MEMHUNTER

AUTOMATED HUNTING OF MALICIOUS MEMORY RESIDENT CODE AT SCALE

Marcos Oviedo (**@marcosd4h**)

Agenda

- About Us
- **WHY** hunting in memory
- **HOW** to hunt for memory resident malicious code
- **WHAT** to use during threat hunting process
- Demo

About Us

Marcos Oviedo - @marcosd4h

Infosec is my Passion!

Software Architect at McAfee

BSides Córdoba Argentina Organizer

bsidescordoba.org/en/

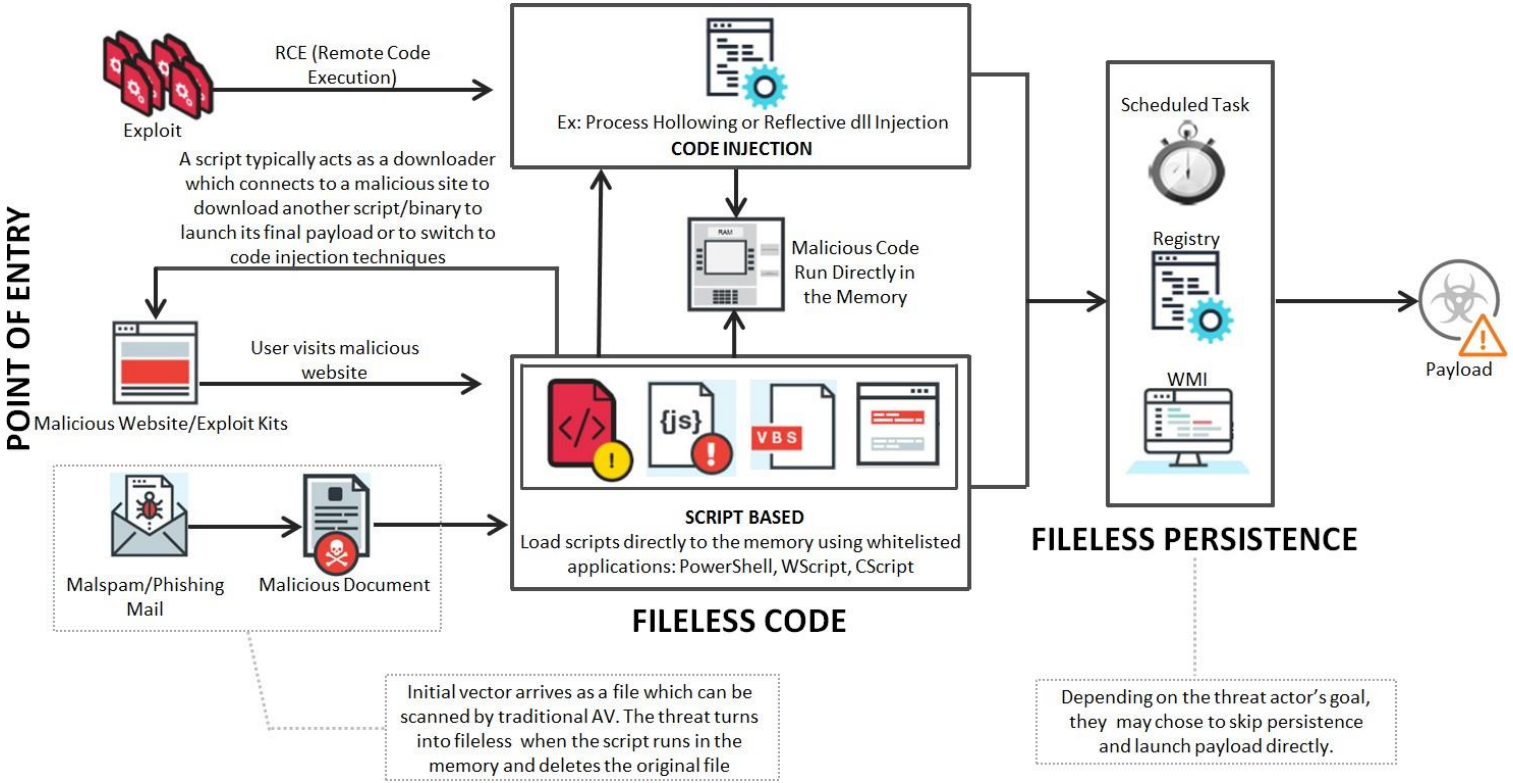
WHY hunting in memory

- **The current threat landscape has evolved in an attempt to evade specialized file-based detection techniques**

