Entwicklung und Integration eines Memory Mapping Plugins für Volatility3 zur forensischen Analyse von Linux-Systemen

vorgelegt von:

Richard Heinz

Studiengang: M.Sc. Informatik

Betreuer: Maximilian Eichhorn

2. Juli 2024

Kurzfassung

Die Entwicklung des memmap-Plugins für Volatility3 zur Analyse von Linux-Speicherabbildern stellt mehrere zentrale Herausforderungen dar. Ein Kernaspekt ist die korrekte Erstellung und Konfiguration der Entwicklungsumgebung, insbesondere die Generierung präziser Symboltabellen für den spezifischen Linux-Kernel. Dies erweist sich als komplex, da es die Extraktion von Debug-Informationen aus Kernel-Dateien und die Umwandlung dieser in das benötigte JSON-Format erfordert.

Ein weiterer wichtiger Punkt ist die Bewältigung von Kompatibilitätsproblemen zwischen verschiedenen Volatility- und Linux-Kernel-Versionen. Dies umfasst die Anpassung des DWARF-Parsers in Volatility2, um neuere Datentypen zu unterstützen, sowie Modifikationen zur korrekten Verarbeitung von Kernel-Strukturänderungen. Die Implementierung des Plugins selbst erfordert ein tiefes Verständnis der Volatility3-Plugin-Architektur, insbesondere der Anforderungsdefinition und der Interaktion mit dem Linux-Speichermodell. Diese Aspekte sind entscheidend für die erfolgreiche Entwicklung eines funktionsfähigen und robusten memmap-Plugins für Volatility3.

Die Implementierungsphase des Memmap-Plugins für Volatility3 umfasst drei zentrale Methoden: get_requirements(), run() und _generator(). Die get_requirements()-Methode definiert die spezifischen Anforderungen des Plugins, einschließlich des benötigten Kernel-Moduls, des PsList-Plugins, optionaler PID-Filterung und Optionen zum Dumpen und Aufteilen von Speichersegmenten. Die run()-Methode initiiert die Plugin-Ausführung, erstellt einen TreeGrid zur strukturierten Darstellung der Memory Mappings und nutzt eine Filterfunktion zur gezielten Prozessanalyse.

Die _generator()-Methode bildet das Herzstück des Plugins, indem sie sequenziell Memory Mapping-Details für den ausgewählten Prozess generiert. Sie verarbeitet Prozessinformationen, handhabt mögliche Ausnahmen und ermöglicht das optionale Dumpen von Speichersegmenten. Die Methode extrahiert detaillierte Mapping-Informationen und kann diese bei Bedarf in kleinere Abschnitte aufteilen. Diese Implementierung gewährleistet eine flexible und genaue Analyse der Speicherstrukturen, angepasst an die spezifischen Anforderungen forensischer Untersuchungen.

Inhaltsverzeichnis

1. Einleitung

Die vorliegende Arbeit konzentriert sich auf die Entwicklung eines *memmap*-Plugins für das Volatility3-Framework. Besondere Aufmerksamkeit gilt der Installation, Konfiguration und Sicherstellung der Funktionalität sowie dem Entwicklungsprozess und umfassenden Tests des Plugins. Ebenso werden die während der Entwicklung gewonnenen Erkenntnisse und Herausforderungen diskutiert sowie die praktische Anwendung des Plugins für forensische Untersuchungen beleuchtet.

Die Entscheidung, das *memmap*-Plugin zu implementieren, beruht auf seiner zentralen Rolle im Volatility2-Framework. Bisher ist dieses Plugin jedoch nur für das Windows-Betriebssystem in Volatility3 verfügbar und nicht für Linux. Daher ist die Integration dieses Plugins in die aktuellste Version von Volatility von entscheidender Bedeutung. Sie zielt darauf ab, die Funktionalität und Analysemöglichkeiten der Software zu erweitern und an die Bedürfnisse der forensischen Untersuchung anzupassen.

2. Hintergrund

2.1. Volatility

Volatility ist ein leistungsfähiges und weit verbreitetes Open-Source-Framework, das speziell für die Analyse von Speicherauszügen (Memory Dumps) von Computersystemen entwickelt wurde. Es ermöglicht Forensikern und Sicherheitsexperten, detaillierte Einblicke in den Zustand des Arbeitsspeichers (RAM) eines Computers zu gewinnen. Diese Analyse ist entscheidend, um verdächtige oder schädliche Aktivitäten zu erkennen, die während des Betriebs des Systems stattgefunden haben könnten. Volatility ist bekannt für seine Fähigkeit, komplexe Informationen aus Speicherauszügen zu extrahieren, darunter laufende Prozesse, geladene Module, Netzwerkverbindungen, offene Dateien und viele weitere Systemdetails. Das Framework bietet eine modulare Architektur, die es ermöglicht, verschiedene Analysewerkzeuge und Plugins zu integrieren, um spezifische Forensik- und Sicherheitsanforderungen zu erfüllen.

Die aktuelle Version ist *Volatility 3*. *Volatility 3* hebt sich deutlich von seiner Vorgängerversion *Volatility 2* ab. Während *Volatility 2* als robustes Werkzeug für die Analyse von Speicherauszügen angesehen wurde, hat Volatility 3 eine grundlegende Neuausrichtung erfahren, indem es als Bibliothek konzipiert wurde. Diese neue Architektur stellt sicher, dass alle Komponenten unabhängig voneinander sind und jeder Plugin-Ausführungskontext in einem eigenständigen Objekt, das von der ContextInterface abgeleitet ist, verwaltet wird. Im Gegensatz zu *Volatility 2*, das auf Profilen basierte, bietet Volatility 3 eine erweiterte Symbolbibliothek und ermöglicht die dynamische Generierung von Symboltabellen basierend auf den spezifischen Eigenschaften des vorliegenden Speicherabbilds. Diese Flexibilität erleichtert die präzisere Identifizierung von Systemstrukturen durch vordefinierte Offsets, die aus offiziellen Debugging-Informationen stammen.

