ropfind, linux_ropfind - Volatility contest 2019

Submitters: Or Chechik and Inon Weber

Motivation:
As a long-time volatility users we always wanted to contribute to the Framework.
Like many security researchers, we also observed a shift in the usage of ROP in attackers techniques which are not just for exploitation. We thought code reuse attacks deserve detection plugins in volatility. Basically their purpose is to find remnants of ROP chains in Windows and Linux memory dumps.

Let's start by overviewing some examples of ROP in attackers techniques:

-ROP adopted as an execution technique in process injection techniques

(https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All.pdf)
Check Stackbombing, Ghostwriting and Atombombing and there are a lot more examples.

-ROP adopted as an obfuscation method in malware

(https://www.blackhat.com/docs/us-15/materials/us-15-Xenakis-ROPInjector-Using-Return-Oriented-Programming-For-Polymorphism-And-Antivirus-Evasion.pdf)

-ROP adopted as a stealth payload manifestation in malware

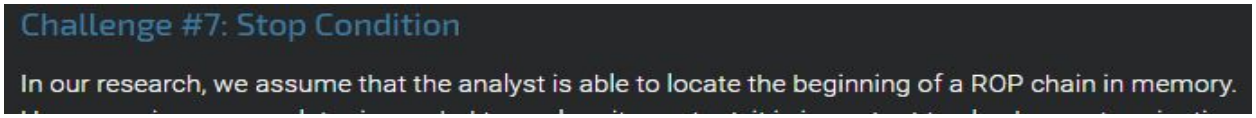
Chuck, name of the persistent ROP rootkit proposed by
S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent data-only malware: Function hooks without code. In Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS), Feb. 2014. [59] Websense S

http://blog.opensecurityresearch.com/2013/10/analysis-of-malware-rop-chain.html
(This is the malware we used to check against in the windows plugin)

-ROP is used as a DEP bypass in most exploits.
There are enough examples documented.

On top of that, we wanted to cover the shortcomings of Malfind (documented a lot in past submissions) and other current plugins by offering other ways to detect malicious code execution.

*There could be confusion and some might say ROPMEMU already does exactly that. ROPMEMU is an excellent framework submitted last year which was built to analyze, dissect and decompile complex code-reuse attacks - it assumes the analyst already found the ROP chain, which by itself isn't a trivial matter at all.



Our plugins purpose is to find the ROP gadgets. We built it to work in Linux, Windows in both x86 and x64 (supports wow64 too!).
The nature of the plugin allows it to find packed malware, malware loaders, malware payloads that utilize ROP and exploits (less likely but still a possibility).

With that said, we would like to introduce ropfind and linux_ropfind.

ropfind - Windows

This plugin's purpose is to find remnants of ROP gadgets in Windows memory dumps (x86 and x64) by extracting threads sp registers and scanning the corresponding stack. The plugin scans small allocations as well to cover pivoted stacks that the sp register has long left.

This plugin will enumerate all the threads. For each thread found it will extract the SP register from the _KTRAP_FRAME and the StackBase and StackLimit from the PEB.

First, it will check if “stack pivoting” has occurred. If the SP is not within the range of StackBase and StackLimit then the plugin will announce stack pivoting for that thread and will scan that VAD allocation as a stack for ROP gadgets.

*Another check we have for stack pivoting is by trying to find the pointer for the symbol ntdll!BaseThreadInitThunk on the stack of every thread - if it is not present we assume stack pivoting.

If no stack pivoting was found the scanning will occur on the regular thread stack.
The scanning of ROP gadgets is done by scanning the stack found and searching for potential suspicious pointers.
In addition, we chose to scan for ROP gadgets in all small allocations in the dump as the sp register might have long left the pivoted area or there might be chains/structs used by the malware in those. The cases we search for in by searching those small allocations are malware that keeps their ROP chains in dynamic memory, malware/loader/packers that use structs to store their pointers to critical functions and exploits that left their pivoted area behind or used heap spray.