- **Fileless Attacks are one trend on this evolution**
  - Threats are now using process manipulation and built-in scripting mechanisms rather than dropping executable files
  - It is common for malware/adversarial code to run completely in memory
  - Harder to investigate

- **Process manipulation techniques are now a commodity**
  - Tech details of multiple code injection techniques have been public for years
  - Designed to evade endpoint security stacks (EPP/EDR)

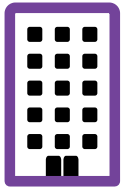
WHY hunting in memory (contd)



Source: Hunting for Ghosts in Fileless Attacks – SANS Institute

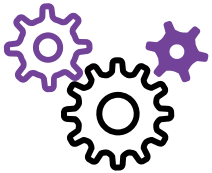
WHY hunting in memory at scale is difficult

Threat Hunters require a memory dumps and offline analysis to detect stuff in memory



On-going attacks are hard to detect on the complex and constantly changing enterprise environment

Threat Hunters rely on personal knowledge and intuition to digest and down-select data to analyze



Threat Hunters also have to be up to date with latest code injections techniques

HOW to hunt for memory resident malicious code

Process Manipulation - A technique used to compromise a legitimate process and have it execute malicious code

It might received different names based on the primitive used

Memory allocation	Memory writing	Execution	Windows Image load mechanisms

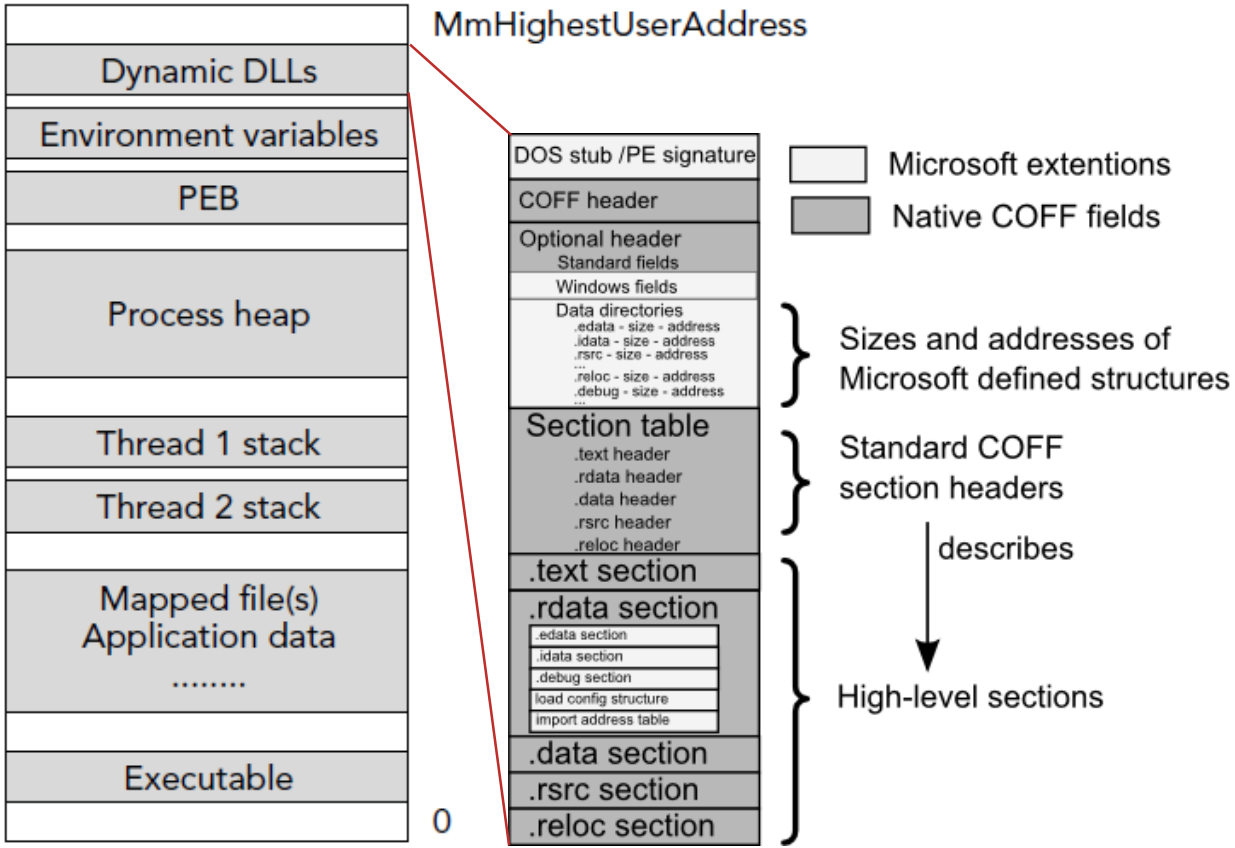
Win32 Process Internals