Ein weiterer wichtiger Unterschied liegt im Objektmodell: In *Volatility 3* werden Objekte direkt von ihren entsprechenden Python-Gegenstücken abgeleitet. Dieser Ansatz verbessert nicht nur die Flexibilität bei der Analyse, sondern steigert auch die Geschwindigkeit der Ausführung, da die nativen Methoden der Python-Objekte direkt genutzt werden können. Diese Neuerungen machen *Volatility 3* zu einer leistungsfähigeren Plattform für forensische Analysen, indem sie die Komplexität reduzieren und gleichzeitig die Effizienz bei der Verarbeitung von Speicherabbildern erhöhen (vgl.Volatility Project [2024]).

2.2. memmap Plugin

In Volatility liefert das *memmap*-Plugin eine detaillierte Karte des physischen und virtuellen Speichersystems eines Computers. Es bietet eine umfassende Übersicht über die Speicherbereiche, die von Prozessen, Kernel-Objekten und anderen Systemressourcen genutzt werden. Diese Funktion ist entscheidend für forensische Analysen, da sie ermöglicht, sowohl den physischen als auch den virtuellen Zustand des Speichers zu verstehen. Durch die Bereitstellung von Informationen über virtuelle Adressen ermöglicht das Plugin Analysten, die Speichernutzung einzelner Prozesse zu verfolgen, die Speicherzuordnung zu untersuchen und potenzielle Anomalien oder Sicherheitsvorfälle zu identifizieren.

2.3. Entwicklungsumgebung

Die Entwicklung des Plugins erfolgte in einer virtuellen Maschine der Software VirtualBox in der Version 7.0.18 mit dem Betriebssystem Kali GNU/Linux in der Version 2024.2 mit dem Kernel 6.8.11-amd64.

Für die Durchführung des Projektauftrages wurden verschiedene Tools und Frameworks verwendet:

- **Analysewerkzeuge**:
  - **avml** in der Version 0.11.2
  - **Volatility Foundation Volatility Framework** in der Version 2.6.1 und **Volatility 3 Framework** in der Version 2.7.1
- **Programmiersprachen und -umgebungen**:
  - **Python3** in der Version 3.11.9 und **Python2** in der Version 2.7.18
  - **pip** in der Version 20.3.4
- **Bibliotheken und Tools**:
  - **distorm3** in der Version 3.5.3
  - **yara** in der Version 4.5.1
  - **pycrypto** in der Version 2.6.1
  - **dwarf2json** in der Version v0.8.0
- **Weitere Werkzeuge**:
  - **go** in der Version 1.22.4
  - **dwarfdump** in der Version 20210528

Die Auflistung der Tools und Frameworks mit ihren Versionen zielt darauf ab, Transparenz zu gewährleisten und die Reproduktion der Ergebnisse zu ermöglichen. Durch die genaue Angabe der Versionen wird sichergestellt, dass die gleichen Werkzeuge in identischer Konfiguration verwendet werden können, um die Konsistenz und Nachvollziehbarkeit der Ergebnisse zu unterstützen

3. Entwicklungsprozess

Dieses Kapitel beginnt mit der Vorbereitungsphase, in der die notwendigen Tools installiert und konfiguriert werden, um die Grundlage für die Entwicklung des Plugins zu schaffen. Im Anschluss wird die Implementierung des Plugins detailliert behandelt, wobei besonderes Augenmerk auf die Umsetzung der Funktionalitäten gelegt wird. Darüber hinaus werden umfangreiche Tests durchgeführt, um sicherzustellen, dass das Plugin zuverlässig arbeitet und den Anforderungen entspricht.

3.1. Vorbereitung

3.1.1. Speicherabbild Kali Linux

Zur Überprüfung der Funktionalität der Volatility-Plugins ist es erforderlich, zunächst ein Speicherabbild des Hauptspeichers des Systems anzufertigen. Hierfür wurde die Software *AVML* (Acquisition of Volatile Memory for Linux) installiert.

Für die Installation von *AVML* ist es notwendig, die neueste Version des Tools über das Git-Repository herunterzuladen. Nach dem Herunterladen navigiert man in das Verzeichnis, wo sich die ausführbare Datei befindet, und stellt sicher, dass die notwendigen Berechtigungen zum Ausführen der Datei gesetzt sind. AVML erfordert keine Kompilierung und ist sofort einsatzbereit, was die Installation vereinfacht (vgl. A.3: AVML Installation).

AVML ist ein Open-Source-Tool, das in der Programmiersprache Rust entwickelt wurde und speziell dafür konzipiert ist, zuverlässige Speicherabbilder von Linux-Systemen zu erstellen. Es bietet eine robuste Alternative zu herkömmlichen Tools und vermeidet viele der häufigen Fehlerquellen bei der Erstellung von RAM-Dumps.

Um ein Speicherabbild zu erstellen, muss AVML lediglich mit den erforderlichen Zugriffsrechten ausgeführt werden. Ein typisches Beispiel für die Ausführung von AVML stellt der folgende Befehl dar.

```
1        sudo ./avml /pfad/zur/Datei
```

Dies ermöglicht es, ein vollständiges Speicherabbild zu erstellen, das für die Analyse mit Volatility verwendet werden kann. Da AVML keine speziellen Kernel-Header oder zusätzliche Kompilierungsschritte benötigt, reduziert sich das Risiko von Kompatibilitätsproblemen erheblich.

Vor der Durchführung des RAM-Dumps wurden gezielt Anwendungen gestartet, darunter PyCharm, eine integrierte Entwicklungsumgebung für Python, sowie der Firefox-Browser. Dies diente dazu, eine repräsentative Nutzungsumgebung zu simulieren und erkennbare Muster in der Memmap-Ausgabe zu generieren. Zusätzlich wurden die Prozess-IDs von PyCharm und Firefox dokumentiert. Ebenfalls wurde im Rahmen der Vorbereitung ein Programm in der Sprache C geschrieben, welches das Muster *0xDEADBEEF* in den Hauptspeicher schreibt und anschließend die Prozess-ID von sich sowie die virtuellen Speicheradresse des Musters ausgibt (vgl. A.8: create_ram_pattern.c). Diese Informationen sind essentiell, um später die Memmap-Ausgabe mit den erwarteten Daten abzugleichen. Abschließend wurde ein Speicherabbild des Systems erzeugt, das als Grundlage für die Entwicklung des Memmap-Plugins dienen soll.

