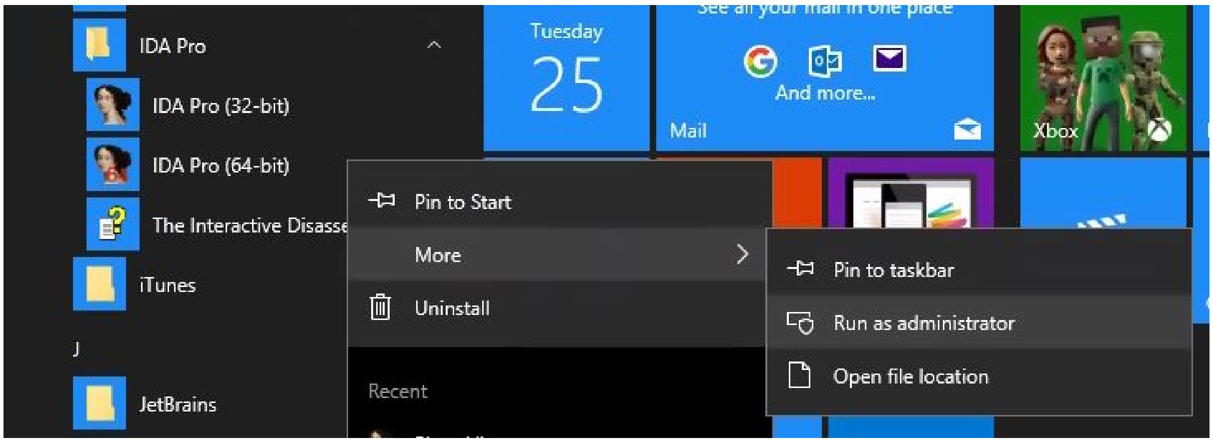
Shredder: Breaking exploits through API SPecialization

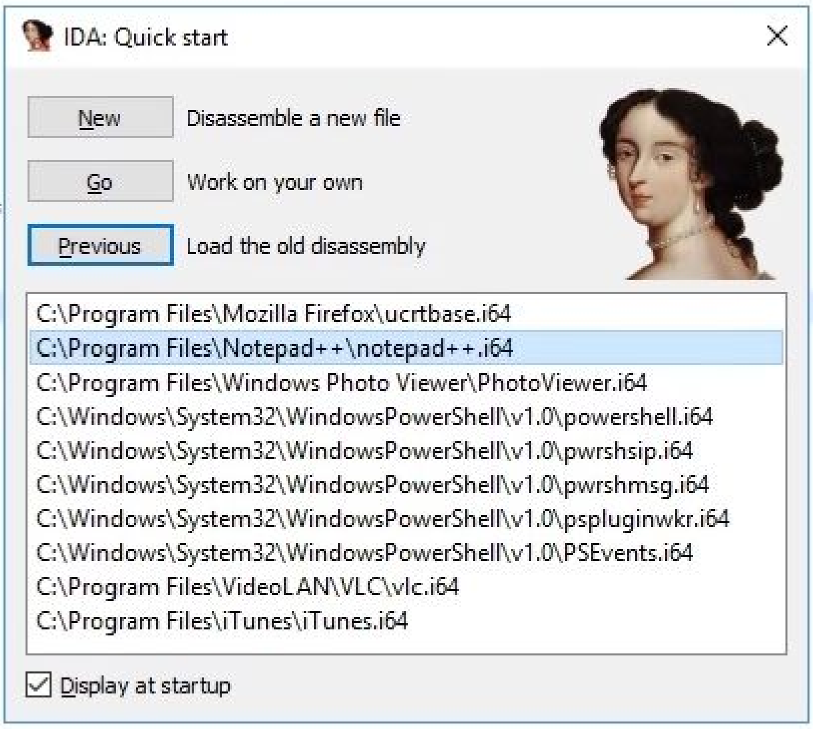
-Shachee Mishra (shmishra [AT] cs.stonybrook.edu)