OS kernel allocates WCX regions for process object container

Process metadata and different sections will be allocated

Modules will be mapped and running modules list will be set

Threads objects will be instantiated and assigned with running address



Win32 Process Internals (contd)

Process Properties

- Parent Process
- Process Name
- Process Signer
- Signature Validation
- Command Line
- Current working directory

Thread Properties

- Thread ID
- Start Address
- Start Address Module
- Priority

Memory Properties

- Base Address
- Type of Memory
- State of Memory
- Size
- File
- Initial Permissions
- Permissions

Win32 Process Internals Anomalies

A little spoiler for you - You can hunt for artifacts, process anomalies and side-effects caused by process manipulation techniques by looking at

Thread Start Address

- No module associated to thread start address
- Memory permissions

Memory Properties

- Regions of memory with RWX permissions and private commit
- Odd AllocationProtect - Protect pair

Memory content

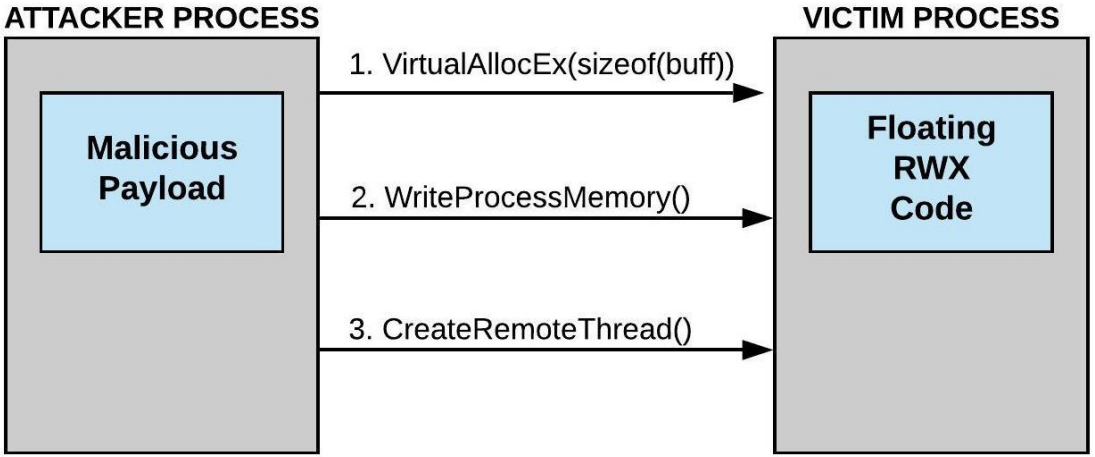
- Signs of PE file
- Strings associated with attack tradecraft

Code Injection Example: Reflective DLL Injection

Attackers allocate a remote RWX memory section into a victim process, then copy a bootstrapping shellcode and a payload blob containing regular DLL code, to finally spawn a remote thread that executes from recently allocated memory region.