3.1.2. Installation von Volatility 3

Für die Entwicklung des Plugins ist zunächst die Software *volatility3* erforderlich. Diese kann aus dem Github-Repository von *Volatility Foundation* auf das lokale System geklont werden (siehe Volatility Foundation [2024]). Die Installation erfolgt in mehreren Schritten. Zunächst wird das Repository lokale System geklont. Anschließend wechselt man in das neu erstellte Verzeichnis und installiert alle erforderlichen Abhängigkeiten. Die hierfür notwendigen Terminal-Befehle können aus dem Anhang entnommen werden (vgl. A.1: Volatility3 Installation).

Anschließend ist es erforderlich eine Symboltabelle für den Kernel *6.8.11-amd64* zu erstellen. Eine Symboltabelle ist eine spezifische Datei, die die Beziehung zwischen Symbolnamen und Speicheradressen im Kernel eines Betriebssystems definiert. Sie enthält normalerweise Informationen wie Funktionen, Variablen und andere Symbole, die im Kernelcode verwendet werden. Diese Informationen sind entscheidend für die korrekte Analyse von Kernelstrukturen und -daten während der forensischen Analyse.

Für die Erstellung der Symboltabelle ist die Software *dwarf2json* erforderlich. *dwarf2json* ist ein Werkzeug, das verwendet wird, um Debugging-Informationen im DWARF-Format aus ausführbaren Programmen oder Bibliotheken zu extrahieren und sie in das JSON-Format zu konvertieren. Dies ermöglicht eine strukturierte und maschinenlesbare Darstellung der Debugging-Informationen, die für die Analyse und das Debugging von Softwareanwendungen verwendet werden können. Dieses in Go entwickelte Programm extrahiert Debug-Informationen aus Kernel-Dateien und erzeugt daraus eine Intermediate Symbol File (ISF) im JSON-Format, die für die Speicheranalyse von Linux- und macOS-Systemen benötigt wird.Die Erstellung einer solchen Symboltabelle erfordert zwei wesentliche Komponenten: die *vmlinux*-Datei und optional die *System.map*-Datei. Während vmlinux alle nötigen Debug-Informationen enthält, kann die System.map-Datei zusätzliche Genauigkeit bieten.

Es ist jedoch zu beachten, dass im /boot-Verzeichnis eines typischen Linux-Systems standardmäßig nur die vmlinuz-Datei vorhanden ist. Diese Datei ist eine komprimierte Version der vmlinux-Datei, bei deren Kompression wesentliche Debug-Informationen entfernt wurden. Daher kann die vmlinuz-Datei nicht für die Analyse verwendet werden, da sie nicht die benötigten symbolischen Debug-Informationen enthält. Aus diesem Grund muss die originale vmlinux-Datei, die die vollständigen Debug-Symbole enthält, aus den Debug-Symbolpaketen des entsprechenden Kernel-Repositories bezogen werden (vgl. CPUU [2023]).

Anhand des folgenden Befehls kann die *vmlinux*- und die *System.map*-Datei zu dem Kernel *6.8.11-amd64* heruntergeladen werden.

```
1        sudo apt install linux-image-amd64 linux-image-amd64-dbg
```

Abschließend muss basierend auf den heruntergeladenen Dateien die Symboltabellen-Datei generiert und korrekt positioniert werden. Dieser Prozess erfolgt durch die Ausführung des folgenden Befehls:

```
1        sudo ./dwarf2json linux --elf /usr/lib/debug/boot/vmlinux-6.8.11-amd64
             --system-map /boot/System.map-6.8.11-amd64 > symbol_table_kali.json
```

Hierbei wird die ISF-Datei *symbol_ table_ kali.json* mithilfe der Dateien *vmlinux-6.8.11-amd64* und *System.map-6.8.11-amd64* erzeugt. Im Anschluss wird die generierte Symboltabellen-Datei in das *symbols*-Verzeichnis des Volatility3-Ordners verschoben:

```
mv symbol_table_kali.json ../volatility3/volatility3/symbols
```

Diese Schritte stellen sicher, dass die für die Arbeit mit Volatility3 benötigte Symboltabelle korrekt generiert und an der richtigen Stelle platziert wird. Die gesamte Befehlskette kann im Anhang eingesehen werden (vgl. A.12).

3.1.3. Installation von Volatility 2

Da *volatility2* bereits über eine Implementierung des *memmap*-Plugins sowohl für Windows als auch für Linux verfügt, ist es empfehlenswert, auch *Volatility 2* zu installieren. Dies ermöglicht eine Verifikation der Ausgabe des neu entwickelten *memmap*-Plugins für *Volatility 3* durch einen direkten Vergleich der Ergebnisse.

Indem *Volatility2* für Python2 entwickelt wurde, ist es im ersten Schritt notwendig Python2 und den Python-Paketmanager PIP zu installieren.

Im zweiten Schritt erfolgt die Installation von Systemabhängigkeiten. Dazu gehört die Installation von *libdistorm3-dev*, welches eine Entwicklungsbibliothek für Distorm3 darstellt. Distorm3 ist dabei ein x86/AMD64-Disassembler und wird in *volatility2* dazu verwendet, um Maschinencode aus Speicherabzügen zu disassemblieren und zu analysieren. Ein weiteres wichtiges Werkzeug ist *Yara*, das zur Identifikation und Klassifikation von Malware dient. Yara ermöglicht es, Muster in Dateien und Speicher abzugleichen, um schädliche Software zu erkennen. Dies ist besonders nützlich für forensische Analysen, da Yara spezifische Signaturen verwenden kann, um bekannte Bedrohungen schnell zu identifizieren. Yara wird über die Paketverwaltung von Ubuntu installiert. Zusätzlich wird die Kryptografiebibliothek *PyCrypto* benötigt, die für verschiedene kryptografische Operationen in Volatility 2 eingesetzt wird. PyCrypto bietet eine Vielzahl von kryptografischen Funktionen wie sichere Hashing-Algorithmen, Verschlüsselungsalgorithmen und andere Sicherheitsfunktionen. Die Bibliothek wird durch das Herunterladen des entsprechenden Archivs, das Entpacken und die anschließende Installation bereitgestellt.

Die hierfür notwendigen Terminal-Befehle können aus dem Anhang entnommen werden (vgl. A.2: Volatility2 Installation).

