Hunting For Memory Resident Malware

Memhunter tool

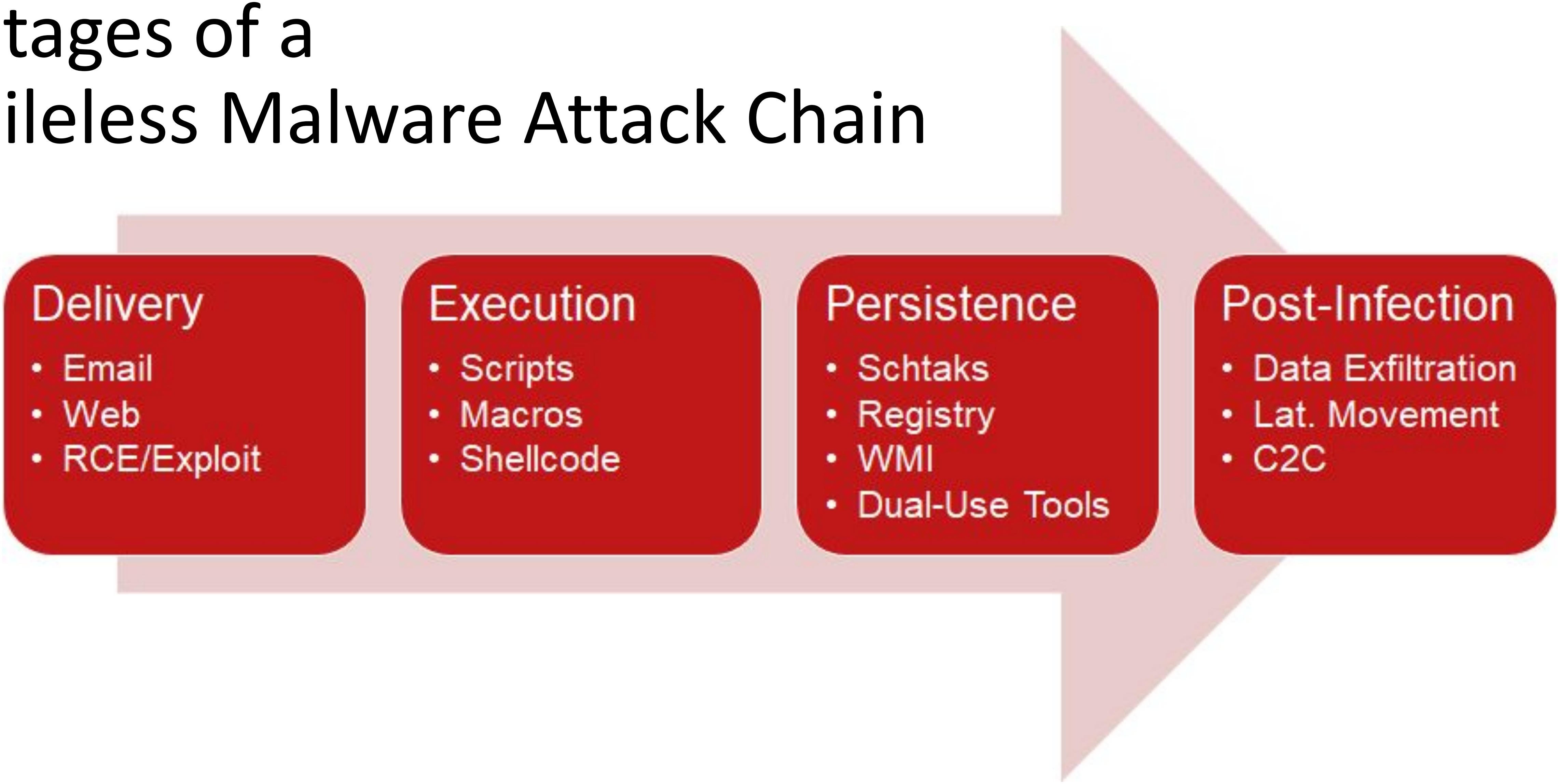
https://github.com/marcosd4h/memhunter

Marcos Oviedo – McAfee SW Architect

Agenda

- Problem
- Tool Summary
- Challenges tool wants to address
- Memhunter key takeaways
- Memhunter architecture
- Memhunter Hunting Process
- Current functionalities

Stages of a
Fileless Malware Attack Chain



Delivery

- Email
- Web
- RCE/Exploit

Execution

- Scripts
- Macros
- Shellcode

Persistence

- Schtaks
- Registry
- WMI
- Dual-Use Tools

Post-Infection

- Data Exfiltration
- Lat. Movement
- C2C

Tool summary

Memhunter automate the hunting of memory resident malware at scale, improving the threat hunter analysis process and remediation times

Memhunter in a nutshell

- 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 footprints left by common injection techniques.
- Forensic information on findings gets reported through console or event logs for forwarding

