

# PIC Your Malware!



# Whoami

- Ben Heimerdinger and Sebastian Feldmann
- Specialized on Offensive Tooling / Evasion
- Code White Red Team (Ulm/Mannheim, Germany)

# Syllabus

- Concepts for fileless malware
  - Artifacts popular in memory PE loaders leave (Reflective DLL, Donut)
- Leveraging position independent code (PIC) to avoid these artifacts
- Tool release: Lsass dumping without ProcessAccess event (as PIC)
- Solving some tedious problems of creating PIC
- Protecting tools with self-decrypting PIC
- Pipelining the build process

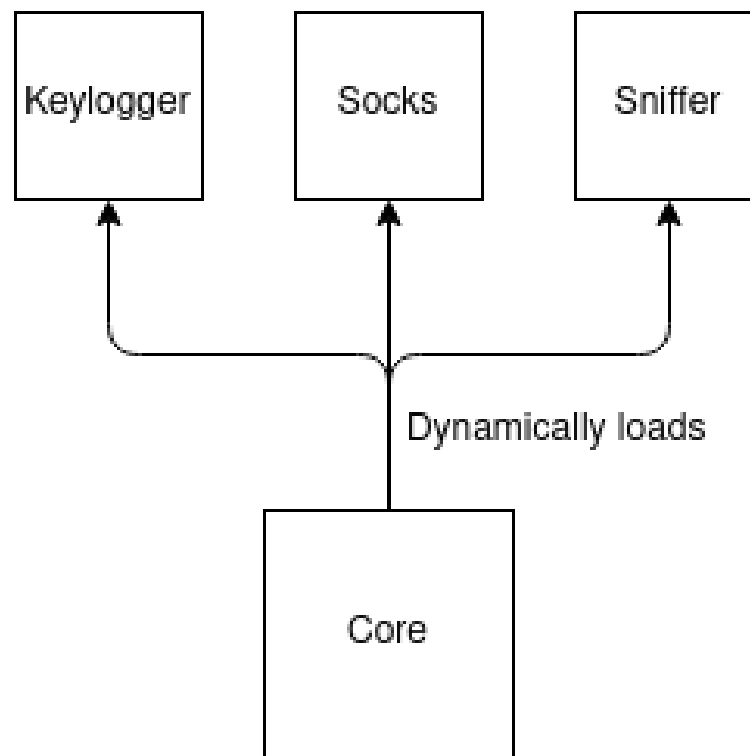
# Fileless Malware and Memory Artifacts



# Motivation for Fileless Malware

- Development of malware is a complex software project
- If analysts ever find your malware, all hard work is burned
- Therefore, sophisticated malware has a 'Plug and Play' concept for most of its capabilities
  - Malware consists of a main module. Capabilities are then loaded on the fly
  - If one module is flagged it is easy to replace / only one module needs to be adapted

# Motivation for Fileless Malware



# Plug and Play

- Back in the days, an extension was dropped as a DLL and loaded from disk
  - Problem 1: AV heuristics / automatic analysis of dropped files
  - Problem 2: Files left on the filesystem will be forgotten by operators
- Malware authors started looking for ways to load extensions without ever touching disk

# Fileless Malware

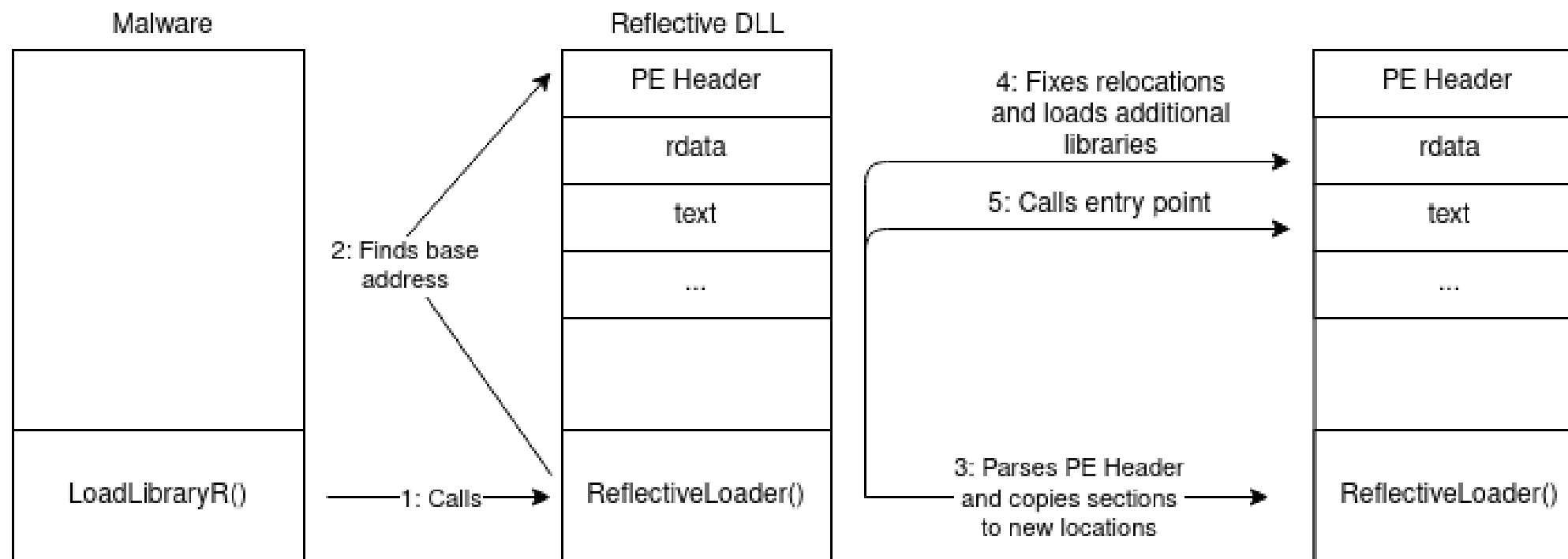
- Monitoring of memory operations is prone to false positives and also resource intense.
  - DRM uses similar concepts for legitimate use
- Many concepts for fileless malware: {Reflective DLL, sRDI/Donut, in memory .NET}
  - All of them have their drawbacks which enable analysts and security products to find them



# Reflective DLL

- Self loading DLL
  - First implemented by Stephen Fewer in 2011
- Regular DLL with one special export: ReflectiveLoader
  - Implements a PE loader: takes care of fixing IAT, Relocations and so on
  - Loader itself is fully position independent (PIC)
- Malware needs to be able to find the ReflectiveLoader in memory
  - Requires additional code

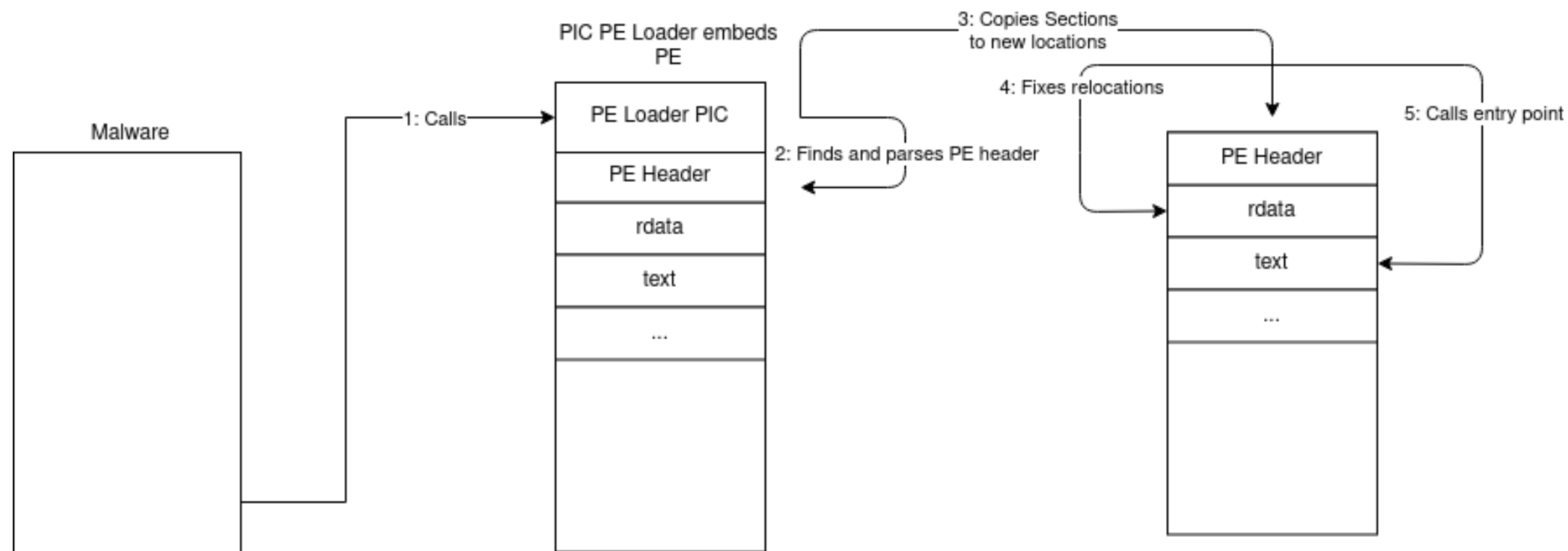
# Recap Reflective DLL



# PE2Shellcode

- Implementation of a PE Loader as pure PIC
  - PIC embeds a given a PE file which it loads and executes
  - No PE loader or LoadLibraryR has to be compiled into malware
- PE are not actually converted they are just wrapped with a PIC PE loader
- PE loader finds the embedded PE file and loads it in memory
- PE – To – Shellcode - Concept
- Example open source implementations:
  - [https://github.com/hasherezade/pe\\_to\\_shellcode](https://github.com/hasherezade/pe_to_shellcode)
  - <https://github.com/TheWover/donut>
  - <https://github.com/monoxgas/sRDI>