3.1.4. Volatility2 Profile

Die Erstellung eines Volatility-Profils unter Volatility 2 unterscheidet sich grundlegend von der Vorgehensweise unter Volatility 3. Während bei Volatility 2 ein spezifisches Profil für das zugrunde liegende System des Speicherabzugs definiert werden muss, generiert Volatility 3 ISF-Dateien zur Analyse.

Um ein präzises Volatility-Profil für die forensische Analyse unter Linux mit Volatility 2 zu erstellen, sind spezifische Schritte erforderlich, um relevante Kernelinformationen zu erfassen. Dieser Prozess beginnt mit der Navigation zum Verzeichnis volatility/tools/linux/, das die erforderlichen Tools zur Profilerstellung bereitstellt.

Zunächst müssen Systemabhängigkeiten wie *dwarfdump*, build-essential und die aktuellen Linux-Header installiert werden, um die Entwicklungsumgebung vorzubereiten. *dwarfdump* ist ein Werkzeug, das Debugging-Informationen aus ELF-Dateien extrahiert. ELF (Executable and Linkable Format) ist ein Dateiformat, das hauptsächlich für ausführbare Programme, Bibliotheken

und Core-Dumps auf Unix-Systemen verwendet wird. Diese Informationen umfassen Details über Variablen, Funktionen und ihre Strukturen im Binärcode. Sie sind entscheidend, um detaillierte Einblicke in den Kernel zu gewinnen und ermöglichen eine präzise Analyse von Speicherabbildern. Die Linux-Header sind eine wichtige Komponente, die für die Kompilierung von Kernel-Modulen unerlässlich sind. Sie enthalten Header-Dateien und andere notwendige Dateien, die für die Entwicklung von Kernel-Modulen erforderlich sind. Diese werden mit dem folgenden Befehl installiert, wobei *$(uname -r)* die aktuelle Kernel-Version darstellt.

```
1         sudo apt install linux-headers-$(uname -r)
```

Damit das Kernel-Modul kompiliert werden kann, ist es erforderlich die Datei "modules.c" um die folgende Anweisung zu erweitern.

```
1         MODULE\_LICENSE("GPL");
```

Ohne diese Lizenzangabe kann das Modul nicht kompiliert werden, da es eine Voraussetzung für die GPL-Kompatibilität darstellt.

Nachdem das Kernel-Modul erfolgreich kompiliert wurde, wird das Profil erstellt, indem die Debugging-Informationen des Kernel-Moduls ("module.dwarf") und die System.map-Datei des aktuellen Kernels in einem ZIP-Archiv zusammengefasst werden. Diese Dateien sind entscheidend für die Analyse von Speicherabbildern und ermöglichen eine präzise Rekonstruktion der Systemaktivitäten während der forensischen Untersuchung.

Eine detaillierte Anleitung und technische Informationen zu diesem Prozess finden sich im Anhang des Berichts, um eine genaue Reproduktion der Ergebnisse zu gewährleisten (vgl. A.5: Vol2 - Linux Profil erstellen).

3.1.5. Herausforderungen bei der Konfiguration von Volatility 2

Mit dem Abschluss der Installation ist Volatility 2 vollständig eingerichtet und bereits funktionsfähig. Jedoch treten noch Fehlfunktionen im Kontext des Plugins *linux_ memmap* in Kombination mit verschiedenen Linux-Speicherabzügen auf.

Zunächst tritt ein Fehler auf, der durch eine fehlende Zuordnung für die Datentypen "__int128 unsigned" und "__int128" verursacht wird. Dieser Datentyp ist in der Debugging-Information (DWARF) des Speicherabbilds nicht korrekt aufgelöst, was zu einem KeyError führt.

Um diesen Fehler zu beheben, müssen die Datentypen "__int128 unsigned" und "__int128" im DWARF-Parser von Volatility hinzugefügt werden. Dies kann durch die Aktualisierung der Typzuordnung (tp2vol) in der Datei volatility/dwarf.py erfolgen. Konkret sollte dem Datentyp "__int128 unsigned" der Wert "unsigned long long" und dem Datentypen "__int128" der Wert "long long" in der tp2vol-Liste zugeordnet werden, um das Problem zu lösen (vgl. A.6: Vol2 - volatility/dwarf.py).

Im Zuge einer Änderung im Linux-Kernel ab Version 5.14-rc1 wurde das Feld für den Aufgabenstatus nicht mehr als ßtate", sondern als "__state"bezeichnet. Dies führte zu einem Fehler bei der Verarbeitung von Speicherabbildern mit neueren Kernel-Versionen, speziell durch den Fehler, dass ein inkompatibles Profil für die Analyse des Speicherabbilds verwendet wird (vgl. A.8). Um dieses Problem zu lösen, wurde die Datei volatility/plugins/overlays/linux/linux.py angepasst, um zunächst nach dem Feld "state" zu suchen. Falls dieses nicht gefunden wird, wird

nun nach "__state"gesucht. Diese Anpassung ermöglicht es Volatility, korrekt mit den neuen Kernelversionen umzugehen und die Analyse von Speicherabbildern fehlerfrei durchzuführen (vgl. Volatility Foundation [2023b]).

Der abschließende Fehler resultiert aus der Verwendung von DWARF Debug-Informationen in der Version 5, die Volatility 2 nicht unterstützt. Konkret äußert sich der Fehler durch den KeyError bei dem diverse Schlüsselwert nicht ausgelesen werden können. Damit dieser Fehler gelöst und die Plugins korrekt ausgeführt werden können, ist es notwendig die Datei volatility/dwarf.py zu modifizieren. Diese Modifikationen an der Datei belaufen sich auf eine Reihe von Änderungen, die hauptsächlich darauf abzielen, die Handhabung von DWARF Debug-Informationen zu verbessern. Zuvor wurden bestimmte Datenelemente wie 'DW_AT_data_member_location' und 'DW_AT_data_bit_offset' auf spezifische Weisen verarbeitet, die zu Problemen bei der Analyse von Speicherabbildern führten, insbesondere wenn diese Felder nicht vorhanden waren. Die Aktualisierung fügt eine robustere Fehlerbehandlung hinzu, um sicherzustellen, dass auch in solchen Fällen eine korrekte Verarbeitung gewährleistet ist (vgl. Volatility Foundation [2023a]).

