggestions.txt and rop.txt. It took me only a few minutes to find them, fix the chain and have a generic/universal dep/aslr bypass rop chain.

A second example to demonstrate the possibilities to build generic rop chains can be found below. Similar to msvcr71.dll, mona produced the majority of the chain. We simply had to fix one issue (to get a desired value into EDX), but using the rop_suggestions.txt file, this took just a few seconds.

Generic rop chain using mfc71u.dll :

```
# MFC71U.DLL generic ROP Chain
# Module: [MFC71U.DLL]
# ASLR: False, Rebase: False, SafeSEH: True, OS: False,
# v7.10.3077.0 (C:\Program Files\CyberLink\PowerProducer\MFC71U.DLL)
# mona.py
rop_gadgets =
        [
        0x7c259e0c,     # POP ECX # RETN (MFC71U.DLL)
        0x7c2512f0,     # <- *&VirtualProtect()
        0x7c2fe7bc,     # MOV EAX,DWORD PTR DS:[ECX] # RETN (MFC71U.DLL)
        0x7c26f014,     # XCHG EAX,ESI # RETN (MFC71U.DLL)
        0x7c2c0809,     # POP EBP # RETN (MFC71U.DLL)
        0x7c289989,     # ptr to 'jmp esp' (from MFC71U.DLL)
        0x7c259e0c,     # POP ECX # RETN (MFC71U.DLL)
        0x7c32b001,     # RW pointer (lpOldProtect) (-> ecx)
        0x7c2de810,     # POP EDI # RETN (MFC71U.DLL)
        0x7c2de811,     # ROP NOP (-> edi)
        0x7c284862,     # POP EAX # RETN (MFC71U.DLL)
        0xfffffffc0,    # value to negate, target 0x00000040, -> reg : edx, via ebx
        0x7c252ea0,     # NEG EAX # RETN (MFC71U.DLL)
        0x7c316b89,     # XCHG EAX,EBX # RETN (MFC71U.DLL)
        0x7c288c52,     # XOR EDX,EDX # RETN (MFC71U.DLL)
        0x7c265297,     # ADD EDX,EBX # POP EBX # RETN 10 (MFC71U.DLL)
        0x41414141,     # EBX
        0x7c284862,     # POP EAX # RETN (MFC71U.DLL)
        0x41414141,
        0x41414141,
        0x41414141,
        0x41414141,     # compensate for RETN 10
        0xfffffdff,     # value to negate, target 0x00000201, target reg : ebx
        0x7c252ea0,     # NEG EAX # RETN (MFC71U.DLL)
        0x7c316b89,     # XCHG EAX,EBX # RETN (MFC71U.DLL) (dwSize)
        0x7c284862,     # POP EAX # RETN (MFC71U.DLL)
        0x90909090,     # NOPS (-> eax)
        0x7c2838ef,     # PUSHAD # RETN (MFC71U.DLL)
        ].pack("V*")
```

If you have produced a generic chain (a chain which will work under all circumstances, bypassing DEP and ASLR), please send it to me (peter [dot] ve [at] corelan [dot] be). If the chain complies with all requirements, we will post it here : Corelan ROPdb.

Short demo on the use of !mona rop, used during my talks at AthCon and Hack In Paris :

View video online here

ropfunc

Ropfunc will attempt to locate pointers to interesting functions in terms of bypassing DEP. Output is written to ropfunc.txt.

The functions it will look for are :

- virtualprotect
- virtualalloc
- heapalloc
- winexec
- setprocessdeppolicy
- heapcreate
- setinformationprocess
- writeprocessmemory
- memcpy
- memmove
- strncpy
- createmutex
- getlasterror
- strcpy
- loadlibrary
- freelibrary
- getmodulehandle

seh

This command will search for pointers to routines that will lead to code execution in a SEH overwrite exploit. By default, it will attempt to bypass SafeSEH by excluding pointers from rebase, aslr and safeseh protected modules. If you specify the optional -all parameter, it will also search for pointers in memory locations outside of loaded modules. Output will be written into seh.txt

The command will search for the following instruction gadgets :

- pop r32 / pop r32 / ret (+ offset)
- pop r32 / add esp+4 / ret (+ offset)
- add esp+4 / pop r32 / ret (+offset)
- add esp+8 / ret (+offset)
- call dword [ebp+ or -offset]
- jmp dword [ebp+ or -offset]
- popad / push ebp / ret (+ offset)

  Tip : If you can’ t find a working pointer, you can still use the “stackpivot” command, using -distance 8,8

stackpivot

This command will search for gadgets that will result in pivoting back into the stack, at a given offset from the current value of ESP. By default, it will look for pivots in non aslr, non rebase and non os modules, and it will group safeseh/non safeseh protected modules together.