The plugin checks whether each pointer points to an executable region, its disassembly contains code control instructions (code control instructions in our logic are assembly instructions that can affect the IP register, for example, ret, call, jmp, syscall and more..) and it was not placed on the stack using the ‘CALL’ instruction. Another option for a pointer to be an ROP gadget by our logic is if it’s an address to a critical function. (Critical functions are library functions we chose that are essential in exploitation or malware - for example VirtualProtect).

Of course, searching using only those criteria will yield false positives so we thought of a way we could filter most pointers by thinking of the legit scenarios and filtering for gadgets instructions we thought that just doesn't make sense to be used in ROP chains.

The options we thought of for pointers to be on legitly placed on the stack and point to an executable code are: Return addresses, callback parameters, variables/structs/arrays containing pointers to functions prologues.

As users, we also know there could be unwanted false positives in any plugin so we thought of a way for analysts to easily whitelist gadgets of their choosing using the gadget content, symbol and offset. You can also whitelist gadgets using symbols only like we did for 'ntdll!KiFastSystemCallRet' and 'ntdll!KiFastSystemCall' which were common false positives in our testing. plus, It's really easy to use.

The output of the plugin consists of the following view for each gadget found.

```
PID: 1028  Process name: AcroRd32.exe TID: 3048
Stack address: 0x3d1f040 Gadget address: 0x777014eb
Gadget VAD: 0x8c271898 Gadget VAD permissions: PAGE_EXECUTE_WRITECOPY
Gadget Symbol: None+0x14eb
Gadget address disassembly:

0x777014eb c3              RET
0x777014ec 90              NOP
0x777014ed 90              NOP
0x777014ee 90              NOP
0x777014ef 90              NOP
0x777014f0 90              NOP
0x777014f1 3b0df4817777    CMP ECX, [0x777781f4]
0x777014f7 0f              DB 0xf
0x777014f8 8565fa          TEST [EBP-0x6], ESP

Stack Start: 0x3610000 Stack End: 0xfe10fff Stack Permissions: PAGE_READWRITE
Stack view:
0x3d1f030         0x75d43c01 kernel32!LoadLibraryW
0x3d1f034         0x777014eb clbcatq+0x14eb
0x3d1f038          0x8080048
0x3d1f03c         0x75d3ba46 kernel32!Sleep
0x3d1f040         0x777014eb clbcatq+0x14eb
0x3d1f044           0x36ee80
0x3d1f048           0x3a0043
0x3d1f04c           0x55005c
0x3d1f050           0x650073
```

* As we are looking for remnants of ROP chains we thought the output should be per gadget and not a chain. The analyst should be the one to conclude if the output is a chain/malicious or not.

If the plugin finds pivoted stack the output will be as so:

```
Volatility Foundation Volatility Framework 2.6.1
INFO    : volatility.debug    : Possible Stack pivoting found at 0x000000000807f9fc when the the Stack range is 0x0000000001dae000-0x0000000001db0000 in PID 1028 and TID 3048
```

This was the case for the malware we tested.

As we can see the plugin prints in which process and thread the gadget was found. The stack address it found, the VAD properties for that stack allocation (if the stack was made executable we will see it here), the VAD in which the gadget code resides on, the gadget address 0x777014eb, the gadget disassembly RET and the stack view of the stack the gadget is located on, with its surrounding values.
The gadget pointer 0x777014eb is located right in the middle at 0x3d1f040.
In this stack view, we can also see something that resembles symbols to the associated values on the stack (like in a debugger). We parsed the exports of the loaded modules to supply function names and offsets for easy analysis. For example, we can see the symbols for the functions of LoadLibraryW and Sleep.
In the stack view, we can see the possible chain related to that gadget, as well as arguments passed to functions. Using this view, adjacent gadgets views and volshell the Analyst can understand if the gadgets are connected and if it's truly an ROP chain, a Malware IAT or a false positive.
Using this stack view and volshell we can also resolve arguments to their content in memory. We could do it for anything - strings, objects, Structs, vtables etc…