3.2. Architektur von Volatility3 Plugins

Die Grundstruktur eines Volatility 3 Plugins besteht aus mehreren wichtigen Komponenten und Konzepten, die zusammenarbeiten, um die Funktionalität des Plugins bereitzustellen.

Jedes Plugin wird als eine Klasse implementiert, die von der Basisklasse *PluginInterface* erbt, welche im *interfaces.plugins* Modul definiert ist. Diese Vererbung stellt sicher, dass das Plugin über die notwendigen Methoden und Eigenschaften verfügt, die das Framework erwartet. Die Plugin-Klasse enthält die Hauptlogik des Plugins und implementiert spezifische Methoden wie run() und get_requirements().

Zusätzlich zu den Methoden definiert jedes Plugin auch seine eigene Version sowie die minimal benötigte Version des Frameworks. Diese werden durch die Klasseneigenschaften _version und _required_framework_version festgelegt, um Kompatibilität und Funktionsfähigkeit sicherzustellen.

```
1         _version = (1, 0, 0)
2         _required_framework_version = (2, 0, 0)
```

3.2.1. Methode: get_requirements()

Ein zentrales Merkmal der Plugin-Architektur in Volatility 3 ist die Definition von Anforderungen, die sicherstellen, dass alle notwendigen Parameter und Ressourcen vor der Ausführung des Plugins bereitgestellt werden. Diese Anforderungen werden durch die Methode get_requirements() spezifiziert, welche eine Liste von Anforderungsobjekten zurückgibt. Diese Anforderungen umfassen verschiedene Typen, darunter ModuleRequirement, ListRequirement, PluginRequirement, TranslationLayerRequirement und SymbolTableRequirement.

1. **ModuleRequirement**

   Die Anforderung *ModuleRequirement* spezifiziert das notwendige Submodul, wie z.B. den Kernel. Ein ModuleRequirement erfordert normalerweise eine TranslationLayer und eine SymbolTable, die durch die Automagic-Mechanismen des Frameworks bereitgestellt werden. Automagic-Module ermöglichen es dem Framework, Konfigurationselemente, die vom

Benutzer nicht bereitgestellt wurden, automatisch zu füllen. Dadurch wird sichergestellt, dass alle notwendigen Ressourcen und Konfigurationsparameter ohne Benutzerintervention geladen werden. Eine mögliche Definition eines ModuleRequirements könnte folgendermaßen aussehen.

```
        requirements.ModuleRequirement(
                name='kernel',
                description='Windows kernel',
                architectures=["Intel32", "Intel64"]
        )
```

Diese Definition spezifiziert, dass das Plugin einen Kernel benötigt, der auf den Architekturen "Intel32" und "Intel64" basiert. Der Name des Moduls wird als "kernel" gespeichert, und das Modulobjekt selbst kann aus der Kontext-Sammlung context.modules abgerufen werden. Das ModuleRequirement sorgt somit dafür, dass alle notwendigen Layer und Symbole für den Kernel geladen sind, bevor das Plugin ausgeführt wird.

2. **ListRequirement**

   Diese Anforderung wird verwendet, um Listen von Werten zu spezifizieren, die vom Benutzer bereitgestellt werden können, aber nicht zwingend erforderlich sind. Ein häufiges Beispiel ist die Angabe von Prozess-IDs (PIDs), die das Plugin analysieren soll.

```
        requirements.ListRequirement(
                name='pid',
                description='Process IDs to include',
                element_type=int,
                optional=True
        )
```

   Diese Definition spezifiziert, dass das Plugin eine Liste von Prozess-IDs erwartet, die vom Typ Integer sind. Die Beschreibung Process IDs to include (all other processes are excluded)"wird dem Benutzer angezeigt, und das Attribut öptional=True"gibt an, dass das Plugin auch ohne diese Angabe funktionieren kann.

3. **PluginRequirement**

   Diese Anforderung gibt an, dass das Plugin auf einem anderen Plugin basiert und spezifiziert die notwendige Version dieses Plugins. Dies stellt sicher, dass alle Abhängigkeiten korrekt aufgelöst sind, bevor das Plugin ausgeführt wird. Die Version wird nach Semantic Versioning angegeben und gewährleistet Kompatibilität innerhalb der angegebenen Versionsgrenzen.

```
        requirements.PluginRequirement(
                name='pslist',
                plugin=pslist.PsList,
                version=(2, 0, 0)
        )
```

   Diese Definition spezifiziert, dass das Plugin die Funktionen des pslistPlugins nutzt, wobei die Version des pslistPlugins mindestens 2.0.0 sein muss. Dies stellt sicher, dass die benötigten Funktionen und Merkmale des anderen Plugins verfügbar sind.

4. **TranslationLayerRequirement**

   Diese Anforderung gibt an, dass das Plugin auf einem bestimmten TranslationLayer operieren muss, der die Architektur des Speichers angibt. Der Name des Layers wird in der Konfiguration des Plugins gespeichert und kann dynamisch angepasst werden, je nach den bereits im Kontext vorhandenen Layern.

```
1        requirements.TranslationLayerRequirement(
2                name='primary',
3                description='Memory layer for the kernel',
4                architectures=["Intel32", "Intel64"]
5        )
```

   Diese Definition spezifiziert, dass das Plugin auf einer Speicher-TranslationLayer mit dem Namen primaryärbeitet, die auf den Architekturen Ïntel32ünd Ïntel64"basiert. Der Name des geladenen Layers wird in der Plugin-Konfiguration unter dem Namen “primary“ erscheinen.

5. **SymbolTableRequirement**

   Diese Anforderung spezifiziert die Notwendigkeit einer bestimmten SymbolTable, die im Zusammenhang mit einem TranslationLayer steht. Die Automagic-Mechanismen des Frameworks füllen diese Anforderung automatisch, indem sie die entsprechende SymbolTable laden und den Namen in der Konfiguration speichern.

```
1        requirements.SymbolTableRequirement(
2                name='nt_symbols',
3                description='Windows kernel symbols'
4        )
```

   Diese Definition spezifiziert, dass das Plugin eine SymbolTable mit dem Namen “nt_symbols“ benötigt, die die Symbole für den Windows-Kernel enthält. Die Automagic-Mechanismen füllen diese Anforderung automatisch, indem sie die entsprechende SymbolTable laden.