Types of Fileless Malware Techniques

Fileless Manipulation

Memory resident malware

Non-PE/Scripts Based

Dual-Use Tools

Types of Fileless Malware Techniques - Memory Resident Malware

- Malware code residing in the memory of a legit running processes
- Malware distributes and inject itself into critical processes that are critical to windows operation and that cannot be whitelisted, scanned or killed
- It relies on multiple injection techniques to evade detections
  - Classical injection through remote thread creation
  - APC injection
  - Atom bombing
  - Process Hollowing
  - Local Shell code injection + thread creation
  - Reflective DLL loading
  - DLL module
  - Gargoyle

Challenges tool wants to address

Memory resident malware has become increasingly sophisticated

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

Threat Hunters rely on personal knowledge and intuition to digest enterprise data and detect problems

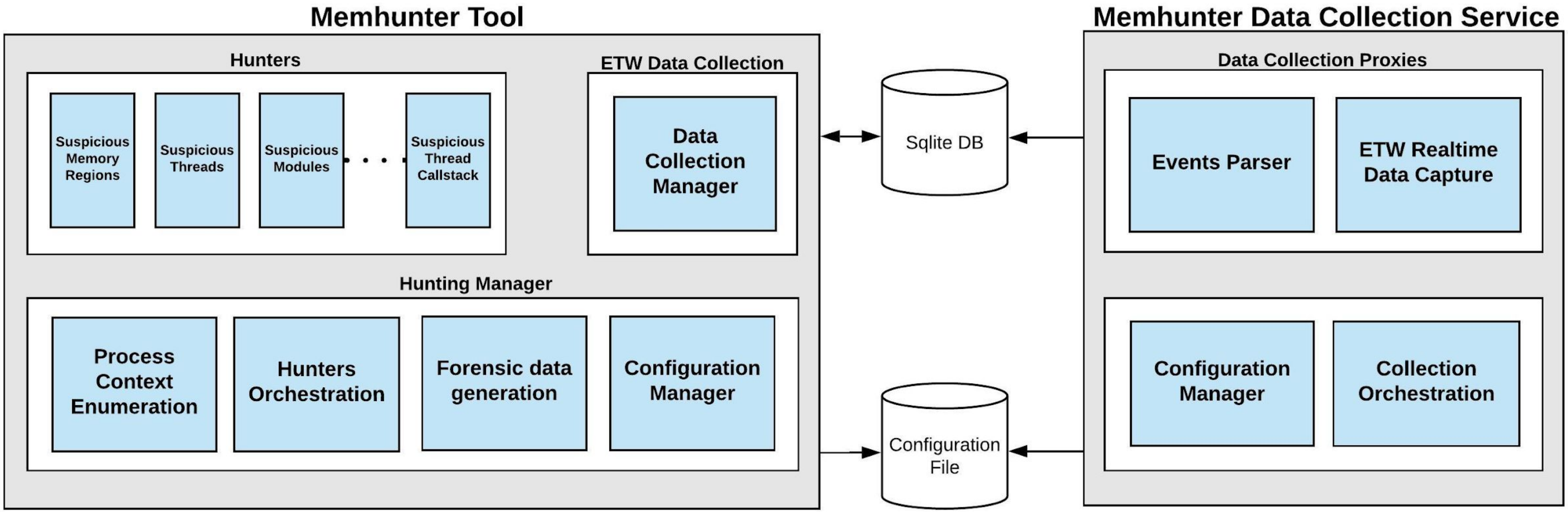
Threat Hunters expertise is critical and needs to be up-to-date to cope with latest threats