# PIC PE Loader

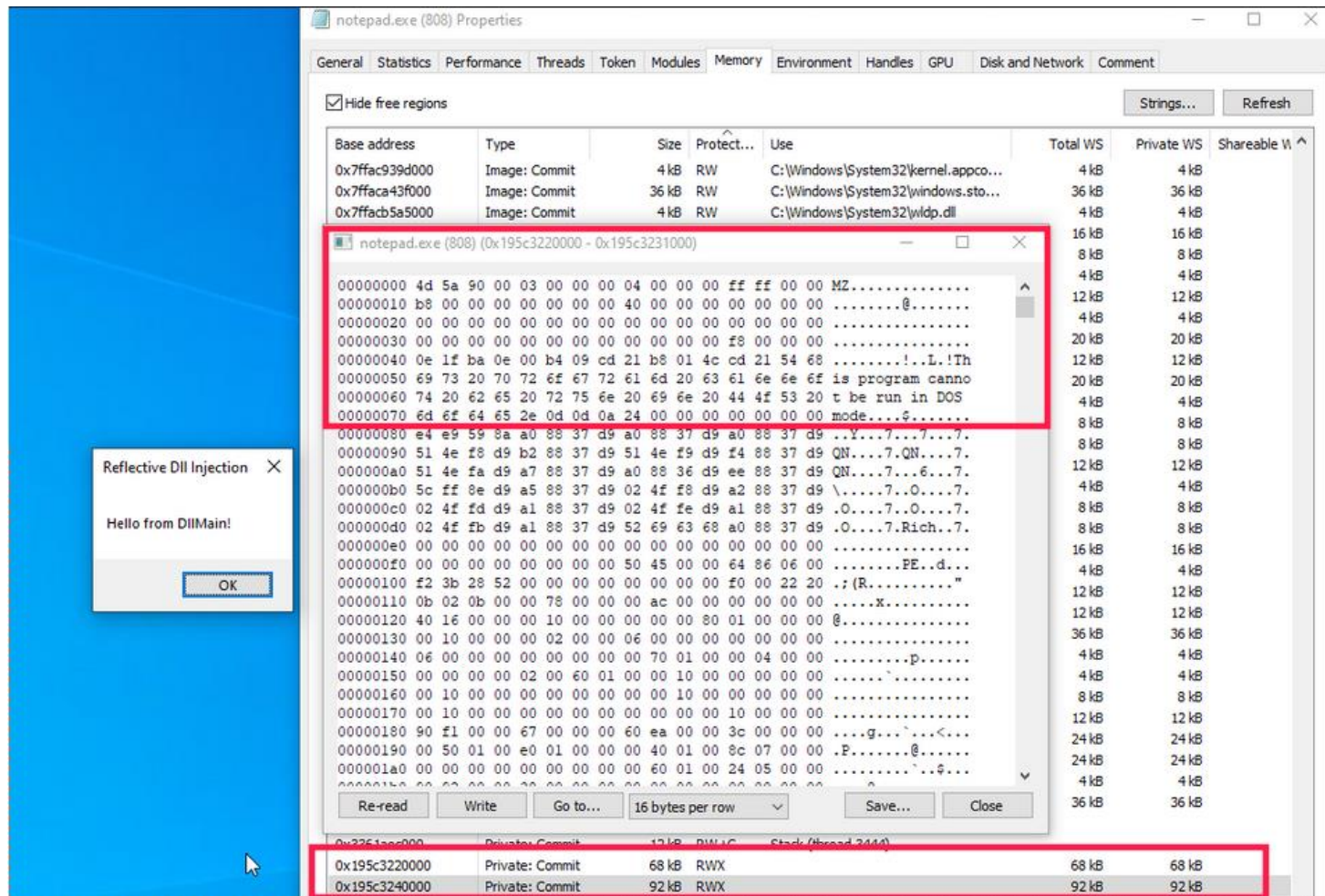


# Suspicious Memory Artifacts

- PE loaders need to allocate more memory to load the PE file
  - Some sections need to be executable (.text segment)
- Suspicious Memory Pages such as R(W)X
- Private Committed and Executable Memory
  - VirtualAlloc
  - Not backed by a file on disk
  - R(W)X pages should only exist as part of a PE file
- PE Header in private committed memory

# Artifacts Reflective DLL

- PE Header in private committed memory
- Executable and private committed pages



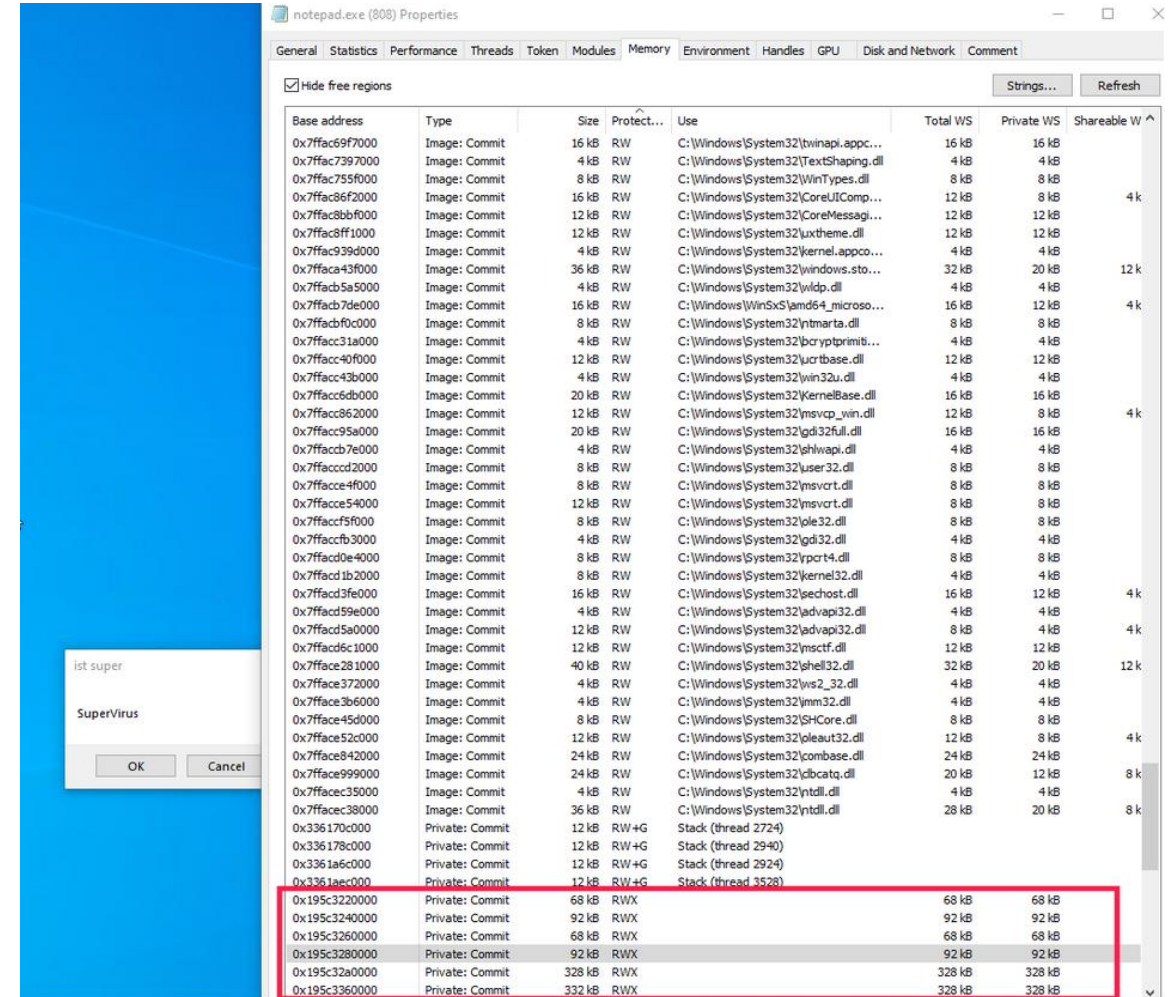
The screenshot illustrates the state of a Windows system during a reflective DLL injection. A dialog box titled "Reflective DLL Injection" is displayed, showing the message "Hello from DllMain!" and an "OK" button. In the background, the "notepad.exe (808) Properties" window is open, with the "Memory" tab selected. The memory dump shows the following data:

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable W
0x7fac939d000	Image: Commit	4 kB	RW	C:\Windows\System32\kernel.appco...	4 kB	4 kB	
0x7faca43f000	Image: Commit	36 kB	RW	C:\Windows\System32\windows.sto...	36 kB	36 kB	
0x7facb5a5000	Image: Commit	4 kB	RW	C:\Windows\System32\wldp.dll	4 kB	4 kB	
<b>notepad.exe (808) (0x195c3220000 - 0x195c3231000)</b>							
00000000	4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00	MZ.....					
00000010	b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	.....@.....					
00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....					
00000030	00 00 00 00 00 00 00 00 00 00 00 00 f8 00 00 00	.....					
00000040	0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68	.....!..L!Th					
00000050	69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f	is program canno					
00000060	74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20	t be run in DOS					
00000070	6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00	mode....\$. ....					
00000080	e4 e9 59 8a a0 88 37 d9 a0 88 37 d9 a0 88 37 d9	..Y...7...7...7...					
00000090	51 4e f8 d9 b2 88 37 d9 51 4e f9 d9 f4 88 37 d9	QN....7.QN....7...					
000000a0	51 4e fa d9 a7 88 37 d9 a0 88 36 d9 ee 88 37 d9	QN....7...6...7...					
000000b0	5c ff 8e d9 a5 88 37 d9 02 4f f8 d9 a2 88 37 d9	\....7..O....7...					
000000c0	02 4f fd d9 a1 88 37 d9 02 4f fe d9 a1 88 37 d9	.O....7..O....7...					
000000d0	02 4f fb d9 a1 88 37 d9 52 69 63 68 a0 88 37 d9	.O....7.Rich..7...					
000000e0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....PE..d...					
000000f0	00 00 00 00 00 00 00 00 50 45 00 00 64 86 06 00	.....PE..d...					
00000100	f2 3b 28 52 00 00 00 00 00 00 00 00 f0 00 22 20	;(R....."					
00000110	0b 02 0b 00 00 78 00 00 ac 00 00 00 00 00 00 00	.....x.....					
00000120	40 16 00 00 00 10 00 00 00 00 80 01 00 00 00 00	@.....					
00000130	00 10 00 00 00 02 00 00 06 00 00 00 00 00 00 00	.....p.....					
00000140	06 00 00 00 00 00 00 00 00 70 01 00 00 04 00 00	.....p.....					
00000150	00 00 00 00 02 00 60 01 00 00 10 00 00 00 00 00	.....					
00000160	00 10 00 00 00 00 00 00 00 00 10 00 00 00 00 00	.....					
00000170	00 10 00 00 00 00 00 00 00 00 00 00 10 00 00 00	.....					
00000180	90 f1 00 00 67 00 00 00 60 ea 00 00 3c 00 00 00	...g...`...<...					
00000190	00 50 01 00 e0 01 00 00 40 01 00 8c 07 00 00 00	.P.....@.....					
000001a0	00 00 00 00 00 00 60 01 00 24 05 00 00 00 00	...\$. ....					
000001b0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....					
0x2261ac000	Private: Commit	12 kB	RW	Stack (Thread 3440)			
0x195c3220000	Private: Commit	68 kB	RWX		68 kB	68 kB	
0x195c3240000	Private: Commit	92 kB	RWX		92 kB	92 kB	

# Memory Artifacts Donut

- Executable and private committed memory
- Donut can wipe the PE header from memory
  - Header is in memory during loading
  - Rest of PE structure is still in memory. PESieve can find it [1]