- The code maps itself into victim process memory (Allocate memory, map sections, resolve imports, fixup relocations, call entry, etc)
- It is very handy because it does not require a DLL living on disk
- It can be considered as a mix of shellcode Injection and DLL Injection.
- One of the most notable users of this library is Metasploit’s Meterpreter and Empire

Code Injection Example: Reflective DLL Injection



Anomalies

New thread running from floating code

Unbacked stack frames on new thread callstack

Big and ugly RWX section

Defense Evasion Example: Process Hollowing

Attackers create a legit victim process in a suspended state, then de-allocate one of the victim process modules, so it can replace its memory with malicious code.

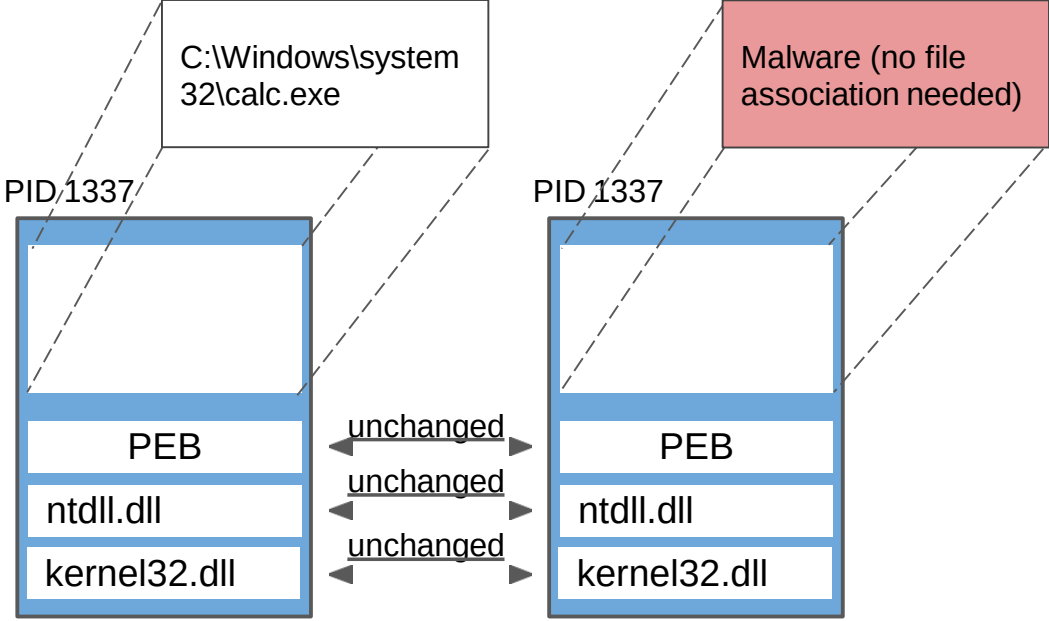
Once the memory is replaced, the main thread is resumed so new code can run

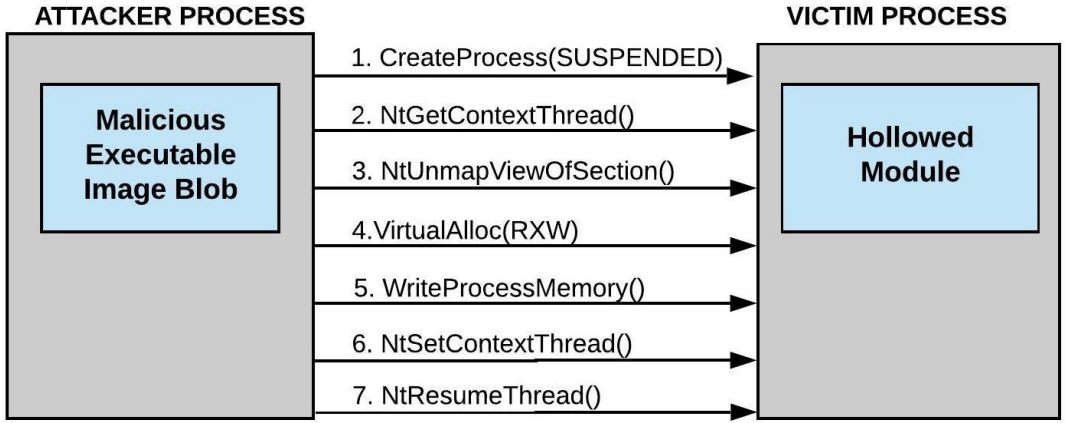
- It is typically used for AV/EDR evasion
- Legitimate process is loaded to act as a container for hostile code

