MALWARE ANALYSIS USING PYMAL & MALPIMP

Amit Malik

Idiot @SecurityXploded Research Group

Researcher @Fireeye Labs

Agenda

- Tools introduction
- Malpimp
  - Configuration file
  - Tracing
  - Demo
- Pymal
  - Features and functions
  - Demo
  - More examples

Tools Introduction

- Malpimp – based on pydbg (pure python debugger)
  - API tracing, using configuration file you can configure the tool according to your needs.
  - Light weight and very easy, just serves the purpose
- PyMal – Python interactive shell for malware analysis
  - Based on three powerful pure python tools: pefile, pydbg, volatility
  - Pydbg != debugger in pymal
  - Process manipulation & live memory analysis.
  - Some powerful features like hook detection (proprietary), Injected code detection.
  - And full python support ☺

Malpimp

```
C:\Documents and Settings\Administrator\Desktop\Malpimp>malpimp.exe

[*] Author: Amit Malik (m.amit30@gmail.com)
[*] http://www.securityxploded.com

[*] Usage: malpimp.exe <exe_file> <address>
[*] Usage: malpimp.exe -p pid
[*] example: malpimp.exe sample.exe 0
[*] example: malpimp.exe -p 540

C:\Documents and Settings\Administrator\Desktop\Malpimp>malpimp.exe ..\procexp.exe 0
Setting breakpoints on the exports of dll: C:\WINDOWS\system32\kernel32.dll
Setting breakpoints on the exports of dll: C:\WINDOWS\system32\WS2_32.dll
Setting breakpoints on the exports of dll: C:\WINDOWS\system32\ADVAPI32.dll
Setting breakpoints on the exports of dll: C:\WINDOWS\system32\Secur32.dll
Setting breakpoints on the exports of dll: C:\WINDOWS\system32\msvcrt.dll
Setting breakpoints on the exports of dll: C:\WINDOWS\system32\MPR.dll
```

- Second argument on command line is the address from where we want to start tracing. Zero means entry point.
- Configuration file
  - Fine control over tracing
  - Loop detection based on return address – believe me this is really a beautiful feature, I saw couple of big heavy commercial products that are suffering on it. Also this technique is unique to this tool and it greatly improves the tracing time. [Depending on your configs it is capable to reduce tracing time from 2 hours to 2 seconds with almost same information.]
  - Inclusion and exclusion policies

Malpimp Configuration

- TraceInclude – Apply hooks only on these DLLs or APIs, if this field have some value either in DLL or API then TraceExclude will be ignored.
  - Syntax : for DLL: simple dll name like : kernel32.dll, user32.dll etc. , for API: DLL!API name e.g: kernel32!VirtualAlloc
- TraceExclude works only when we have all fields empty in TraceInclude policy.

```
# Hooking policies
# TracingExclude - During hooking exclude the DLLs and APIs mentioned in this policy.
# TracingInclude - During hooking only hook the DLLs and APIs mentioned in this policy, If this policy have values in its
# fields then TracingExlude entries will be ingnored.

## seperate the multiple values using comma (,)
# For API just use API name. eg: LoadLibraryA
[TracingExclude]
DLLs = USER32.dll,GDI32.dll,ntdll.dll,PSAPI.DLL,REGAPI.dll,WS2HELP.dll,ole32.dll,USERENV.dll,AUTHZ.dll,MSASN1.dll,RPCRT4.dll
API =

# For API use DLL!API syntax eg: kernel32!LoadLibraryA
[TracingInclude]
DLLs =
API =
```

Malpimp Configuration cont.

- Loop detection settings
- Report logging addresses – set start and end addresses for logging, it allow us to log only important trace. For example: we want to trace API calls from newly allocated region or from a specific DLL address space.

```
# control the execution in a better way.
[Additional]
####
# solveloop (yes/no) - remove the hook from the apis that are called with same
solveloop = yes
apithreshold = 5

# Arguments to application
args = None

####
# Everything between these addresses will be logged into the trace file.
# change according to your requirements
# default: 167772160 (0x0A000000)
loggingaddrmax = 1879048192
loggingaddrmin = 0

#### End of File ####
```

- You can also attach malpimp to any running process using the following command
- Malpimp.exe –p <process id>

Demo

- Bamital sample Trace!

Limitations

- Based on a debugger so debugger detection techniques can easily detect.
- Unreliable for heavy applications with hooks on lots of DLLs.

Pymal

- Python interactive shell for malware analysis
- Wrapper functions around pefile, pydbg and volatility
- Helpful in active process manipulation and live memory analysis
- Interactive shell with full python support so additional modules can be easily imported, operations on data are much easier.
- Tab completion, use object “pm” to see pymal methods.
- Uses distorm3 library for disassembly
- Some features like hook detection and injected code detection are awesome.
  - Please read the PyMal disclaimer carefully before using its code/technique/theory into your tools.

Pymal Functions

- Only the important ones.
- Process related:
- DumpModule – Dump the loaded dll from memory to disc (it will fix the headers automatically)
- DumpMem – Dump exe image from memory to disc (no header fix)
- DumpPidFix – Dump exe image from memory and fix the headers
- DumpMemToPE – Dump the PE file from memory (just need an address but it is your responsibility to verify the valid image at that address)

- OpenProcess, ReadMemory, WriteMemory, ShowProcesses, ShowModules, ShowThreads etc.

- FindDll – search for a dll in all processes.
- FindProcess – retrieve pid using exe name.

Pymal Functions cont.

- Pefile related functions
- LoadPE – load the exe file
- ImageBase – get image base address
- EntryPoint – get entry point address
- Sections, ImportTable, ExporTable etc.
- You can access original pefile and pydbg objects using pm.pe and pm.dbg

- Advanced functions
- ScanModInPid – scan a dll in process for hooks
- ScanPidForMod – scan all loaded modules for hooks in a process.
- FindInjectedCode – find the RWE allocations in the process

- Others
- Disasm* - show disassembly

- In case of confusion use help(pm.function_name) eg: help(pm.Disasm)

Pymal Demo

- Pymal Demo
- Online users: http://nagareshwar.securityxploded.com/2013/08/28/bamital-analysis-using-malpimp-and-pymal/

Pymal – more examples!

- Helpful in many scenarios
- Read/write remote process memory, helpful mainly when one process injects code in other processes
- Monitor addresses or values at addresses without using or attaching a debugger.
- Read data from process and apply your logics from a single shell eg: xor data, calculate hash etc.
- Import you own modules
- Etc. etc.

Thank You!