System Requirements:
Windows 10
IDA Pro 64 bit

1. Open Ida64 - Always run as administrator



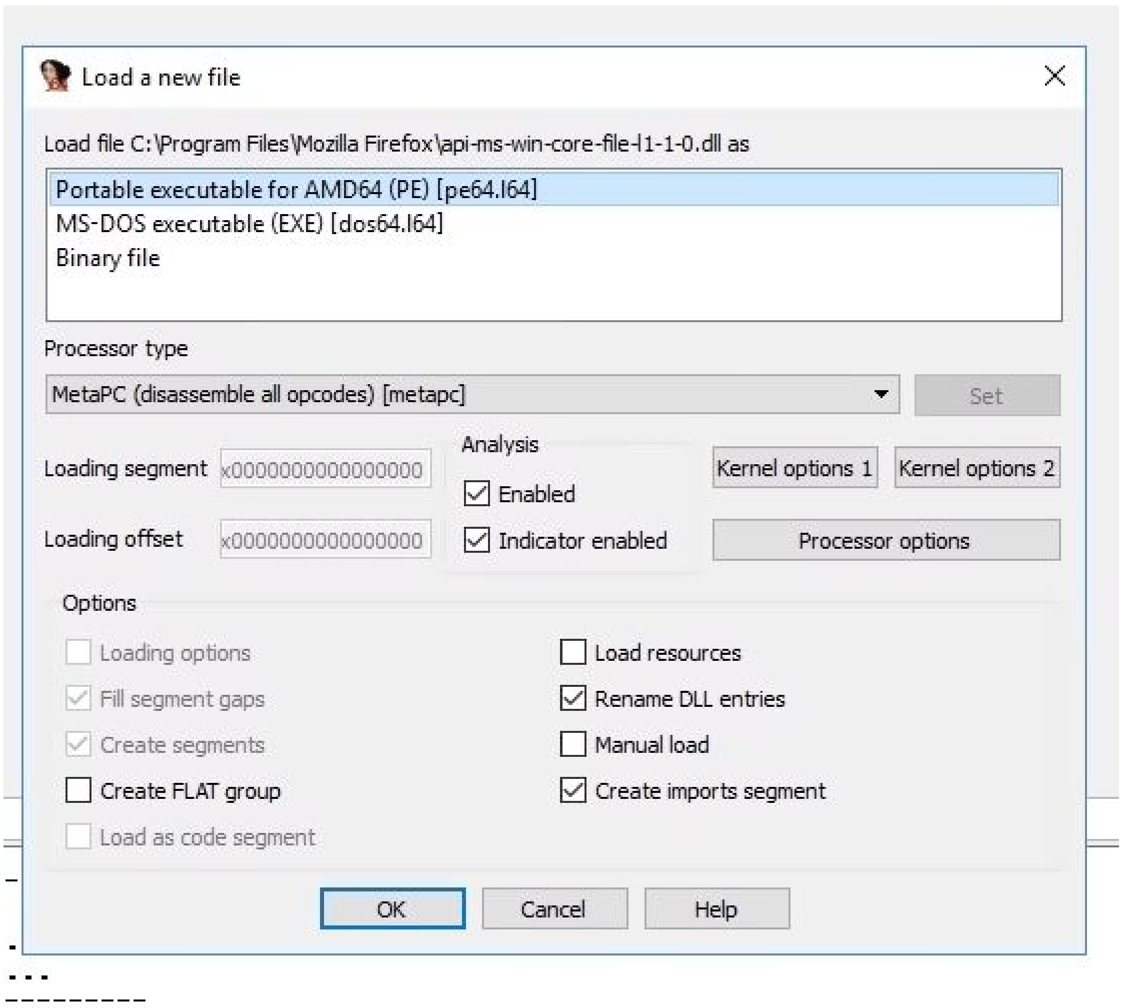
2. New file disassembly
3. Select - notepad++.exe (or any other .exe/.dll)



```
C:\Program Files\Mozilla Firefox\ucrtbase.i64
C:\Program Files\Notepad++\notepad++.i64
C:\Program Files\Windows Photo Viewer\PhotoViewer.i64
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.i64
C:\Windows\System32\WindowsPowerShell\v1.0\pwrshsip.i64
C:\Windows\System32\WindowsPowerShell\v1.0\pwrshmsg.i64
C:\Windows\System32\WindowsPowerShell\v1.0\pspluginwkr.i64
C:\Windows\System32\WindowsPowerShell\v1.0\PSEvents.i64
C:\Program Files\VideoLAN\VLC\vlc.i64
C:\Program Files\iTunes\iTunes.i64
```