For example, in this gadget, we can see the pointer 0x8080048 passed to LoadLibraryW.
Checking that address in volshell, we can see it's a DLL path, which seems to be located in the %appdata% directory.

```
>>> db(0x8080048)
0x08080048  43 00 3a 00 5c 00 55 00 73 00 65 00 72 00 73 00   C.:.\.U.s.e.r.s.
0x08080058  5c 00 4f 00 52 00 43 00 48 00 45 00 43 00 7e 00   \.O.R.C.H.E.C.~.
0x08080068  31 00 5c 00 41 00 70 00 70 00 44 00 61 00 74 00   1.\.A.p.p.D.a.t.
0x08080078  61 00 5c 00 4c 00 6f 00 63 00 61 00 6c 00 5c 00   a.\.L.o.c.a.l.\.
0x08080088  54 00 65 00 6d 00 70 00 5c 00 4c 00 32 00 50 00   T.e.m.p.\.L.2.P.
0x08080098  2e 00 54 00 00 00 00 00 66 66 66 66 66 66 66 66   ..T.....ffffffff
0x080800a8  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66   ffffffffffffffff
0x080800b8  66 66 66 66 66 66 66 66 66 66 66 66 66 66 66 66   ffffffffffffffff
```

Checking Dlllist plugin for that AcroRd32.exe process we can see that this DLL is already loaded.

```
0x73570000    0x12000        0x1 2019-09-12 19:22:56 UTC+0000   C:\Windows\system32\DHCPCSVC.DLL
0x73590000     0xd000        0x1 2019-09-12 19:22:56 UTC+0000   C:\Windows\system32\dhcpcsvc6.DLL
0x74900000     0x9000        0x1 2019-09-12 22:12:34 UTC+0000   C:\Windows\system32\VERSION.dll
0x6d800000    0x16000      0x3ea 2019-09-12 22:12:35 UTC+0000   C:\Windows\system32\MAPI32.DLL
0x603f0000    0x50000        0x1 2019-09-12 22:12:35 UTC+0000   C:\Users\ORCHEC~1\AppData\Local\Temp\L2P.T
```

Using Dlldump we can retrieve that DLL and if we upload it to VirusTotal we'll find it to be a malware. The usage of reuse attacks makes memory analysis a lot harder.

The entire dropper code was manifested using a huge ROP chain and could go entirely undetected in any of the volatility plugins. There's no injection or suspicious executable allocations, just a big RW allocation with pointers.
Malfind, for instance, did not detect anything in the mentioned AcroRd32.exe process.

Inspecting the entire output for that AcroRd32.exe process we can see it has a lot of gadgets. The gadgets seem repetitive which might indicate usage of heap spray. Although that pivoted stack allocation is huge and consists of a lot of gadgets the plugin runs relatively fast as we implemented a cache mechanism for symbols and found gadgets.

Using our plugins, analysts should find hints for reuse attacks that they could use to supply to other tools \ volatility plugins for further investigation. Volshell, vaddump, vadinfo, Dlllist, Dlldump, procdump, yarascan, IDA Pro and possibly many more could be very handy in those situations.

ropfind_linux

The linux plugin works almost the same just with different structs obviously..

```
PID: 11417  Thread name:vuln
Stack address: 0x7fffffffdf38 Gadget address: 0x7ffff7a16a4d
Gadget VMA name: /lib/x86_64-linux-gnu/libc-2.19.so Gadget VMA permissions: r-x
Gadget Symbol: h_errno+0x29b1
Gadget address disassembly:

0x7ffff7a16a4d ffe4            JMP RSP
0x7ffff7a16a4f 1b7fce          SBB EDI, [RDI-0x32]
0x7ffff7a16a52 dbce            FCMOVNE ST0, ST6
0x7ffff7a16a54 46f0e4a9        IN AL, 0xa9
0x7ffff7a16a58 6656            PUSH SI
0x7ffff7a16a5a a7              CMPSD
0x7ffff7a16a5b 15              DB 0x15
0x7ffff7a16a5c 13              DB 0x13

Stack VMA: [stack] Stack Permissions: rwx
Stack view:
0x7fffffffdf18     0x7ffff7a2d3b8 libc-2.h_errno+0x1931c
0x7fffffffdf20               0x11
0x7fffffffdf28     0x7ffff7a50bed libc-2.labs+0xd
0x7fffffffdf30     0x7ffff7ad6c95 libc-2.getppid+0x5
0x7fffffffdf38     0x7ffff7a16a4d libc-2.h_errno+0x29b1
0x7fffffffdf40 0x6800000100c48148
0x7fffffffdf48 0x5c11686601fea8c0
0x7fffffffdf50 0x6a106a2a6a026a66
0x7fffffffdf58 0x485e5f026a016a29
```