[1] <https://github.com/hasherezade/pe-sieve>

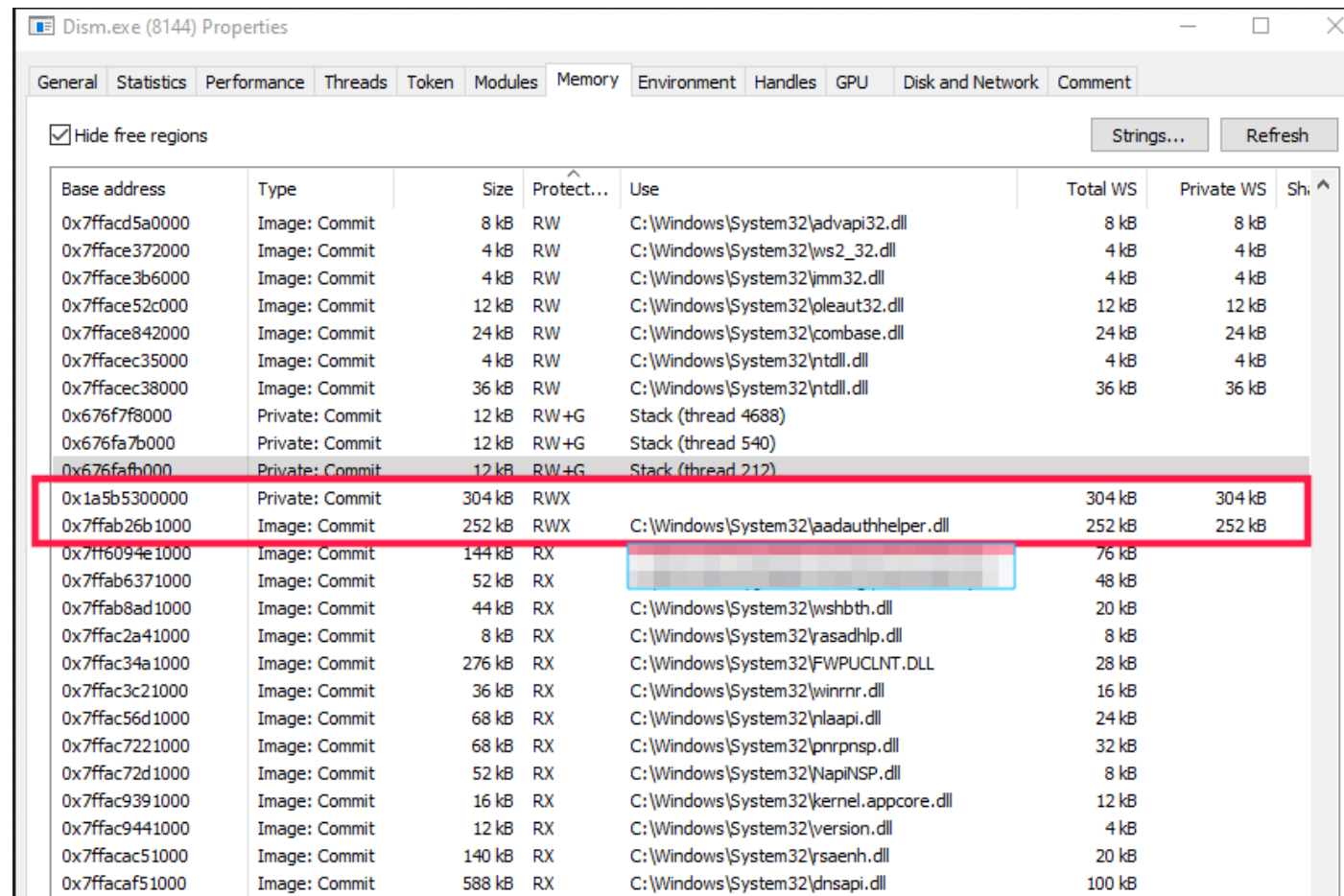


Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable W
0x7ffac69f7000	Image: Commit	16 kB	RW	C:\Windows\System32\winapi.app...	16 kB	16 kB	
0x7ffac7397000	Image: Commit	4 kB	RW	C:\Windows\System32\TextShaping.dll	4 kB	4 kB	
0x7ffac755f000	Image: Commit	8 kB	RW	C:\Windows\System32\WinTypes.dll	8 kB	8 kB	
0x7ffac86f2000	Image: Commit	16 kB	RW	C:\Windows\System32\CoreUIComp...	12 kB	8 kB	4 k
0x7ffac8bbf000	Image: Commit	12 kB	RW	C:\Windows\System32\CoreMessagi...	12 kB	12 kB	
0x7ffac8ff1000	Image: Commit	12 kB	RW	C:\Windows\System32\uxtheme.dll	12 kB	12 kB	
0x7ffac939d000	Image: Commit	4 kB	RW	C:\Windows\System32\kernel.appco...	4 kB	4 kB	
0x7ffaca43f000	Image: Commit	36 kB	RW	C:\Windows\System32\windows.sto...	32 kB	20 kB	12 k
0x7ffacb5a5000	Image: Commit	4 kB	RW	C:\Windows\System32\wldp.dll	4 kB	4 kB	
0x7ffacb7de000	Image: Commit	16 kB	RW	C:\Windows\WinSxS\amd64_microso...	16 kB	12 kB	4 k
0x7ffacf0c000	Image: Commit	8 kB	RW	C:\Windows\System32\ntmarta.dll	8 kB	8 kB	
0x7ffacc31a000	Image: Commit	4 kB	RW	C:\Windows\System32\bcryptprimit...	4 kB	4 kB	
0x7ffacc40f000	Image: Commit	12 kB	RW	C:\Windows\System32\ucrtbase.dll	12 kB	12 kB	
0x7ffacc43b000	Image: Commit	4 kB	RW	C:\Windows\System32\win32u.dll	4 kB	4 kB	
0x7ffacc6db000	Image: Commit	20 kB	RW	C:\Windows\System32\KernelBase.dll	16 kB	16 kB	
0x7ffacc862000	Image: Commit	12 kB	RW	C:\Windows\System32\msvcp_win.dll	12 kB	8 kB	4 k
0x7ffacc95a000	Image: Commit	20 kB	RW	C:\Windows\System32\gdi32full.dll	16 kB	16 kB	
0x7ffaccb7e000	Image: Commit	4 kB	RW	C:\Windows\System32\shlwapi.dll	4 kB	4 kB	
0x7ffaccdd2000	Image: Commit	8 kB	RW	C:\Windows\System32\user32.dll	8 kB	8 kB	
0x7ffacce4f000	Image: Commit	8 kB	RW	C:\Windows\System32\msvrt.dll	8 kB	8 kB	
0x7ffacce54000	Image: Commit	12 kB	RW	C:\Windows\System32\msvrt.dll	8 kB	8 kB	
0x7ffaccf5f000	Image: Commit	8 kB	RW	C:\Windows\System32\ole32.dll	8 kB	8 kB	
0x7ffaccfb3000	Image: Commit	4 kB	RW	C:\Windows\System32\gdi32.dll	4 kB	4 kB	
0x7ffacd0e4000	Image: Commit	8 kB	RW	C:\Windows\System32\pct14.dll	8 kB	8 kB	
0x7ffacd1b2000	Image: Commit	8 kB	RW	C:\Windows\System32\kernel32.dll	4 kB	4 kB	
0x7ffacd3fe000	Image: Commit	16 kB	RW	C:\Windows\System32\sechost.dll	16 kB	12 kB	4 k
0x7ffacd59e000	Image: Commit	4 kB	RW	C:\Windows\System32\advapi32.dll	4 kB	4 kB	
0x7ffacd5a0000	Image: Commit	12 kB	RW	C:\Windows\System32\advapi32.dll	8 kB	4 kB	4 k
0x7ffacd6c1000	Image: Commit	12 kB	RW	C:\Windows\System32\msctf.dll	12 kB	12 kB	
0x7ffacd81000	Image: Commit	40 kB	RW	C:\Windows\System32\shell32.dll	32 kB	20 kB	12 k
0x7ffacd972000	Image: Commit	4 kB	RW	C:\Windows\System32\ws2_32.dll	4 kB	4 kB	
0x7ffacd9b6000	Image: Commit	4 kB	RW	C:\Windows\System32\mm32.dll	4 kB	4 kB	
0x7ffacd9d000	Image: Commit	8 kB	RW	C:\Windows\System32\SHCore.dll	8 kB	8 kB	
0x7ffacd9e000	Image: Commit	12 kB	RW	C:\Windows\System32\oleaut32.dll	12 kB	8 kB	4 k
0x7ffacd9f2000	Image: Commit	24 kB	RW	C:\Windows\System32\combase.dll	24 kB	24 kB	
0x7ffacd999000	Image: Commit	24 kB	RW	C:\Windows\System32\cbcatq.dll	20 kB	12 kB	8 k
0x7ffacd935000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll	4 kB	4 kB	
0x7ffacd938000	Image: Commit	36 kB	RW	C:\Windows\System32\ntdll.dll	28 kB	20 kB	8 k
0x336170c000	Private: Commit	12 kB	RW+G	Stack (thread 2724)			
0x336178c000	Private: Commit	12 kB	RW+G	Stack (thread 2940)			
0x3361a6c000	Private: Commit	12 kB	RW+G	Stack (thread 2924)			
0x3361a6c000	Private: Commit	12 kB	RW+G	Stack (thread 3528)			
0x195c3220000	Private: Commit	68 kB	RWX		68 kB	68 kB	
0x195c3240000	Private: Commit	92 kB	RWX		92 kB	92 kB	
0x195c3260000	Private: Commit	68 kB	RWX		68 kB	68 kB	
0x195c3280000	Private: Commit	92 kB	RWX		92 kB	92 kB	
0x195c32a0000	Private: Commit	328 kB	RWX		328 kB	328 kB	
0x195c3360000	Private: Commit	332 kB	RWX		328 kB	328 kB	



# Artifacts with File Backed Memory

- Try to first copy the PIC to file backed memory
- Donut / RDLL does not stay where it was first copied to
- Breaks concepts, like Phantom DLL Hollowing [1] (Forrest orr)



The screenshot shows the 'Memory' tab of the 'Dism.exe (8144) Properties' window. The 'Hide free regions' checkbox is checked. A table lists memory regions with columns: Base address, Type, Size, Protect..., Use, Total WS, Private WS, and Sh. A red box highlights the entry for 'C:\Windows\System32\aaauthhelper.dll' at base address 0x7ffab26b1000, which has a size of 252 kB and RWX permissions. Another red box highlights the entry for 'Stack (thread 212)' at base address 0x676fafb000, which has a size of 12 kB and RW+G permissions.

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Sh
0x7ffacd5a0000	Image: Commit	8 kB	RW	C:\Windows\System32\advapi32.dll	8 kB	8 kB	
0x7fface372000	Image: Commit	4 kB	RW	C:\Windows\System32\ws2_32.dll	4 kB	4 kB	
0x7fface3b6000	Image: Commit	4 kB	RW	C:\Windows\System32\imm32.dll	4 kB	4 kB	
0x7fface52c000	Image: Commit	12 kB	RW	C:\Windows\System32\oleaut32.dll	12 kB	12 kB	
0x7fface842000	Image: Commit	24 kB	RW	C:\Windows\System32\combase.dll	24 kB	24 kB	
0x7ffacec35000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll	4 kB	4 kB	
0x7ffacec38000	Image: Commit	36 kB	RW	C:\Windows\System32\ntdll.dll	36 kB	36 kB	
0x676f7f8000	Private: Commit	12 kB	RW+G	Stack (thread 4688)			
0x676fa7b000	Private: Commit	12 kB	RW+G	Stack (thread 540)			
0x676fafb000	Private: Commit	12 kB	RW+G	Stack (thread 212)			
0x1a5b5300000	Private: Commit	304 kB	RWX		304 kB	304 kB	
0x7ffab26b1000	Image: Commit	252 kB	RWX	C:\Windows\System32\aaauthhelper.dll	252 kB	252 kB	
0x7ff6094e1000	Image: Commit	144 kB	RX		76 kB		
0x7ffab6371000	Image: Commit	52 kB	RX		48 kB		
0x7ffab8ad1000	Image: Commit	44 kB	RX	C:\Windows\System32\wshbth.dll	20 kB		
0x7ffac2a41000	Image: Commit	8 kB	RX	C:\Windows\System32\yasadhlp.dll	8 kB		
0x7ffac34a1000	Image: Commit	276 kB	RX	C:\Windows\System32\FWPUCLNT.DLL	28 kB		
0x7ffac3c21000	Image: Commit	36 kB	RX	C:\Windows\System32\winnr.dll	16 kB		
0x7ffac56d1000	Image: Commit	68 kB	RX	C:\Windows\System32\laapi.dll	24 kB		
0x7ffac7221000	Image: Commit	68 kB	RX	C:\Windows\System32\pnprnsp.dll	32 kB		
0x7ffac72d1000	Image: Commit	52 kB	RX	C:\Windows\System32\NapiNSP.dll	8 kB		
0x7ffac9391000	Image: Commit	16 kB	RX	C:\Windows\System32\kernel.appcore.dll	12 kB		
0x7ffac9441000	Image: Commit	12 kB	RX	C:\Windows\System32\version.dll	4 kB		
0x7ffacac51000	Image: Commit	140 kB	RX	C:\Windows\System32\rsaenh.dll	20 kB		
0x7ffacaf51000	Image: Commit	588 kB	RX	C:\Windows\System32\dnsapi.dll	100 kB		

[1] <https://www.forrest-orr.net/post/malicious-memory-artifacts-part-i-dll-hollowing>



# Weakness of RDLL / Donut

- In memory PE loaders leave memory artifacts
- Some artifacts can be removed after loading
  - During loading, automated products have a timeframe to work with
- In memory execution of .NET would mean constantly fighting AMSI
- If the whole tool was PIC, there would never be a PE to load ...