Defense Evasion Example: Process Hollowing (contd)

Anomalies

Mismatch between PE header fields from module on disk vs counterpart on memory





WHAT can be used during threat hunting process?

Introducing MEMHUNTER

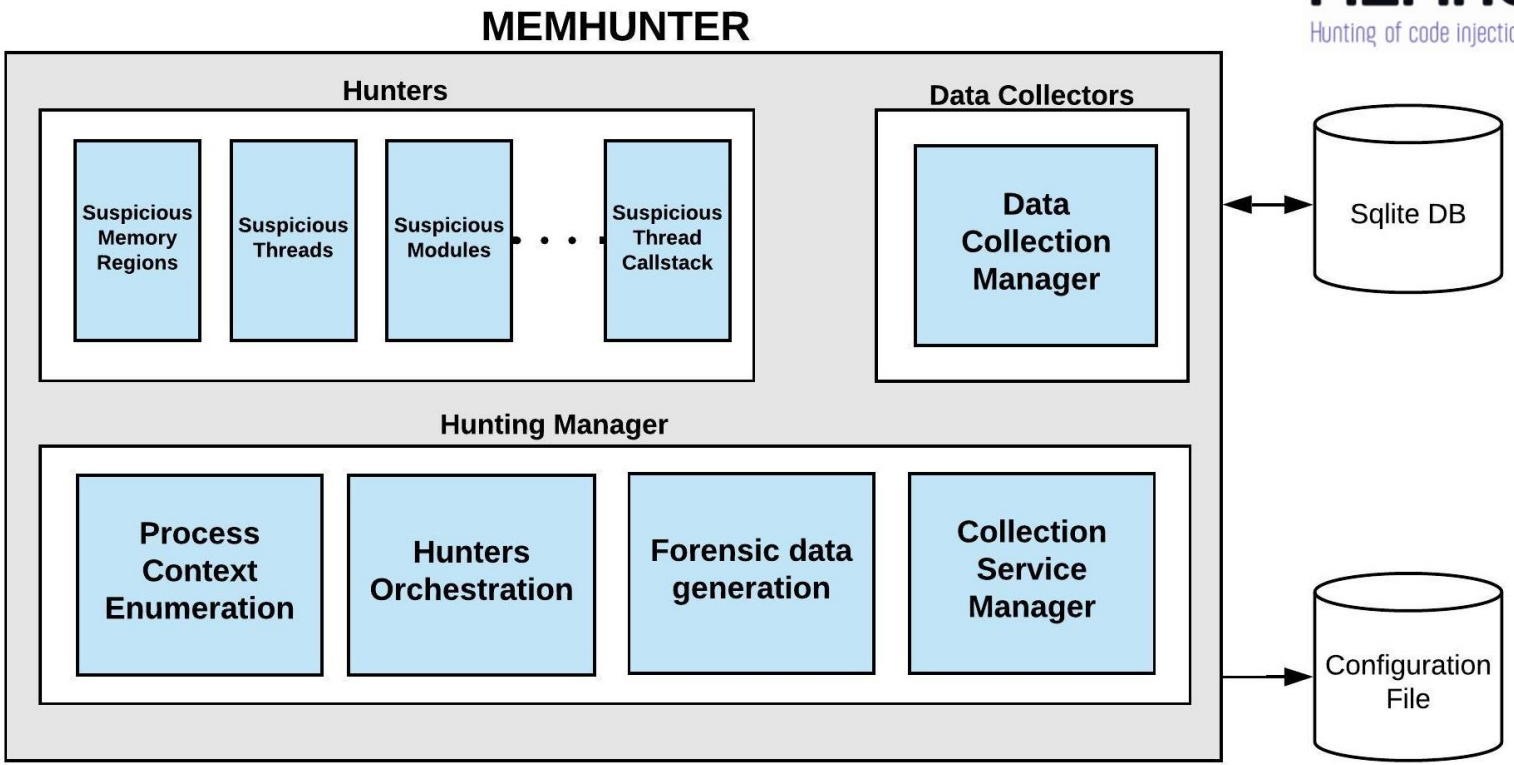
- It is an standalone binary that gets itself deployed as a windows service
- It uses a set of memory inspection heuristics and ETW data collection to find side-effects, footprints and process anomalies left by common injection techniques.
- It reports forensic information on findings through console or event logs