Threat Hunters need an automated way to detect fileless threats at scale

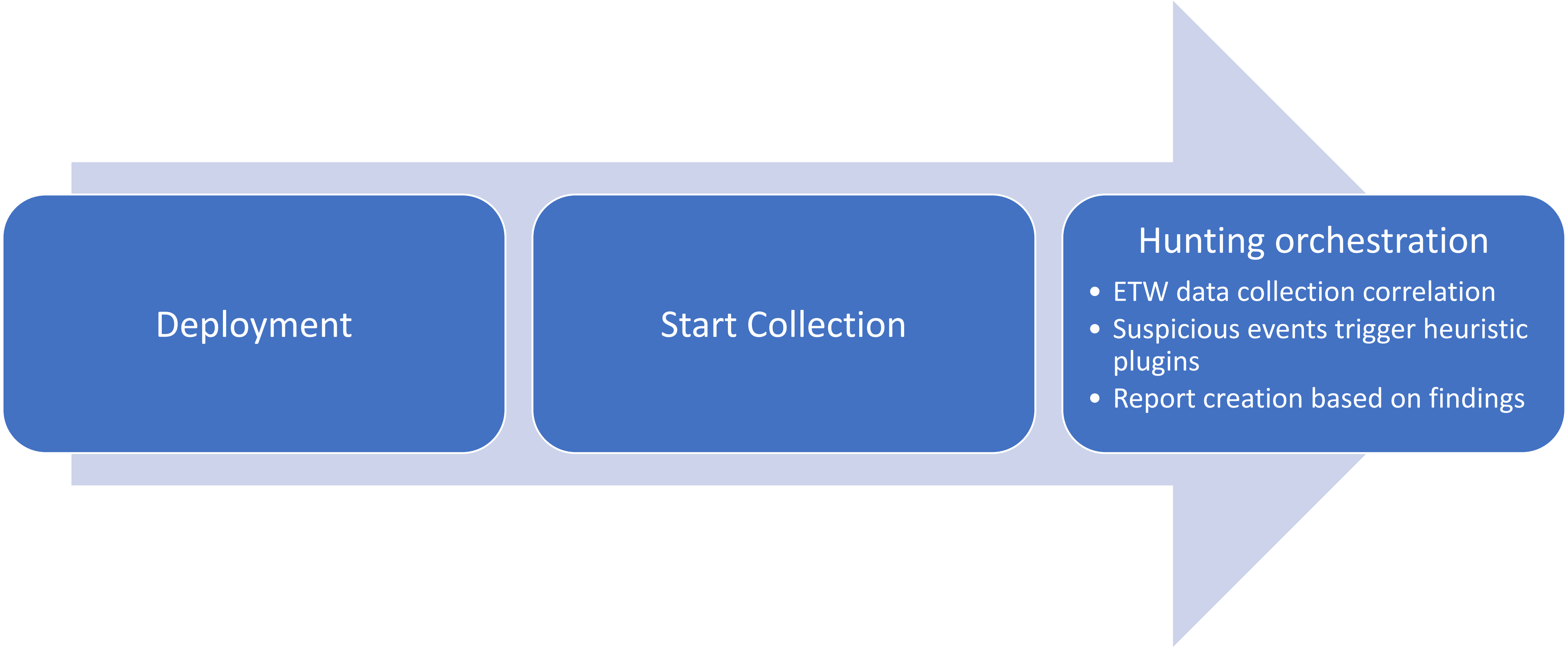
Memhunter key takeways

- Automates the detection of in-memory fileless attacks
- Improve hunting analysis and remediation times
- Self contained binary that can be deployed and managed at scale
- It does not use memory dumps
- It purely relies on memory inspection to do its work
- It does not require complex infrastructure

Memhunter Architecture



Memhunter hunting process



Deployment

Start Collection

Hunting orchestration

- ETW data collection correlation
- Suspicious events trigger heuristic plugins
- Report creation based on findings

Current functionalities

- 9 hunter heuristics included (see next slide)
- 15 code injection techniques implemented on 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

ETW Suspicious Events

- 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)
- Virtual Alloc Operations (Page Fault Provider at 3d6fa8d3-fe05-11d0-9dda-00c04fd7ba7c)
- Image Load Operations (Image load provider at 2cb15d1d-5fc1-11d2-abe1-00a0c911f518)
- Kernel Audit APIs usage (Microsoft-Windows-Kernel-Audit-API-Calls)
- Future usage - Only on win10 - Suspicious APIs via Microsoft-Windows-Threat-Intelligence

Hunters (Hunting Heuristics)

- **Suspicious Modules** (status: implemented)
  - Look for Modules that are associated with RWX memory regions

- **Suspicious Threads** (status: implemented)
  - Inspect memory regions associated with threads looking for RWX flags, starting with memory regions associated to thread base address
  - Unbacked or Floating code living in the memory regions of the process

- **Suspicious Memory regions** (status: implemented)
  - Inspect memory regions of the entire process looking for RWX flags
  - Check PE header over these regions (fuzzy PE match)

Hunters (Hunting Heuristics) (contd)

- **Suspicious Call stack** (status: implemented)
  - Check call stack of threads looking for unbacked symbols (floating code)
- **Suspicious Base Address** (status: implemented)
  - Base Address of main module (.exe) is private: commit and marked as RWX (should never happen, it should be memory mapped always. Detects Process Hollowing
- **Suspicious Exports** (status: implemented)
  - Look for exports like “ReflectiveLoader()” on the list of modules/exe exports

Hunters (Hunting Heuristics) (contd)

- **Suspicious hollowed modules** (status: implemented)
  - In-memory vs on-disk comparison
  - Comparing linker version, entry points, size of code (PE header). LDR vs PEB.

- **Suspicious Registry Persistence** (status: implemented)
  - It looks for common registry injection/persistence techniques such as IFEO (Image File Execution Options), Appinit_DLL and AppCertDLLs

- **Suspicious Shellcodes** (status: implemented)
  - It looks for RXW memory regions that starts well known x86 or x64 prologues opcodes

Hunters (Hunting Heuristics) (contd)

- **Suspicious PEB modification** (status: code being tested - not pushed)
  - PEB Unlinking. Look for hidden DLLs modules. Compare what is reporting by win32 APIs with what can be obtained from the kernel (kernel call through EPROCESS)

- **Suspicious CLR Reflection** (status: code being tested - not pushed)
  - Detect .NET loaded serialization (System.Reflection.Assembly.Load(byte[]).
  - It looks for CLR module loaded without file backing. Memory regions associated is MEM_MAPPED, RW and MZ/PE at address.

- **Suspicious Spoofing** (status: code being tested - not pushed)
  - It cross check process cmdline from PEB with cmdline from ETW kernel 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

Forensic information

- Suspicious PID
- Suspicious TID
- Thread integrity levels
- Abnormal user tokens
- SE Debug privileges. Debug Token
- Integrity levels
- EoP tokens
- Unique Thread token
- 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

Thanks!