Optional arguments :

- -offset <value> : define the maximum offset for RET instructions (integer, default : 40)
- -distance <value> : define the minimum distance for stackpivots (integer, default : 8) If you want to specify a min and max distance, set the value to min,max
- -depth <value> : define the maximum nr of instructions (not ending instruction) in

each gadget (integer, default : 6)

Output will be written to rop_stackpivot.txt

> Tip : if you are looking for a way to take advantage of a SEH overwrite to return to your payload, you can also look for pop r32/pop r32/pop esp/ret, which will bring you back to nseh (but treating the 4 bytes at nseh as a pointer, not as instructions). Look for stackpivots with a distance of 12 to find possible candidates. At nseh, you can simply put a pointer to ADD ESP,4 + RETN to skip the pointer at SEH, and kick off the rop chain placed after the SEH record.

stacks

This command will list stack information for each thread in the application (base, top, size, thread ID)

suggest

This command will automatically run findmsp (so you have to use a cyclic pattern to trigger a crash), and then take that information to suggest an exploit skeleton.

pvefindaddr already suggested a payload layout based on that information, mona takes things one step further. In fact, it will attempt to produce a full blown Metasploit module, including all necessary pointers and exploit layout.

If the findmsp (ran automatically as part of "suggest") has discovered that you control EIP or have overwritten a SEH record, it will attempt to create a Metasploit module.

First, it will ask you to select the "type of exploit". You can choose between

- fileformat
- network client (tcp)
- network client (udp)

(Simply pick the one that is most relevant, you can obviously change it afterwards)

If you have selected "fileformat", it will prompt you for a file extension. If you have selected one of the "network client" types, it will ask you to provide a port number.

Next, mona will ask you if you have a link to an exploit-db advisory. If you are porting an exploit from exploit-db, you can simply provide the full URL (or the exploit-db ID) to the advisory and mona will attempt to extract the exploit name and the original authors from the exploit-db header.

If you don't have a link to an existing exploit-db entry, just leave the "exploit-db" entry empty.

Finally, mona will build a Metasploit file (exploit.rb) based on the information it could gather using the findmsp command (and the information you provided)

While the resulting module may not work right out of the box (you may have to add a header for example, or fix a few things), it will try to build/suggest something useful.

Optional arguments in terms of pattern to look for: see pattern_create. Hint : make sure to use the same charset options as the ones you used to create the cyclic pattern

Example :

http://www.youtube.com/watch?v=k6C7kWBe0Vs

View the video online here

skeleton

This command allows you to create an "empty" metasploit skeleton module for a given type of exploit (fileformat, tcp or udp client), using a cyclic pattern as payload. If you are trying to build an exploit from scratch, this may be a good start. Output will be written to msfskeleton.rb

Optional argument :

- -c <number> : length of the cyclic pattern. Default value is 5000

Metasploit

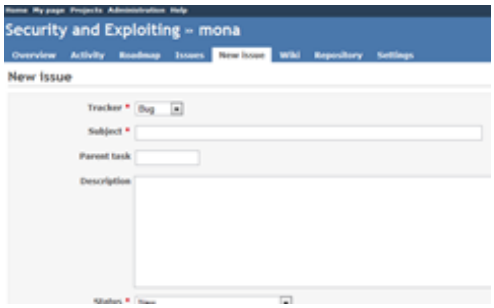
You may have noticed that the output of !mona rop and !mona suggest are very much Metasploit friendly (to say the least). The fact that Corelan Team likes Metasploit and appreciates all the hard work that is being put into the framework is just one of the reasons. We also believe that writing Metasploit modules will (or should) 'force' you to write better and more reliable exploits. Figuring out bad chars, determining available payload size, laying out the most optimal stucture for your exploit, building reliable exploits that work across OSs, bypass DEP/ASLR, etc are just a few requirements for good Metasploit modules. It's clear that it requires more dedication and in some cases more skills than writing a one-shot script with a hardcoded payload that happens to work because you got lucky :)

Anyways, If mona.py can assist you in some parts of your exploit development experience, then we accomplished our goal.It's not a perfect tool. It still requires you and your skillset to direct it and to make it return what you are looking for.