Memhunter is a free tool that automates the hunting of memory resident malicious code at scale, improving the threat hunter analysis process and remediation times

https://github.com/marcosd4h/memhunter

Memhunter Architecture

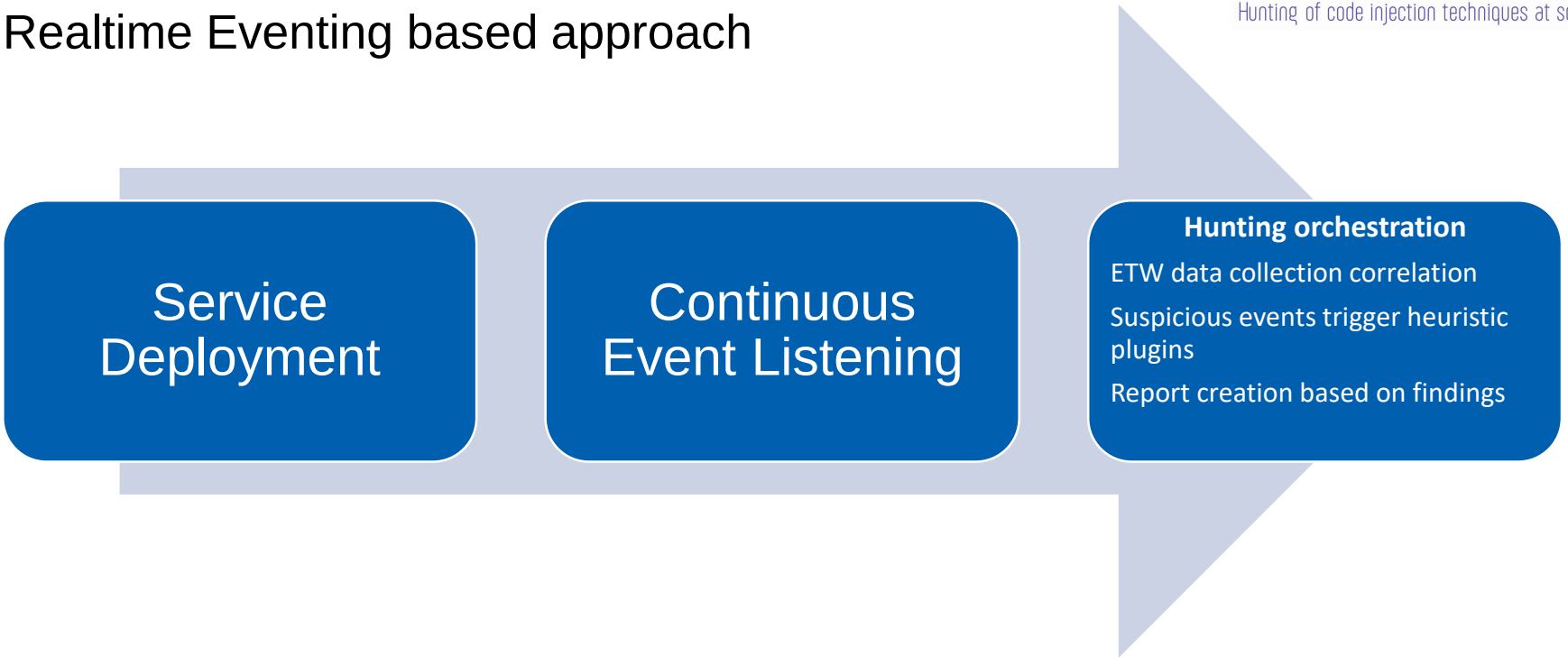
MEMHUNTER
Hunting of code injection techniques at scale