# PIC Your Malware!

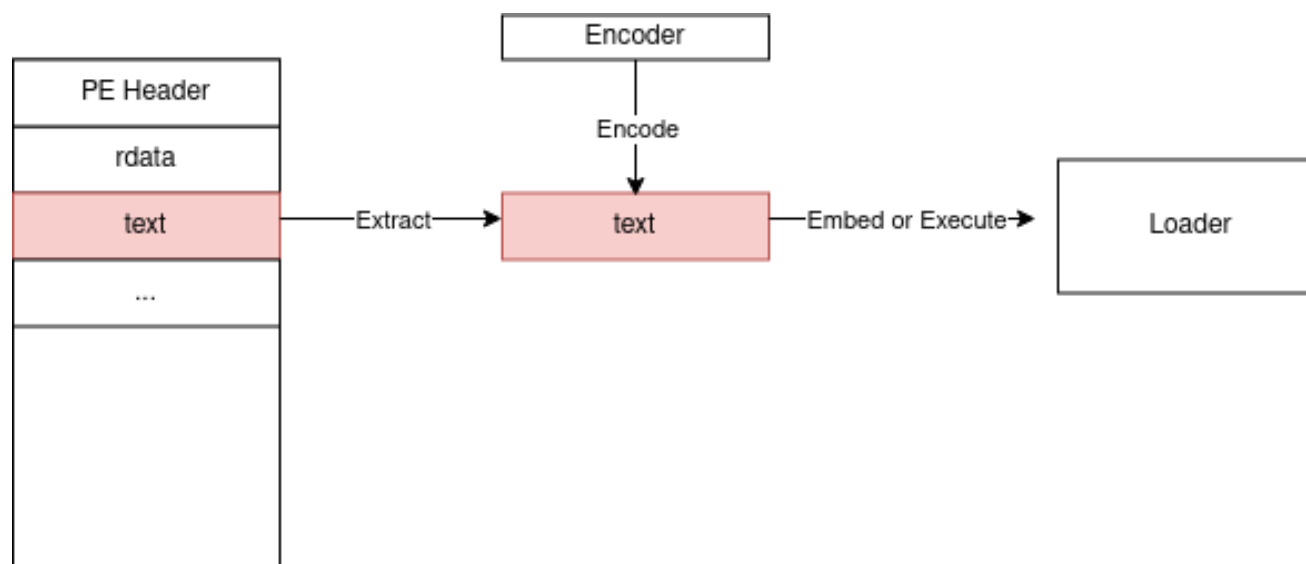


# PIC Your Malware!

- C code can be compiled to PE files living fully in its .text segment [1]
- Extracting the .text segment means obtaining PIC
  - No need to load it, just place it in memory and jump to it
- Conditions
  - No relocations
  - No other segments in use than .text
  - Uses string stacking and avoids global / static variables
  - Must be able to parse DLLs for function pointers

➤ [1] <https://vxug.fakedoma.in/papers/VXUG/Exclusive/FromaCprojectthroughassemblytoshellcodeHasherezade.pdf>

# Position Independent Code

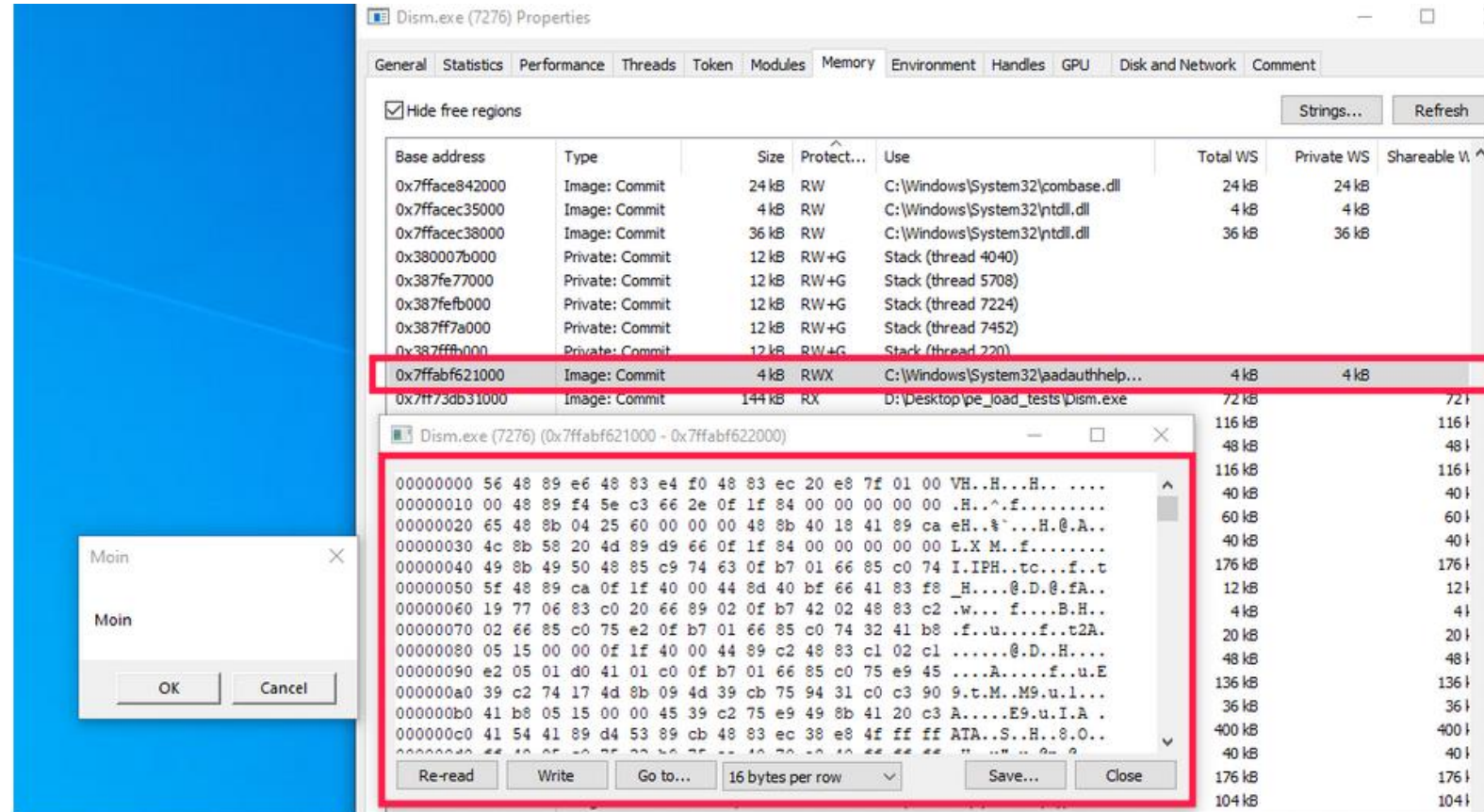


# Position Independent Code

- The .text segment of such a program is fully position independent
- Stays where it is copied to
- Can be executed like classic shellcode
  - Needs a host process
- Bonus: can be encoded with any shellcode encoder to break signatures!
  - As long as no parameters are passed and you expect the shellcode to return properly

# PIC in File Backed Memory

- PIC stays where it is copied to
- No PE loading necessary
- No new memory allocated
- No additional private committed pages
- No PE header



The screenshot shows the 'Dism.exe (7276) Properties' window with the 'Memory' tab selected. The 'Hide free regions' checkbox is checked. The table below lists the memory regions:

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable V.
0x7fface842000	Image: Commit	24 kB	RW	C:\Windows\System32\combase.dll	24 kB	24 kB	
0x7ffacec35000	Image: Commit	4 kB	RW	C:\Windows\System32\ntdll.dll	4 kB	4 kB	
0x7ffacec38000	Image: Commit	36 kB	RW	C:\Windows\System32\ntdll.dll	36 kB	36 kB	
0x380007b000	Private: Commit	12 kB	RW+G	Stack (thread 4040)			
0x387fe77000	Private: Commit	12 kB	RW+G	Stack (thread 5708)			
0x387feb000	Private: Commit	12 kB	RW+G	Stack (thread 7224)			
0x387ff7a000	Private: Commit	12 kB	RW+G	Stack (thread 7452)			
0x387fffb000	Private: Commit	12 kB	RW+G	Stack (thread 7220)			
0x7ffabf621000	Image: Commit	4 kB	RWX	C:\Windows\System32\aaauthhelp...	4 kB	4 kB	
0x7ff73db31000	Image: Commit	144 kB	RX	D:\Desktop\pe_load_tests\Dism.exe	72 kB	72 kB	

Below the table, a hex dump window titled 'Dism.exe (7276) (0x7ffabf621000 - 0x7ffabf622000)' is shown, displaying the raw memory contents of the loaded image. The dump shows hexadecimal values and their corresponding ASCII representations, including the start of a PE header.

# Nothing is Fully Undetectable

- Also not in memory
- PIC helps reducing memory artifact fingerprint
- Abnormal allocation of file backed memory itself can still be fingerprinted [1]
  - Prone to false positives
- Analysts will still catch malicious behavior of processes
  
- Let us try to avoid suspicious Sysmon events using PIC!

[1] <https://www.forrest-orr.net/post/masking-malicious-memory-artifacts-part-ii-insights-from-moneta>

# PIC Lsass Dumper





# Lsass Dumper as PIC

- If a process opens Lsass with `PROCESS_ALL_ACCESS` or `PROCESS_VM_READ | PROCESS_QUERY_INFORMATION` it is most likely going to dump Lsass
- ProcessAccess event every time a process uses `OpenProcess()`
- Defenders definitely monitor this event related to Lsass
- What if we never open Lsass?