Die get_requirements() Methode ist als classmethod implementiert, da sie aufgerufen wird, bevor das spezifische Plugin-Objekt instanziiert wird. Dies ermöglicht es dem Framework, die notwendigen Konfigurationsparameter zu ermitteln und sicherzustellen, dass alle Anforderungen erfüllt sind, bevor das Plugin ausgeführt wird. Die definierten Anforderungen sorgen dafür, dass das Plugin die benötigten Ressourcen und Informationen erhält, um korrekt zu funktionieren und die gewünschte Analyse durchzuführen (vgl. Volatility 3 Documentation).

3.2.2. Methode: run()

Die run-Methode repräsentiert das zentrale Element jedes Volatility 3 Plugins, das bei der Ausführung aufgerufen wird. Diese Methode erfordert keine direkten Parameter, da alle notwendigen Informationen über den Kontext und die Konfiguration des Plugins bereitgestellt werden. Das Rückgabeobjekt dieser Methode ist eine unpopulated TreeGrid-Instanz. Diese Instanz dient als Container für die Ausgabe des Plugins und definiert die Struktur der Daten, einschließlich der Spaltennamen und Datentypen. Während der Ausführung wird die TreeGrid durch einen Generator mit den analysierten Informationen gefüllt, die dann für weitere Analyse oder Anzeige verwendet werden können.

3.2.3. Methode: _generator()

Die _generator()-Funktion in Volatility 3 ist der Bestandteil des Plugin-Prozesses, der die wesentliche Datenverarbeitung für die Erstellung der Ausgabe übernimmt. Sie wird verwendet, um eine strukturierte Darstellung der Informationen zu erzeugen, die durch das Plugin analysiert wurden. Dabei iteriert die Funktion über eine Liste von Prozessen, ruft für jeden Prozess die geladenen Module ab und extrahiert relevante Informationen wie DLL-Namen und Prozess-IDs. Während dieses Prozesses stellt die Funktion sicher, dass unlesbare oder nicht verfügbare Daten angemessen behandelt werden, um die Genauigkeit der Plugin-Ausgabe zu gewährleisten.

3.3. Implementierungsphase

Nach der erfolgreichen Installation und Sicherstellung der Funktionsfähigkeit von Volatility 2 und 3 steht nun der Entwicklungsprozess im Fokus. Im Weiteren wird insbesondere auf die Bestandteile des Memmap-Plugins eingegangen.

3.3.1. LinuxMemmap: get_requirements()

Die Methode get_requirements() dient der systematischen Spezifikation der spezifischen Anforderungen eines Plugins innerhalb des Volatility-Frameworks. Im Kontext des Memmap-Plugins werden die Anforderungen derart spezifiziert, dass die Rückgabe dieser Methode aus einer geordneten Liste von RequirementInterface-Objekten besteht, die die präzisen Anforderungen des Plugins definieren. Diese Objekte stellen sicher, dass das Plugin korrekt initialisiert und ausgeführt werden kann, unter Berücksichtigung spezifischer Abhängigkeiten und Konfigurationen.

Konkret werden in der Funktion, die folgenden Schlüsselanforderungen für das Plugin spezifiziert.

1. **Kernel-Modul-Anforderung:**

   Dies definiert das erforderliche Linux-Kernel-Modul, das für die Analyse der Memory Mappings verwendet werden soll. Diese Anforderung stellt sicher, dass das Plugin kompatibel ist und effektiv mit den zugrunde liegenden Systemressourcen interagieren kann, unabhängig von der Prozessorarchitektur (Intel32 oder Intel64).

2. **PsList Plugin-Anforderung:**

   Durch diese Anforderung wird das PsList-Plugin spezifiziert, das zur Auflistung der Prozesse innerhalb des Ziel-Linux-Systems erforderlich ist. Zur Gewährleistung der Kompatibilität ist die Version auf 2.0.0 spezifiziert.

3. **PID-Anforderung:**

   Optional kann eine spezifische Prozess-ID (PID) angegeben werden, um das Plugin gezielt auf bestimmte Prozesse zur Analyse zu beschränken. Diese Funktion bietet eine Flexibilität ähnlich der Memmap-Funktionalität in Volatility 2 für Linux und Windows oder der Memmap-Funktionalität in Volatility 3 für Windows.

4. **Dump-Option:**

   Diese Boole'sche-Anforderung ermöglicht das Extrahieren von aufgelisteten Memory Segmenten in separate Dateien. Diese Funktion ist besonders wichtig für die forensische Analyse, da sie die Sicherung und spätere Detailuntersuchung spezifischer Speicherinhalte ermöglicht.

5. **Split-Option:**

   Ebenfalls als Boolean-Anforderung implementiert, erlaubt diese Option das Aufteilen von großen Memory Segmenten in kleinere Abschnitte von 4096 Byte. Diese Funktionalität unterstützt detaillierte Untersuchungen von Speicherbereichen und ermöglicht dem Analysten eine Ausgabe ähnlich dem Memmap-Plugin von Volatility2 zu erhalten.

Zusammengefasst ermöglicht die Methode get_requirements() (vgl. A.9: Memmap - get_requirements()) der Klasse LinuxMemmap durch die Definition dieser Anforderungen eine Initialisierung des Volatility-Plugins gemäß folgendem Format.

```
$ python3 vol.py -f /path/to/dump_file linux.memmap --pid=<pid> --split > ./<
    output_file>
```

3.3.2. LinuxMemmap: run()

Die run()-Methode wird aufgerufen, um das Plugin auszuführen und einen TreeGrid zurückzugeben, der die strukturierten Details der Memory Maps enthält. Der folgende Code demonstriert die Implementierung dieser Funktion:

```
        def run(self):
                filter_func = linux_pslist.PsList.create_pid_filter([self.config
                    .get("pid", None)])

                return renderers.TreeGrid(
                        [
                                ("PID", int),
                                ("Virtual", format_hints.Hex),
                                ("Physical", format_hints.Hex),
                                ("Size", format_hints.Hex),
                                ("Offset in File", format_hints.Hex),
                                ("File output", str),
                        ],
                        self._generator(
                                linux_pslist.PsList.list_tasks(
                                context=self.context,
                                vmlinux_module_name=self.config["kernel"],
                                filter_func=filter_func,
                                include_threads=False,
                                )
                        ),
                )
```

Im ersten Schritt der Methode wird eine Filterfunktion textitfilter_func definiert, die durch die Methode create_pid_filter() des PsList-Plugins von Volatility erstellt wird. Diese Filterfunktion wird verwendet, um die Liste der Prozesse basierend auf einer spezifischen PID zu filtern, die

optional als Konfigurationsparameter angegeben werden kann. Dadurch kann das Plugin gezielt nur diejenigen Prozesse analysieren, die für die aktuelle Untersuchung von Interesse sind.