[✓] Display at startup

4. Load file <file_name> as: Portable executable for AMD64 [pe64.l64]



Load a new file

Load file C:\Program Files\Mozilla Firefox\api-ms-win-core-file-l1-1-0.dll as

Portable executable for AMD64 (PE) [pe64.l64]
MS-DOS executable (EXE) [dos64.l64]
Binary file

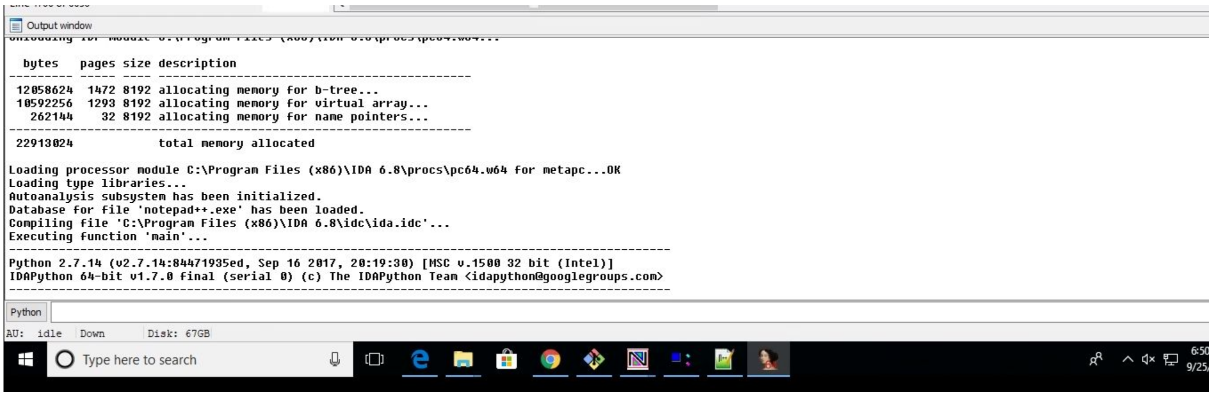
Processor type: MetaPC (disassemble all opcodes) [metapc]

Loading segment: x0000000000000000
Loading offset: x0000000000000000

Analysis
- [✓] Enabled
- [✓] Indicator enabled

Options
- [ ] Loading options
- [✓] Fill segment gaps
- [✓] Create segments
- [ ] Create FLAT group
- [ ] Load as code segment
- [ ] Load resources
- [✓] Rename DLL entries
- [ ] Manual load
- [✓] Create imports segment