# Avoiding ProcessAccess

- Some benign processes already have a handle to Lsass (HandleHolder)
- PROCESS\_QUERY\_INFORMATION | PROCESS\_DUP\_HANDLE
- Clone the existing handle using NtDuplicateObject
- Handle can then be used in a malicious context
- Sysmon only throws ProcessAccess on HandleHolder
- Lsass usually has a handle to itself
  - Open Lsass with access mask not revealing your true intention

# Avoiding ProcessAccess

- MiniDumpWriteDump from DbgHelp internally opens multiple new handles
- *@Rookuu\_\_* [1] demonstrated the usage of ReactOS MiniDumpWriteDump to dump
- However, also this function opens some new handles.
  - Replace EnumerateLoadedModulesW64()
- Using ReactOS MiniDumpWriteDump + ReactOS EnumerateLoadedModulesW64() to dump Lsass using a cloned handle does not appear in Sysmon

[1] <https://github.com/rookuu/BOFs/tree/main/MiniDumpWriteDump>

# Introducing HandleKatz

- HandleKatz enumerates processes for a suitable handle to dump Lsass
- Clones handle and uses ReactOS Code (MiniDumpWriteDump + EnumerateLoadedModules) to dump the process without opening any new handle to Lsass
- Then writes an obfuscated dump to disk
- Uses direct syscalls
- Fully PIC

```
      /\_/\
      (o.o)
      > ^ <
      Gib Handle!

[*] HandleKatz v1.0
[*] Attempting to clone lsass handle from pid: 7688
[*] Outfile: C:\temp\gedumt.dmp
[+] Found and successfully cloned handle to lsass in: HandleHolder.exe (7688)
    [+] Handle Rights: PROCESS_QUERY_INFORMATION | PROCESS_VM_READ
[*] Now trying to dump lsass ...
[+] Lsass dump is complete
[*] Tschau
```

# HandleKatz

Event 10, Sysmon

General Details

Process accessed:  
RuleName: -  
UtcTime: 2021-04-14 07:47:50.584  
SourceProcessGUID: {35223b03-9e26-6076-f90c-000000001b00}  
SourceProcessId: 512  
SourceThreadId: 5384  
SourceImage: [REDACTED] HandleKatz.exe  
TargetProcessGUID: {35223b03-9d45-6076-e70c-000000001b00}  
TargetProcessId: 7688  
TargetImage: [REDACTED] HandleHolder.exe  
GrantedAccess: 0x1440  
CallTrace: C:\Windows\SYSTEM32\ntdll.dll+9cae4| [REDACTED] \HandleKatz\x64\Release\HandleKatz.exe+26a2| [REDACTED] \HandleKatz\x64\Release\HandleKatz.exe+23bc|D:\ [REDACTED] \HandleKatz\x64\Release\HandleKatz.exe+2c64|C:\Windows\System32\KERNEL32.DLL+17034|C:\Windows\SYSTEM32\ntdll.dll+4d241

# Power of PIC

- Feel free to upload on VT or drop it to disk
- HandleKatz can be encoded with SGN [1]
- Different encoded versions of PIC perform the exact same complex task
  - Yet, they look completely different
- Depending on the encoder, encoded version cannot take arguments or do not return properly

```
ssdeep,1.1--blocksize:hash:hash,filename
384:SojqwXNx20brGlgCoQ5DB6+UspgGysl4dF:SyqYNNGLgg96wpgGysKF," /HandleKatz.bin.sgn"
384:pAa6V5kHg4lSTUxULthjS5i+MPqbY+Ux2oRzpZ8:g5kAnTtEvbHiRZ8," /HandleKatz.bin.sgn_1"
384:Q7llcoQv/zR0uidcrsj84Sw9jjNDFxtQeiP02uwe4eAc:Q7llcoQTmuid9/fjNDFLbiLu0u," /HandleKatz.bin.sgn_2"
384:BVa8V5nsX7vK1pfvip88iYaAo7ApK3o1Np+OM:BM8VcjKLu88taAiAEAp+OM," /HandleKatz.bin.sgn_3"
```

# How to integrate HandleKatz

- Comes with a header file for HandleKatz's entry point
- Simply cast pointer to the typedef of HandleKatz
- Easy to integrate into your favorite C2

```
printf("[*] Now executing PIC ... \n");  
dw_success = ((HandleKatz*)buf_shellcode)(b_only_recon, path_dump, dw_pid, buf_output);  
printf("[*] HandleKatz returned: %d\n", dw_success);  
printf("===== output =====\n");  
printf("%s\n", buf_output);
```

# HandleKatz

- Code + Compiled PIC can be found at <https://github.com/codewhitesec/HandleKatz>
- Other PIC examples: <https://github.com/thefLink/C-To-Shellcode-Examples>
- How to build and protect PIC?



# Automating Creation and Protection



# Source Files

- 2010 - @nickharbour - [Writing Shellcode with a C Compiler](#)
- 2013 - @mattifestation - [Writing Optimized Windows Shellcode in C](#)
- 2020 - @hasherezade - [From a C project, through assembly, to shellcode](#)
- 2021 - @passthehashbrwn - [Dynamic payload compilation with mingw](#)

# PE vs. PIC

Characteristic	PE	PIC
<b>Structure</b>	Structured in sections with different memory characteristics, separation of functions and data	All in one binary blob
<b>Relocation</b>	Yes, will be calculated through the loading process of the OS	No relocations, everything must be called relative to IP
<b>Imports</b>	External function calls are resolved through the loading process of the OS	External functions can only be called after they are manually resolved

# Position Independent Problems

- **Imports** – External functions can only be called after they are manually resolved
- **Strings** – Have to be  
`{'s', 't', 'a', 'c', 'k', 's', 't', 'r', 'i', 'n', 'g', 's', 0}; [1]`  
Hashes are often used to hide imports but are not applicable for other use cases
- **Global Data** – Must be passed from one function to another  
→ Especially a problem for our imports

[1] <https://www.fireeye.com/blog/threat-research/2016/06/automatically-extracting-obfuscated-strings.html>

# Position Independent Problems

From MSDN to source code

C++

```
void Sleep(  
    DWORD dwMilliseconds  
);
```

+ Library Name

# Position Independent Problems

From MSDN to source code

C++

```
void Sleep(  
    DWORD dwMilliseconds  
);
```

+ Library Name

python3 GenFunctionPointer.py <https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-sleep>

→ `typedef void(__stdcall *p_Sleep)(DWORD dwMilliseconds);`

# Position Independent Problems

Script generated code pattern

```
CHAR SleepStr[] = { 0 }; //str: Sleep
```

```
DeobfuscateString(SleepStr, SleepStr);
```

```
GetProcAddressManMap(Kernel32, SleepStr, &ProcAddress, Api);
```

```
Api->_Sleep = p_Sleep(ProcAddress);
```

# Position Independent Problems

Script generated code pattern

```
CHAR SleepStr[] = { 0 }; //str: Sleep
```

```
DeobfuscateString(SleepStr, SleepStr);
```

```
GetProcAddressManMap(Kernel32, SleepStr, &ProcAddress, Api);
```

```
Api->_Sleep = p_Sleep(ProcAddress);
```



# Position Independent Problems

Script generated code pattern

```
CHAR SleepStr[] = { 0 }; //str: Sleep
```

```
DeobfuscateString(SleepStr, SleepStr);
```

```
GetProcAddressManMap(Kernel32, SleepStr, &ProcAddress, Api);
```

```
Api->_Sleep = p_Sleep(ProcAddress);
```

# Position Independent Problems

Script generated code pattern

```
CHAR SleepStr[] = { 0 }; //str: Sleep
```

```
DeobfuscateString(SleepStr, SleepStr);
```

```
GetProcAddressManMap(Kernel32, SleepStr, &ProcAddress, Api);
```

```
Api->_Sleep = p_Sleep(ProcAddress);
```

# Position Independent Problems

## Script generated code pattern

```
CHAR SleepStr[] = { 0 }; //str: Sleep
```

```
DeobfuscateString(SleepStr, SleepStr);
```

```
GetProcAddressManMap(Kernel32, SleepStr, &ProcAddress, Api);
```

```
Api->_Sleep = p_Sleep(ProcAddress);
```

- + Scripted the importing process of external functions
- + Write strings as they are
- + Make function pointers available everywhere → Api struct

# Position Independent Problems

## Script generated code pattern

Pre-build script *obfuscate.py* for string processing:

- 1) Find "INT ObfuscationKey" in source code and set random key value
- 2) Parse all files for string patterns, encode them with the obfuscation key and paste encoded version

```
CHAR SleepStr[] = { 0 }; //str: Sleep
```

```
→ CHAR SleepStr[] = {18, 93, 21, 83, 66, -1}; //str: Sleep
```

# Template Based Shellcode

## Gimme Shelter

AVs have different triggers for in-memory scans, for example jumping to the start of an executable region after...

- allocating it (and filling it with data) (RWX)
- changing its characteristics (RW → RX)

# Template Based Shellcode

## Gimme Shelter

AVs have different triggers for in-memory scans, for example jumping to the start of an executable region after...

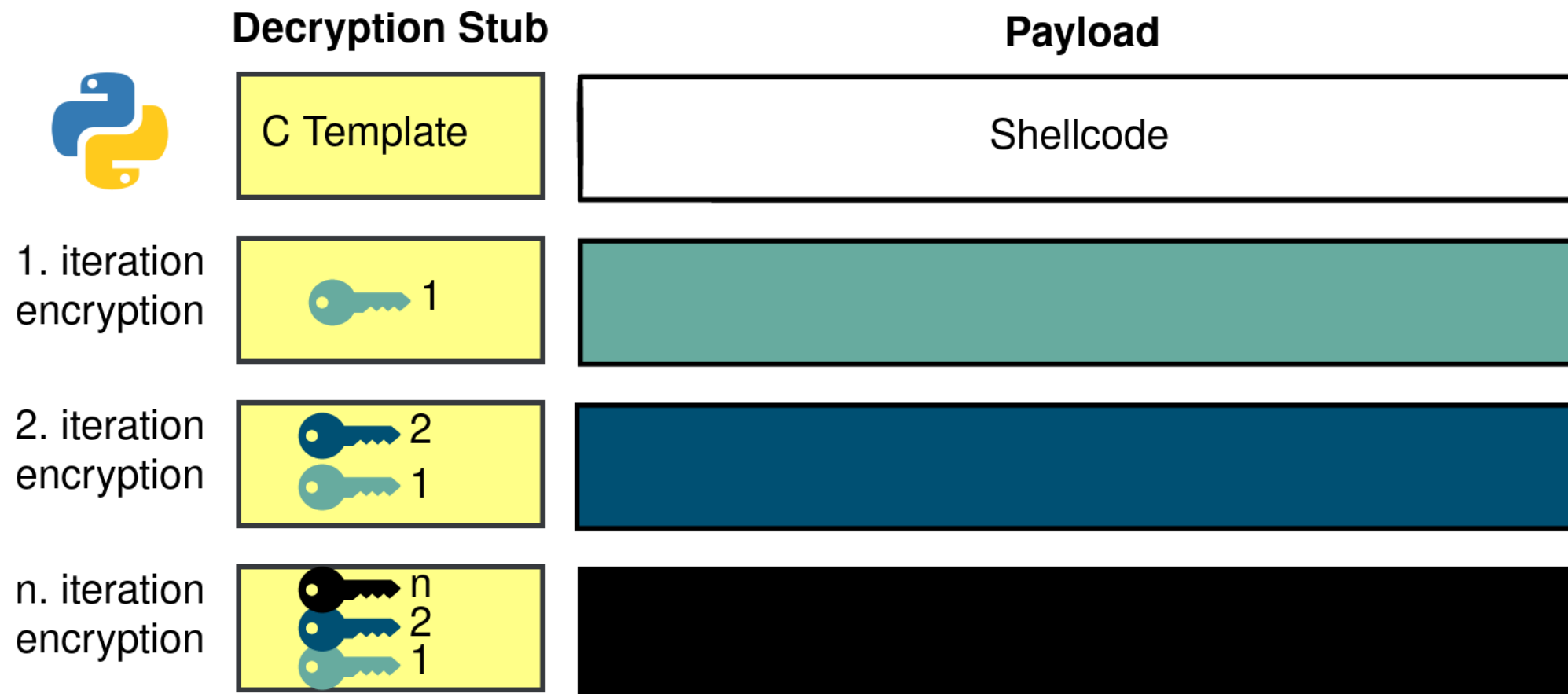
- allocating it (and filling it with data) (RWX)
- changing its characteristics (RW → RX)