Memhunter Hunting Process

MEMHUNTER
Hunting of code injection techniques at scale

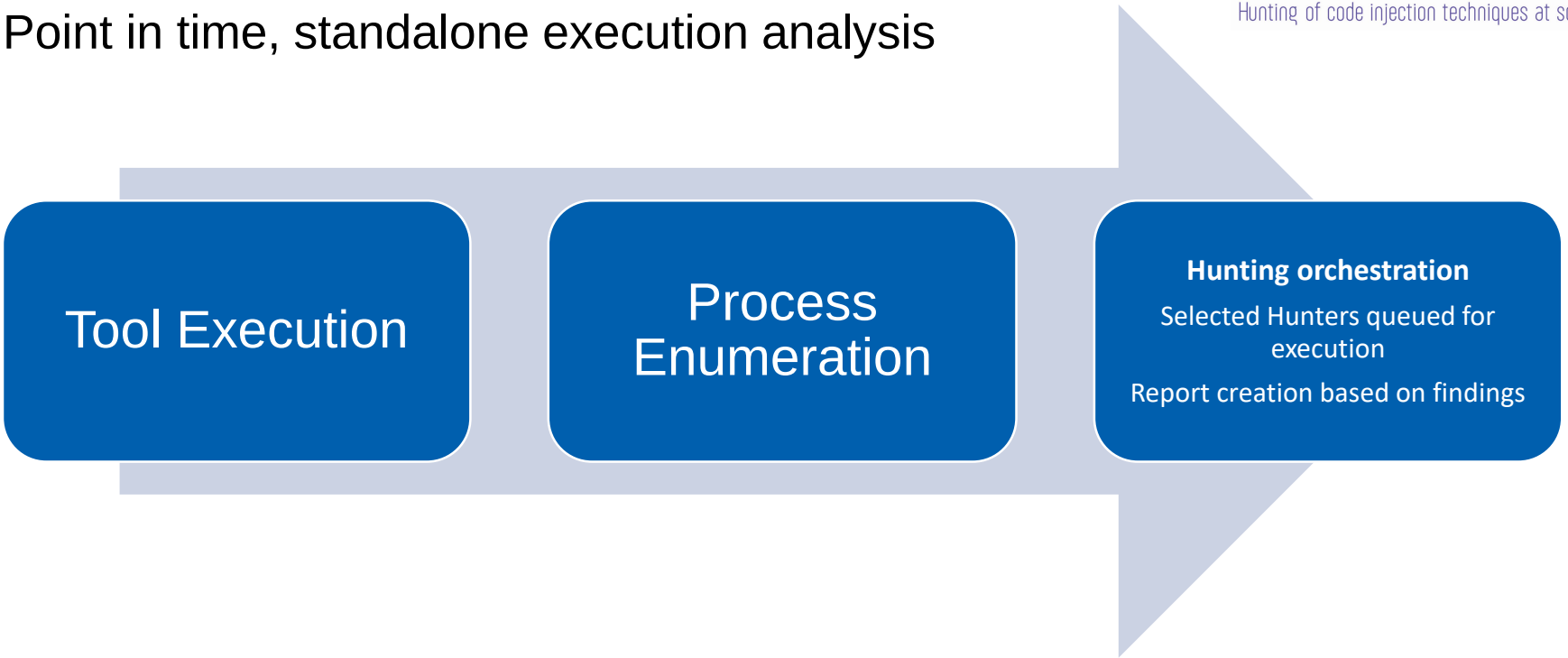
Realtime Eventing based approach



Memhunter Hunting Process (contd)