5. Confirm the bottom left corner of Output window says “Python”



Output window

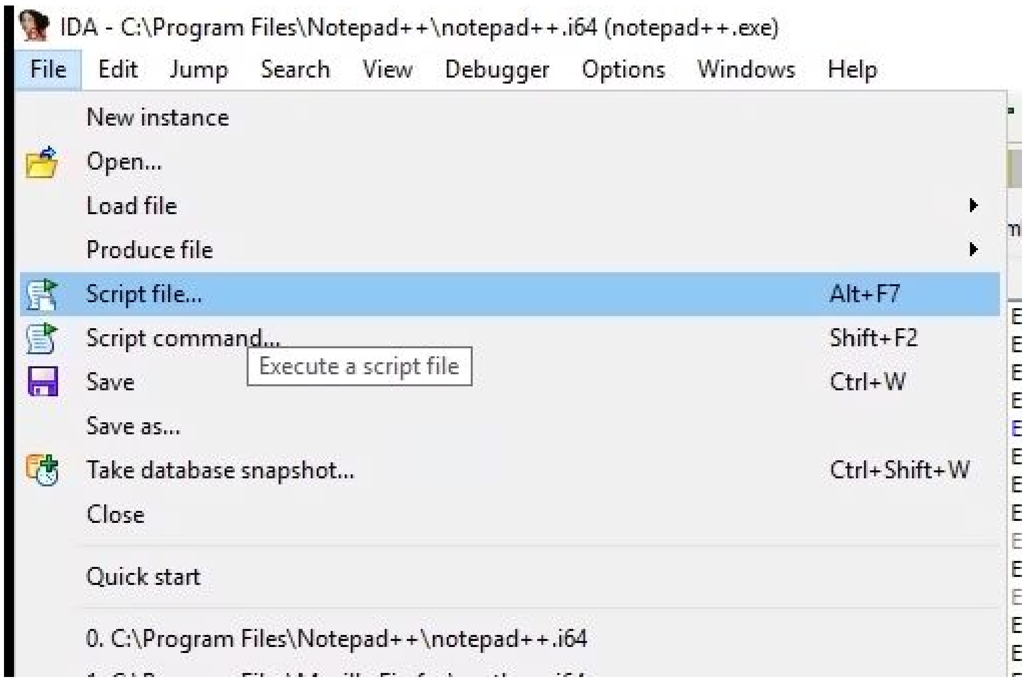
```
 bytes   pages size description
--------- ----- ---- --------------------------------------------
 12058624  1472 8192 allocating memory for b-tree...
 10592256  1293 8192 allocating memory for virtual array...
   262144    32 8192 allocating memory for name pointers...
-----------------------------------------------------------------
 22913024            total memory allocated

Loading processor module C:\Program Files (x86)\IDA 6.8\procs\pc64.w64 for metapc...OK
Loading type libraries...
Autoanalysis subsystem has been initialized.
Database for file 'notepad++.exe' has been loaded.
Compiling file 'C:\Program Files (x86)\IDA 6.8\idc\ida.idc'...
Executing function 'main'...
-----------------------------------------------------------------------------------------
Python 2.7.14 (v2.7.14:84471935ed, Sep 16 2017, 20:19:30) [MSC v.1500 32 bit (Intel)]
IDAPython 64-bit v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
-----------------------------------------------------------------------------------------
```

Python

AU: idle   Down   Disk: 67GB

6. Open script file: File -> Script file...


IDA - C:\Program Files\Notepad++\notepad++.i64 (notepad++.exe)
File
Edit
Jump
Search
View
Debugger
Options
Windows
Help
New instance
Open...
Load file
Produce file
Script file... Alt+F7
Script command... Shift+F2
Execute a script file
Save Ctrl+W
Save as...
Take database snapshot... Ctrl+Shift+W
Close
Quick start
0. C:\Program Files\Notepad++\notepad++.i64

7. Load script1.py -File -> Script file -> select Script1.py
   This gives the number of used API Calls from kernel32.dll, advapi.dll, user32.dll and ole32.dll

8. Load File -> Script file -> select Script2.py -
   Lists the name and number of critical functions imported by this executable.
   It also provides the individual and the total number of references of critical functions excluding CloseHandle().

9. Load File -> Script file -> Script3.py
   Running this, we get list of all the critical functions and the arguments inferences.
   Every I in the final result is an immediate value. Every “D” is a displacement, which is an offset and the value at the offset can be looked up.
   Every “R” is a register whose value could not be found.
   Get all call instances and their args and Perform backward analysis and get all traced args.
   For easier understanding and clear results: In the current script 3, I have only looked for arguments to VirtualAlloc() and VirtualProtect() and commented out the rest of the functions.

   It is clearer to see the results in this case.

10. ropEntry.txt:
    This file contains the specific API name and arguments used by attacks mentioned in Table 6 in Appendix in the paper. Each of these use special API functions to change memory permissions to launch shellcodes.

11. Folder named “Shellcodes” contain all the API calls extracted from in-the-wild shellcodes. I have not added the .exe files for shellcodes (they usually get flagged by virus detectors) - I can provide them, if needed. To get the list of APIs from shellcodes we use ‘apitracker.exe’. This is run using getAPIUsage.py.

12. ROPPayloads.xls contains ROP payloads. Each sheet contains the payload if broken by Shredder or not. Also, it contains the comparison with code stripping.

13. Shellcodes.xls contains the shellcodes that we tested. Each sheet contains the payload and the number of functions in the payload broken by Shredder vs those broken by Code Stripping.