Self-decrypting shellcode as a wrapper to protect payloads

- 1) Place self-decrypting shellcode in RWX memory region
- 2) Execute it → hides payload (long enough) from memory scanner

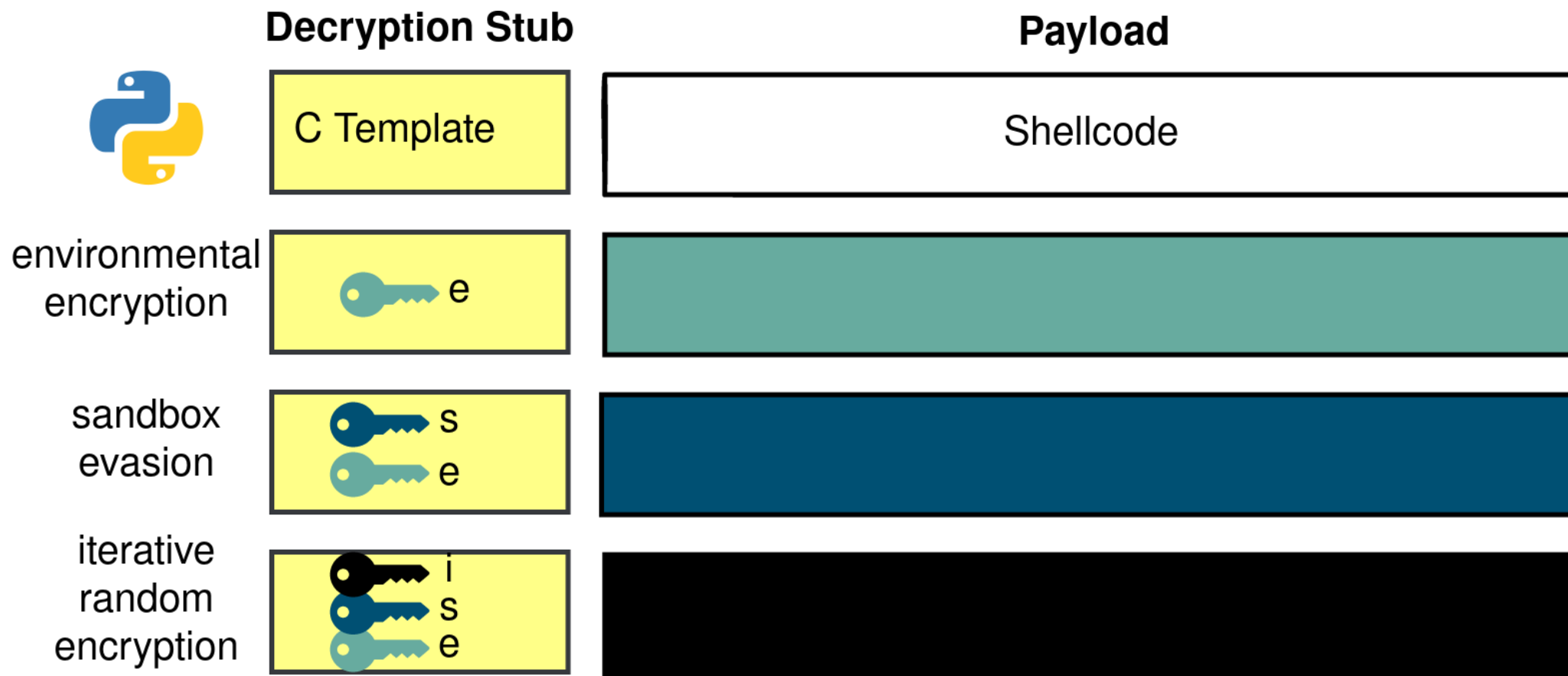
# Template Based Shellcode

Self-decrypting shellcode pre-build script



# Template Based Shellcode

Self-decrypting shellcode pre-build script





# Template Based Shellcode

## Self-decrypting shellcode Template

```
#define LOG 1
#define UK 0
#define DK 0
#define HK 0
#define PK 0

#if UK or DK or HK or PK or LOG [Active Preprocessor Block]
#endif

#if UK or DK or HK or PK [Inactive Preprocessor Block]
#endif

#if LOG [Active Preprocessor Block]
#endif
```

# Template Based Shellcode

## Self-decrypting shellcode Template

```
#define LOG 0
#define UK 1
#define DK 0
#define HK 0
#define PK 0

#if UK or DK or HK or PK or LOG [Active Preprocessor Block]
#endif

#if UK or DK or HK or PK [Active Preprocessor Block]
#endif

#if LOG [Inactive Preprocessor Block]
#endif
```

# Template Based Shellcode

## Self-decrypting shellcode Template

```
#define Parameter 1
#define _WIN64 1

#if Parameter
extern"C" VOID prologue(CHAR *Args);
extern"C" VOID mainAct(CHAR *Args);
#if _WIN64
extern"C" VOID AlignRSP(CHAR *Args);
#endif
#else
extern"C" VOID prologue();
extern"C" VOID mainAct();
#if _WIN64
extern"C" VOID AlignRSP();
#endif
#endif
```

# Template Based Shellcode

## Self-decrypting shellcode Template

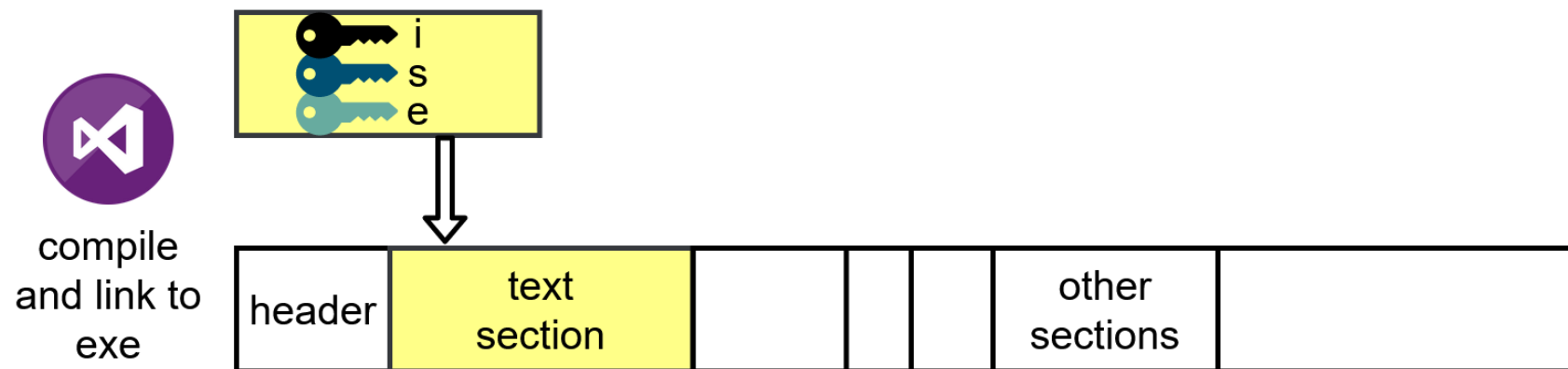
```
#define Parameter 0
#define _WIN64 0

#if Parameter
extern"C" VOID prologue(CHAR *Args);
extern"C" VOID mainAct(CHAR *Args);
#if _WIN64
extern"C" VOID AlignRSP(CHAR *Args);
#endif
#else
extern"C" VOID prologue();
extern"C" VOID mainAct();
#if _WIN64
extern"C" VOID AlignRSP();
#endif
#endif
```

# Template Based Shellcode

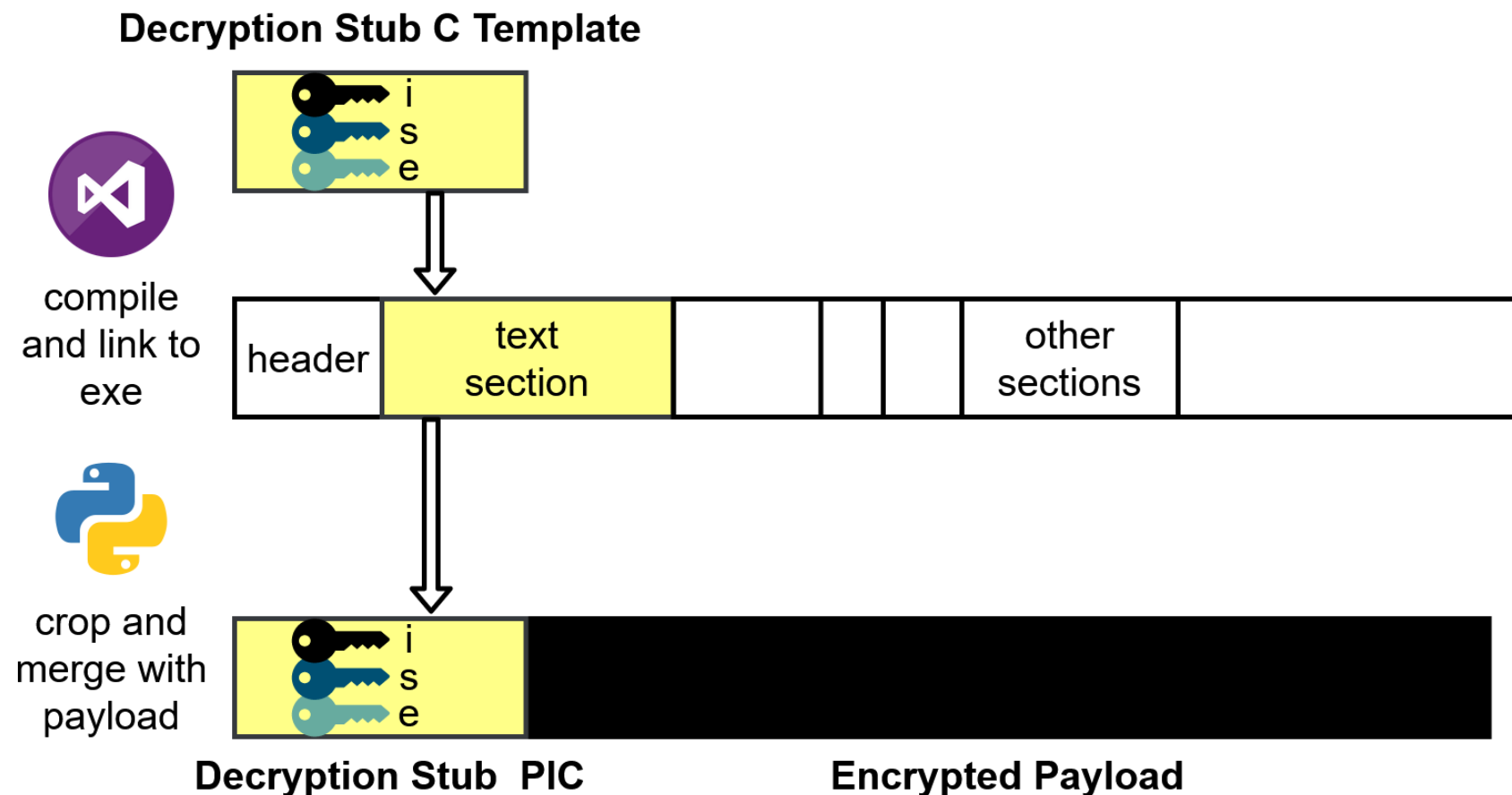
## Self-decrypting shellcode post-build script

### Decryption Stub C Template



# Template Based Shellcode

Self-decrypting shellcode post-build script



# Template Based Shellcode

## Suitable Memory

### Self-Decrypting Shellcode



Suitable memory is needed to run self-decrypting shellcode!  
And here we go again: in-memory artifacts ☹️

Is this really a problem?