Remember : don't forget to submit your exploit module to Metasploit. (msfdev@metasploit.com)

Discovered a bug, suggest new features / want to contribute ?

Head over to https://redmine.corelan.be and register a user account. When your user account has been approved, you will have the ability to create bug reports or feature requests.



Navigate to https://redmine.corelan.be/projects/mona/issues/new to create a new ticket. Set the "tracker" field to either "bug" or "feature", and make sure to use a meaningful subject and a verbose description.

The team really appreciates and welcomes all forms of contribution, so if you want to submit your changes, make sure to use the latest trunk version of mona, implement your changes, and create a "diff" or "patch" file. Attach the patch file to the redmine ticket and we'll test / implement it.

You can see all open tickets here : https://redmine.corelan.be/projects/mona/issues/

> Again, as mentioned earlier, if you have been able to produce a generic ROP chain, feel free to send it over. if the chain is generic, we'll host it on the Corelan ROPdb page

Want to learn how to write exploits ?

Take a look at the Corelan Team exploit writing tutorials : https://www.corelan.be/index.php/category/security/exploit-writing-tutorials/ (start at the bottom of the list and work your way up).

Alternatively, if you prefer class based training, you can check out the Corelan Live Win32 Exploit Writing Bootcamp training here : https://www.corelan-training.com/index.php/training/corelan-live/

Need help ?

If you have questions, you can post them in our forums : https://www.corelan.be/index.php/forum/writing-exploits/

If you prefer to talk to Corelan members directly, or to other people in the community, connect to IRC (irc.freenode.net) register a nickname, and join channel #corelan.

Mona.py on the web

If you want to share your experiences with mona.py, publish a post or document demonstrating the use of one or more mona.py features, let us know and we'll be more than happy to link back to your site.

if you like mona.py, tell it to the world. If you don't like mona.py, let us know too.

thanks

Pictures used with permission: digitalart / FreeDigitalPhotos.net

Metasploit logo used with permission. Metasploit is a registered trademark of Rapid7 LLC

thanks fancy for creating the video's

Related Posts:

- Jingle BOFs, Jingle ROPs, Sploiting all the things… with Mona v2 !!
- Mona 1.0 released !
- Exploit writing tutorial part 11 : Heap Spraying Demystified
- Universal DEP/ASLR bypass with msvcr71.dll and mona.py
- Heap Layout Visualization with mona.py and WinDBG
- Metasploit Bounty – the Good, the Bad and the Ugly
- Starting to write Immunity Debugger PyCommands : my cheatsheet
- Exploit writing tutorial part 6 : Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR
- Exploit writing tutorial part 10 : Chaining DEP with ROP – the Rubik's[TM] Cube
- Happy 5th Birthday Corelan Team

Posted in 001_Security, Exploit Writing Tutorials | Tagged automation, corelan, corelanc0d3r, documentation, exploit, exploit development, immunity, immunity debugger, metasploit, mona, mona.py, pattern, plugin, pycommand, python, rapid7, rop, rop automation, rop gadget, suggest

One Response to *mona.py – the manual*

- *raylook* says:
  February 15, 2012 at 21:31

  Awesome work , thans Boss :)

Search

Donate

Want to support the Corelan Team community ? Click here to go to our donations page.

Want to donate BTC to Corelan Team?

Your donation will help funding server hosting.

Corelan Team Merchandise

You can support Corelan Team by donating or purchasing items from the official Corelan Team merchandising store.

Corelan Live training

Since 2011, Corelan GCV has been teaching live win32 exploit dev classes at various security cons and private companies & organizations.

You can read more about the training and schedules here

Stay posted

- Subscribe to posts via email
- Follow me on twitter

Corelan on IRC

You can chat with us and our friends on #corelan (freenode IRC)

Actions

- Register
- Log in
- Entries RSS
- Comments RSS
- WordPress.org

Categories

Categories Select Category