Die Hauptfunktionalität der run()-Methode besteht darin, einen TreeGrid zu erstellen, der die formatierten Informationen der Memory Mappings enthält. Dies geschieht durch den Aufruf von renderers.TreeGrid() mit folgenden Parametern:

- **Kernel-Modul-Anforderung:** Die Spalten des TreeGrids werden definiert, um die wesentlichen Details der Memory Mappings darzustellen. Diese umfassen unter anderem die PID des Prozesses, die virtuelle und physische Speicheradresse, die Größe des Memory Segments, den Offset in der Datei (falls zutreffend) und den Dateiausgabe-Status.
- **Datenquelle:** Die Datenquelle für den TreeGrid wird durch self._generator() bereitgestellt. Diese Methode wird aufgerufen, um die Memory Mapping-Informationen für die ausgewählten Prozesse zu generieren. Sie nutzt die PsList.list_tasks()-Funktion, um eine Liste der Prozesse zu erhalten, die den definierten Filterkriterien entsprechen. Dabei werden auch Threads ausgeschlossen, um nur Hauptprozesse zu berücksichtigen.

3.3.3. LinuxMemmap: _generator()

Durch die _generator()-Methode werden sequenziell Memory Mapping-Details für jeden Prozess eines gegebenen Listenparameters *procs* bereitgestellt.

Die Methode beginnt mit einer Iteration über die bereigtestellte Liste *procs*, die eine Ansammlung von Prozessobjekten darstellt, die für die Analyse vorgesehen sind. Für jeden Prozess werden zunächst grundlegende Informationen wie die Prozess-ID abgerufen, wobei ein Standardwert "Unknown" verwendet wird, falls die ID nicht verfügbar ist.

Anschließend wird versucht, eine spezifische Schicht (proc_layer) für den Prozess zu erhalten, die im Volatility-Framework als notwendig erachtet wird, um auf den Speicher des Prozesses zugreifen zu können. Dabei können Exceptions auftreten, wie zum Beispiel eine InvalidAddressException, die behandelt wird, indem entsprechende Debugmeldungen generiert werden, um die fehlerhaften Adressen oder nicht gefundenen Schichten zu protokollieren.

Für den Fall, dass das Plugin konfiguriert ist, um Memory Segmente auf Dateiebene auszugeben (dump=True), wird ein Dateihandle (file_handle) geöffnet, um die Ergebnisse zu speichern. Andernfalls wird ein ExitStack verwendet, um sicherzustellen, dass Ressourcen ordnungsgemäß verwaltet werden, selbst wenn kein Dumpvorgang stattfindet.

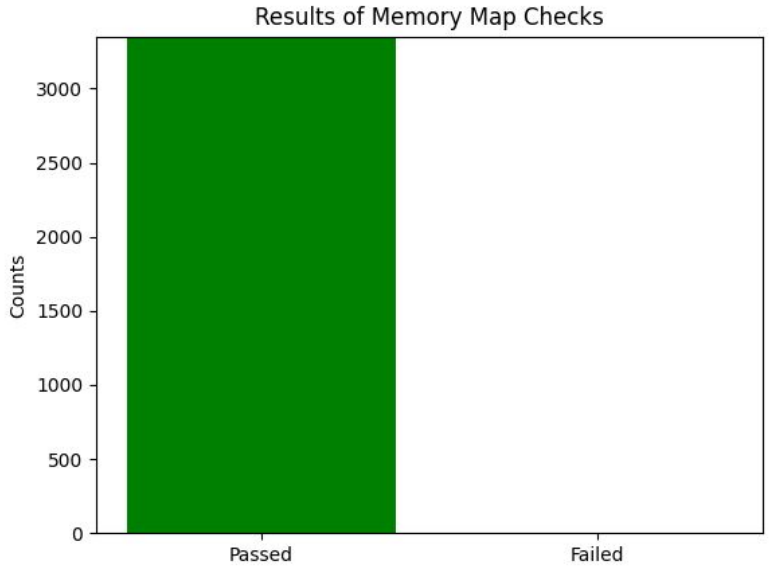
Die Hauptaufgabe der _generator-Funktion besteht darin, die Memory Mapping-Details für jeden Prozess zu extrahieren und als Tupel zu erhalten. Dies geschieht durch Iteration über die mapping()-Methode der proc_layer, die die Speicherabbildungsinformationen des Prozesses liefert. Wenn die Konfiguration das Aufteilen von Memory Segments erfordert (split=True), werden diese Segmente in kleinere Abschnitte von 4096 Byte unterteilt. Andernfalls wird das gesamte Memory Segment auf einmal ausgelesen und im TreeGrid platziert (vgl. A.11: Memmap - _generator()).

4. Testphase

Für die Verifikation der Korrektheit der Ausgabe des Memmap-Plugins wurde das Python-Skript *compare_vol2_vol3_output.py* implementiert (vgl. A.13: Comparsion-Script).