# Template Based Shellcode

## Suitable Memory

### Self-Decrypting Shellcode



Suitable memory is needed to run self-decrypting shellcode!  
And here we go again: in-memory artifacts ☹️

Is this really a problem?

Short answer: No! 😊

Long answer: Depends - you must know what you're doing!

→ PIC itself in RWX memory is hard to identify as malicious



# Template Based Shellcode

## Gimme Suitable Memory

Masking Malicious Memory Artifacts: Part I – III

<https://www.forrest-orr.net/blog>

- VirtualAlloc
- VirtualProtect
- Create PE with RWX section
- Load DLL with RWX section
- <Something> Hollowing
- You name it
- ...

# Template Based Shellcode

Express Yourself – Just Say Yes!

Run shellcode directly

- Link into text section of loader PE
- Execute without prior allocation
- Not feasible for self-modifying code

# Template Based Shellcode

Express Yourself – Just Say Yes!

Run shellcode directly

- Link into text section of loader PE
- Execute without prior allocation
- Not feasible for self-modifying code

Protection?

- Obfuscation by replacing / inserting assembly instructions  
C → ASM → Obfuscation / Mutation → ASM → Binary [1]
- Situational awareness by conditional jumps

[1] <https://vxug.fakedoma.in/papers/VXUG/Exclusive/FromCprojectthroughassemblytoshellcodeHasherezade.pdf>

# Template Based Shellcode

## Loader PE Templates

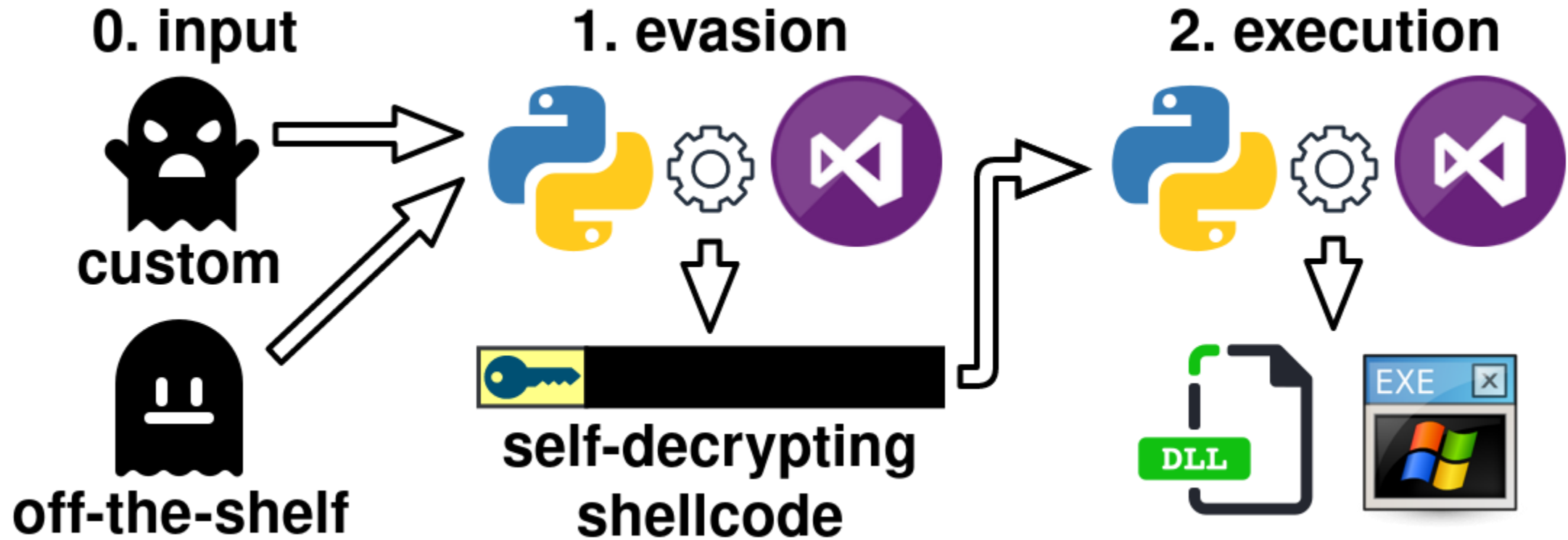
MinGW based approaches

<https://github.com/phra/PEzor>

<https://github.com/optiv/ScareCrow>

- One MSVS solution, multiple configurations
- Generic loader PE templates, mostly DLLs
- Various exports
- Various functionality (different hijacks / drop)
- Shellcode payload as header
- Some resource files for fun

# Git Together



# Git Together

## Why Gitlab CI/CD

- Modularity
- Flexibility
- Parameterization
- Ubiquitous



# Git Together

## Gitlab CI/CD Pipelines

- **Logic** in YAML files (and Python in our case)
- **Jobs** define what to do  
For example, jobs that compile or process (obfuscate)
- **Stages** define when to run the jobs  
For example, stages run different pre- or post-build evasion jobs after each other

# Git Together

## .gitlab-ci.yml Example Handlekatz

```
variables:
```

```
  CW_PROJECT: BruCon
```

```
  CW_PARAMETER: --pid 7688
```

```
  CW_EVASION: Crypto LoaderGen
```

```
  CW_CRYPTO_KEY_USER: KevinSmith
```

```
  CW_CRYPTO_SANDBOX: 1
```

```
  CW_CRYPTO_ITERATIONS: 100
```