MEMHUNTER
Hunting of code injection techniques at scale

Point in time, standalone execution analysis



Memhunter Current Functionalities

- 9 hunter heuristics included (see next slide)
- 15 code injection techniques implemented through minjector test tool
- ETW data collection of suspicious events used for heuristic triggering
- Windows Event Log generation
- Exclusion of baseline detection
- Basic forensic information
- Sqlite storage integration

Tool is still Work in Progress!
Expect new things to appear in the future

Hunter Heuristics

Suspicious Threads

- It inspects memory regions associated with threads looking for RWX flags, starting with memory regions associated to thread base address
- It looks for unbacked or floating code living in the memory regions of the process

Suspicious Call stack

- It checks the process thread call stack looking for unbacked symbols (not memory-mapped code, aka floating code)

Suspicious Memory regions

- It inspects memory regions of the entire process looking for RWX flags
- It checks for PE header over these regions (fuzzy PE match)

Hunter Heuristics

Suspicious Base Address

- Base Address of main module (.exe) is private commit and marked as RWX. This should never happen, it should be memory-mapped always

Suspicious Pairs

- It looks for threads running from memory regions with odd memory allocation pairs: Memory was allocated with RWX and now is RX

Suspicious Modules

- It looks for modules that associated with RWX memory regions

Hunter Heuristics

Suspicious hollowed modules

- It performs in-memory vs on-disk comparison of PE header fields of different modules on the process modules (linker version, entry points, size of code, etc)

Suspicious Registry Persistence

- It looks for common registry injection/persistence techniques such as IFEO (Image File Execution Options), Appinit_DLL and AppCertDLLs

Suspicious Shellcodes

- It looks for RXW memory regions that start with well-known x86 or x64 prologues opcodes: Shellcode baby!

Hunter Heuristics (contd)

Suspicious PEB modification (status: will be released after Defcon)

- PEB Unlinking. It looks for hidden DLLs modules. It compares what is reporting by win32 APIs with what can be obtained from the kernel (kernel call through EPROCESS)

Suspicious Spoofing (status: service mode – will be released after Defcon)

- It cross check process cmdline from PEB with cmdline from ETW kernel process provider to look for signs of cmdline spoofing
- It cross check process parent PID from NtQuerySystemInformation() with process genealogy obtained from ETW kernel provider to look for signs of parent PID spoofing

ETW Eventing

- Process Creations (Microsoft-Windows-Kernel-Process)
- Registry Operations (Registry operations at Microsoft-Windows-Kernel-Registry and AE53722E-C863-11d2-8659-00C04FA321A1
- Threads Operations (thread kernel provider at 3d6fa8d1-fe05-11d0-9dda-00c04fd7ba7c)
- Image Load Operations (Image load provider at 2cb15d1d-5fc1-11d2-abe1-00a0c911f518)
- Virtual Alloc Operations (Page Fault Provider at 3d6fa8d3-fe05-11d0-9dda-00c04fd7ba7c)
- Requires MS signing
  - Kernel Audit APIs usage (Microsoft-Windows-Kernel-Audit-API-Calls)
  - Future usage - Only on win10 - Suspicious APIs via Microsoft-Windows-Threat-Intelligence

Forensic Information

When possible, the fields below will be reported

- Suspicious PID
- Suspicious TID
- Thread integrity levels
- Abnormal user tokens
- SE Debug privileges. Debug Token
- Integrity levels
- Thread BASE Priority (Thread have more priority than other threads)
- Token Integrity level, Enabled Privileges, SID/Username, Logon Session, Logon Type, Authentication Package used, etc
- Group SID

Demo Time

Questions?
THANKS!

MEMHUNTER

Hunting of code injection techniques at scale