Let us explain the output of the linux plugin.
Same as the windows plugin, this is a view for a single ROP gadget.
The gadget is JMP RSP, its pointer is 0x7ffff7a16a4d. The pointer to the gadget is stored at the found stack at 0x7fffffffdf38. We can see this is the original stack and that is was made executable by looking at the found stack VMA properties. The name of the VMA is [stack] and the permissions are rwx. It seems like this is the last gadget of the chain as we can see the shellcode on the stack view and the JMP RSP is right behind it.

```
PID: 11417  Thread name:vuln
Stack address: 0x7fffffffdee8 Gadget address: 0x400623
Gadget VMA name: / Gadget VMA permissions: r-x
Gadget Symbol: gets+0x623
Gadget address disassembly:

0x400623 5f               POP RDI
0x400624 c3               RET
0x400625 66662e0f1f840000000000 NOP WORD [RAX+RAX+0x0]
0x400630 f3c3             RET
0x400632 00               DB 0x0

Stack VMA: [stack] Stack Permissions: rwx
Stack view:
0x7fffffffdec8 0x4141414141414141
0x7fffffffded0 0x4141414141414141
0x7fffffffded8 0x4141414141414141
0x7fffffffdee0 0x4141414141414141
0x7fffffffdee8           0x400623 /.gets+0x623
0x7fffffffdef0     0x7ffffffde000
0x7fffffffdef8     0x7ffff7a386f5 /.__gconv_get_alias_db+0x10a5
0x7fffffffdf00           0x21000
0x7fffffffdf08     0x7ffff7a15b8e /.h_errno+0x1af2
```

This is another gadget of the same chain. We can see some A's in hex at the stack view which is known for filling buffers in exploits.
Using volshell as noted before can be used to determine which gadget will run next and what lead to this gadget. Because of the way the plugins work, the gadgets of the same chain should be printed consecutively but as stuff could be overwritten in memory dumps it's not always the case.
Running netscan on that PID will reveal it's connecting back to an IP on port 4444 - which is consistent with metasploit reverse shell default port.

```
TCP      192.168.254.138 :37078 192.168.254.1   : 4444 SYN_SENT                    vuln/11417
```

Download link for the Plugins, dumps, profiles and output examples:
https://drive.google.com/file/d/1B1aXzcZS-OknXAPqarGIhvJg3qmwQVO_/view?usp=sharing

Github: https://github.com/orchechik/ropfind

Usage:
To use the plugins, place them under the plugins folder and they will be available right away.
The Linux plugin has a profile file for the dump attached. It is named "Ubuntu1404.zip", place it under the linux directory.

Reasons we think our plugins should win:

1. Those are detection plugins and not investigation ones (they could be but it’s unlikely). In our opinion, detection plugins are more crucial as they give you a place to start looking at. You won’t investigate anything if you don't have some kind of lead.
2. The plugins are innovative, no one has written or implemented anything similar as we know.
3. Strengthen the framework detection capabilities - As documented before by many others, the framework has its weak points and we want to provide another angle of detection. Also, the framework doesn't address code reuse attacks at all.
4. The plugins are relevant to the shift of attackers' techniques as documented above.
5. We worked hard on extra features to make the analysis easier for the analyst (symbols, stack view, caching..)
6. The plugins work on most prevalent OS’s today - Windows, Linux on both x86 and x64 (on Windows x64 - WOW64 is supported).

Thanks for the time you took to read and evaluate our submission, we enjoyed writing the plugins and delving into volatility source code.