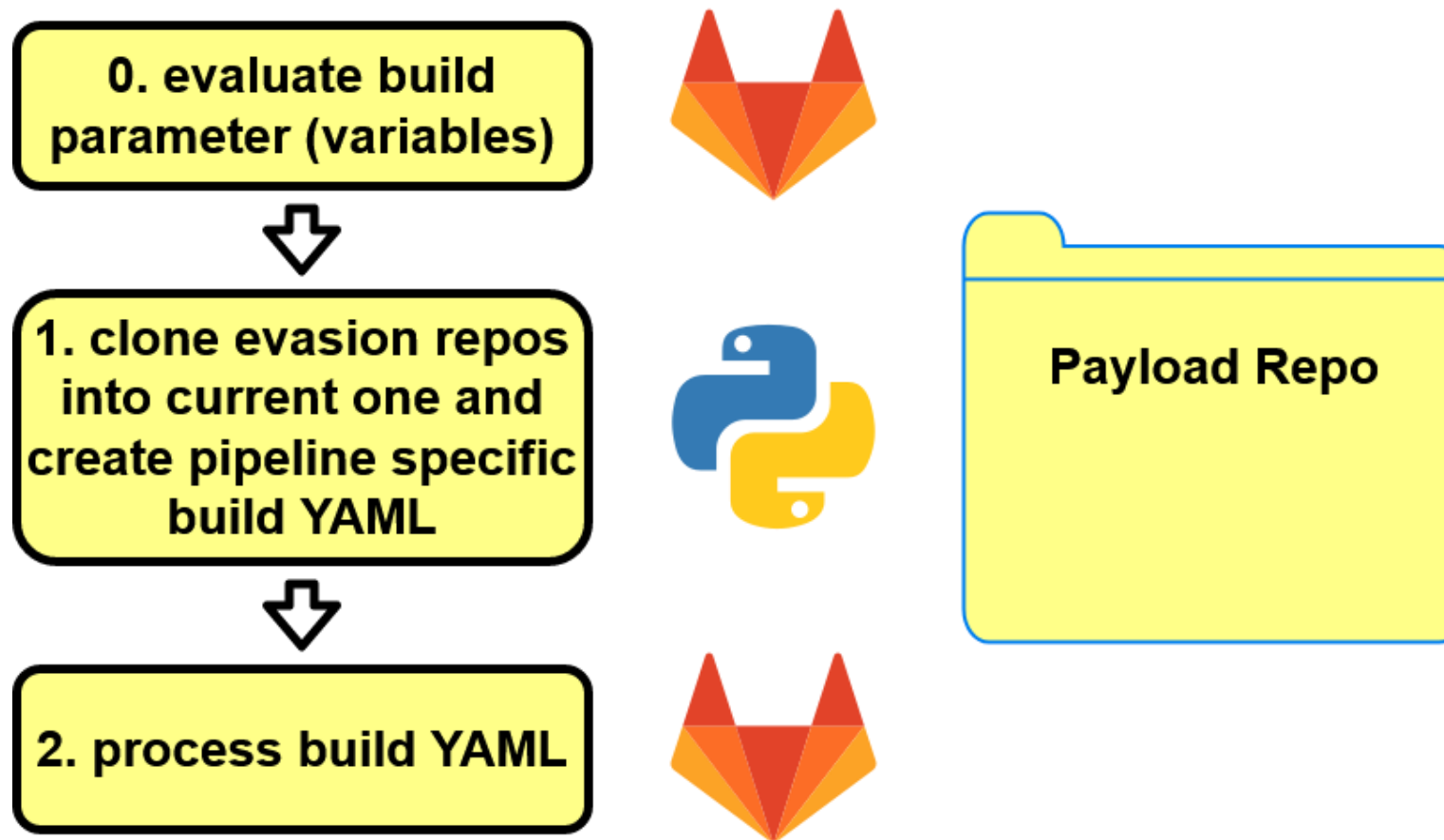
```
  CW_LOADER_TEMPLATE: EXE
```

```
  CW_LOADER_CONSOLE: 1
```



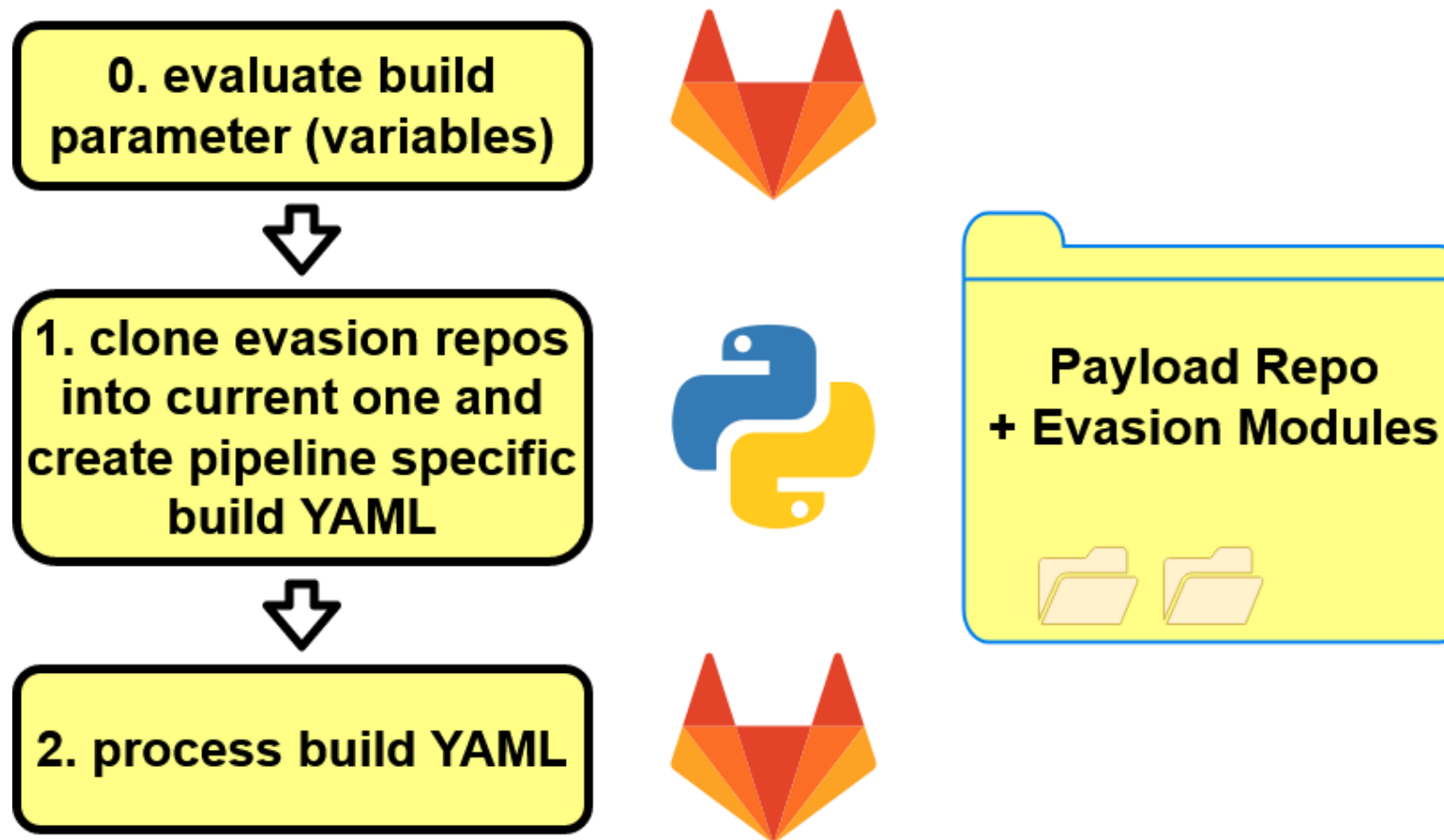
# Git Together

## Build Configuration / Preparation



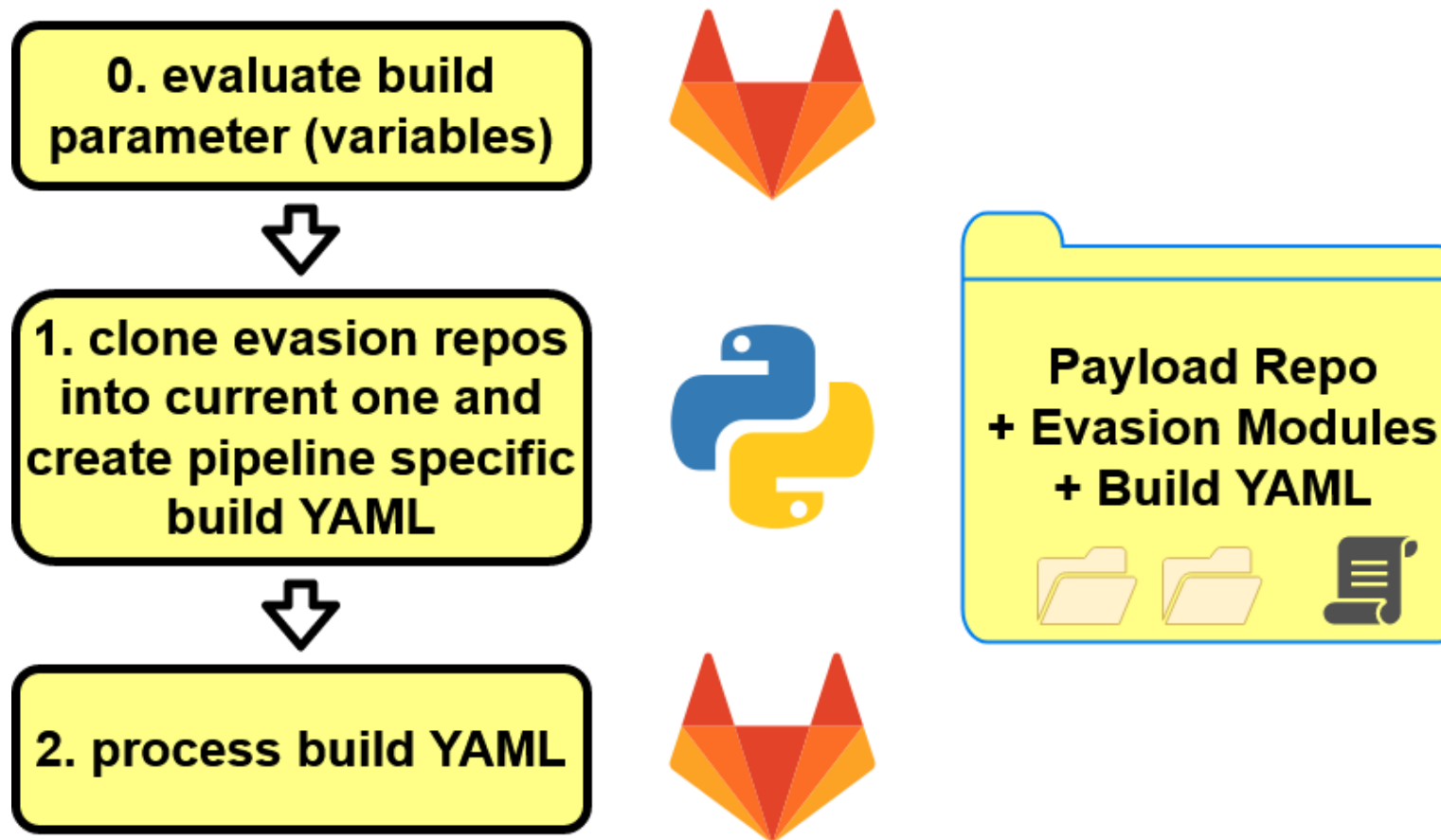
# Git Together

## Build Configuration / Preparation



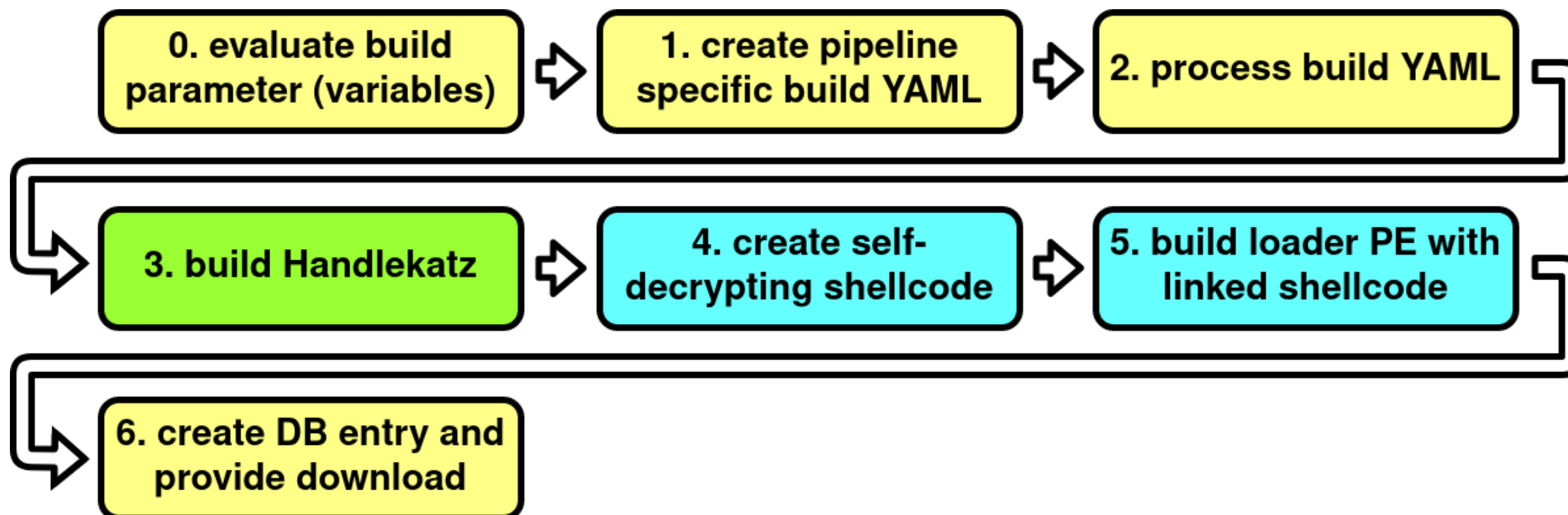
# Git Together

## Build Configuration / Preparation



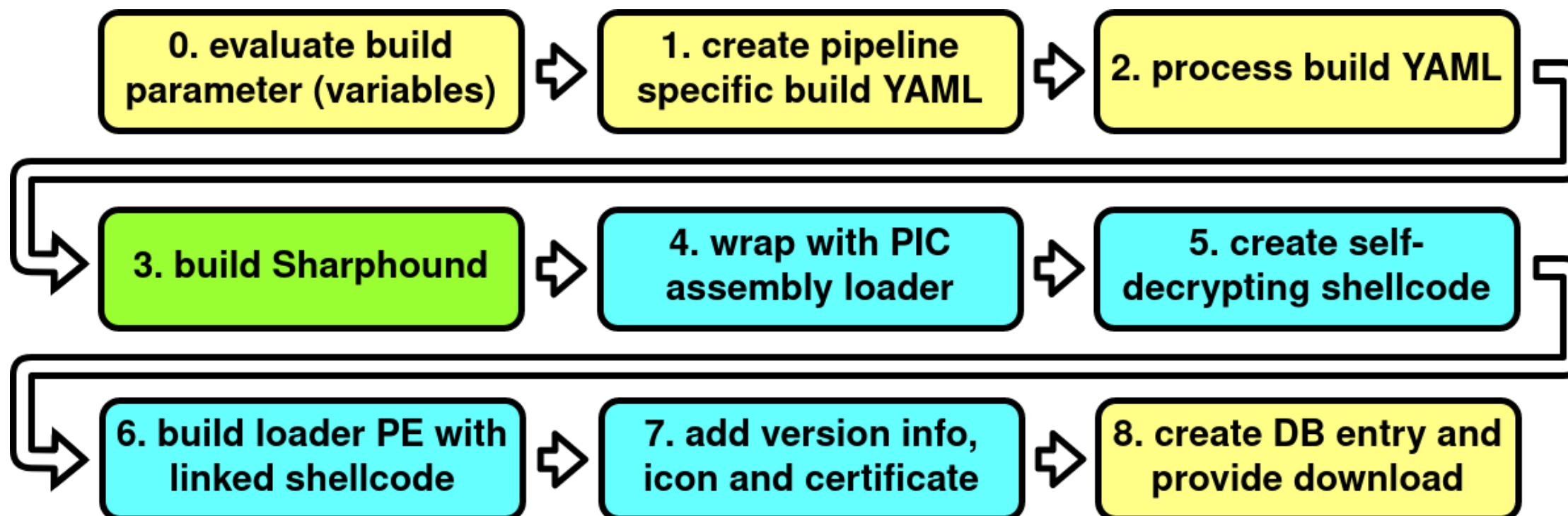
# Git Together

## Pipeline Example Handlekatz



# Git Together

## Pipeline Example Sharphound



# Git Together

## C2 Integration

Great talk about evasion CI/CD

[Dominic Chell - Offensive Development: Post Exploitation Tradecraft in an EDR World](#)

- **Agents:** Trigger pipeline process through CobaltStrike UI and download generated loader to your client
- **Tools / Payloads / Red Teaming as Code:** Trigger pipeline process through the CobaltStrike UI and upload artifact to a running Agent

# Demo

## C2 Integration

**Thank you for your attention!**  
**Questions?**

Check out on Twitter:

@danshaqfu  
@theflinkk

@niph\_  
@b00n10