Das entwickelte Vergleichsskript nutzt eine Kombination aus Python-Funktionalitäten und matplotlib zur Validierung der Output-Konsistenz zwischen den Volatility-Versionen 2 und 3 für das memmap-Plugin. Aufgrund der Unterschiede im Output-Format zwischen den beiden Versionen sind spezielle Hilfsfunktionen erforderlich. Diese Funktionen ermöglichen die Konvertierung zwischen Hexadezimal- und Ganzzahlenformaten sowie das Parsen von Output- und Referenzdateien. Insbesondere sind die Funktionen *hex_to_int()* und *int_to_hex()* entscheidend, um die unterschiedlichen Darstellungen von Virtual- und Physical-Adressen korrekt zu handhaben. Die *parse_output_file-Funktion* extrahiert Informationen aus der Output-Datei von Volatility, während *parse_reference_file()* die Referenzdaten aus einer separaten Datei liest, die von einer anderen Quelle, wie etwa Volatility 2, erstellt wurde. Die *check_memory_maps-Funktion* führt dann den eigentlichen Vergleich aus, indem sie die Virtual- und Physical-Adressen der Output-Daten mit den Referenzdaten abgleicht und statistische Ergebnisse über Erfolgs- und Fehlerquoten ausgibt. Zusätzlich visualisiert die *plot_results-Funktion* die Ergebnisse in einem Balkendiagramm, das die Anzahl der bestandenen und nicht bestandenen Tests darstellt, um eine schnelle visuelle Interpretation der Testergebnisse zu ermöglichen.

Für die Überprüfung der Konsistenz der Memory Mappings wurden verschiedene Ansätze verfolgt. Zunächst wurden Speicherabzüge von verschiedenen Ubuntu-Versionen mit unterschiedlichen Kernel-Versionen durchgeführt: Ubuntu 20.04 LTS mit Kernelversion 5.15.0-113-generic, Ubuntu 22.04 LTS mit Kernelversion 6.5.0-41-generic und Ubuntu 24.04 LTS mit Kernelversion 6.8.0-35-generic, sowie ein Speicherabzug von Kali Linux. Die Ausgaben der Volatility-Versionen 2 und 3 wurden initial ohne Beschränkung auf bestimmte Prozess-IDs verglichen. Anschließend wurden gezielt die Memory Mappings für ausgewählte Prozess-IDs, einschließlich derjenigen von Pycharm, Firefox und des "create_ram_pattern.c", analysiert und die Ergebnisse der verschiedenen Kernelversionen verglichen.



Ein spezifisches Ergebnis dieser Analyse betrifft den Prozess des “create_ram_pattern.c“ im Speicherabzug von Ubuntu 20.04 LTS. Die grafische Darstellung in Abbildung 1 zeigt, dass von insgesamt 3342 virtuellen und physischen Adressen, die aus der Volatility3-Ausgabe extrahiert wurden, exakt 3342 übereinstimmende Adressen in der Volatility2-Ausgabe gefunden werden konnten. Diese Übereinstimmung wurde analog auch für die anderen untersuchten Speicherabzüge und Vergleichsmethoden festgestellt.

Diese Ergebnisse belegen die konsistente Funktionalität und Genauigkeit des entwickelten Memmap-Plugins sowie die Zuverlässigkeit bei den Memory Mappings zwischen den untersuchten Volatility-Versionen und unterschiedlichen Betriebssystemkonfigurationen.

5. Schlussfolgerung

Durch das Upgrade auf Volatility 3 wurde die Bedienung des Frameworks sowie die Entwicklung von Plugins erheblich optimiert. Volatility 2 arbeitete mit Profilen, die aus den Linux-Headern und der System.map-Datei generiert wurden. Diese Profile enthalten Informationen zu Strukturdefinitionen und Speicher-Offsets, die für die Analyse des Speichers spezifischer Betriebssysteme erforderlich sind.Aufgrund der Vielzahl an bestehenden Plugins in Volatility 2, die in Volatility 3 noch nicht verfügbar sind, werden weiterhin Community-Updates für Volatility 2 durchgeführt. Diese Updates sind von Vorteil, da sie die Funktionalität des Frameworks erweitern, indem sie die Kompatibilität zu aktuellen Kernelversionen sicherstellen.

Ein signifikanter Nachteil von Volatility 2 ist jedoch die Abhängigkeit von Python 2, das im Jahr 2020 das Ende seiner Unterstützung (End of Life) erreichte. Dies hat zur Folge, dass Volatility 2 nicht mehr von den neuesten Updates profitiert, was potenzielle Sicherheitslücken und Kompatibilitätsprobleme aufwerfen kann. Darüber hinaus ist Volatility 2 im Vergleich zu Volatility 3 langsamer in der Ausführung von Analysen und erfordert häufig längere Verarbeitungszeiten für komplexe Speicherabbilder.

Im Gegensatz dazu verwendet Volatility 3 Symboltabellen, die eine präzisere und umfassendere Analyse ermöglichen. Diese Symboltabellen werden nicht nur auf Basis der System.map-Datei erstellt, sondern beziehen auch die vmlinux-Datei mit ein, welche die Debug-Symbole des Kernels enthält.

Die Integration der vmlinux-Datei ermöglicht eine genauere und detailliertere Darstellung der Kernel-Datenstrukturen und -Funktionen. Diese Änderung verbessert insbesondere die Benutzerfreundlichkeit des Frameworks, da die Generierung von Symboltabellen im Vergleich zu Profilen wesentlich effizienter und weniger fehleranfällig ist. In Volatility 2 war die Erstellung von Profilen ein komplexer und oft fehleranfälliger Prozess, der eine präzise Abstimmung der Linux-Header und der System.map-Datei erforderte.

Volatility 3 hingegen ermöglicht durch die Nutzung von Symboltabellen eine automatisiertere und robustere Methode zur Speicheranalyse. Die Integration der vmlinux-Datei, die Debug-Symbole enthält, trägt dazu bei, dass die Symboltabellen genauer und vollständiger sind, was die Entwicklung und Anwendung von Plugins vereinfacht und die Genauigkeit der Analyse verbessert. Ein wesentlicher Nachteil von Volatility 3 besteht darin, dass die Generierung von Symboltabellen für einige Linux-Kernel problematisch sein kann, da die erforderlichen Debug-Symbole (vmlinux-Dateien) nicht immer verfügbar sind. Verschiedene Linux-Distributionen bieten ihre Debugging-Pakete unter unterschiedlichen Namen an und archivieren möglicherweise nicht alle alten Versionen der Debug-Symbole. Dies kann dazu führen, dass die notwendigen Symbole für die Analyse eines bestimmten Linux-Speicherabbilds nicht auffindbar sind.

Literatur

CPUU. How to perform memory forensic analysis in linux using volatility 3, 2023. URL https://cpuu.hashnode.dev/how-to-perform-memory-forensic-analysis-in-linux-using-volatility-3. Accessed: 2024-06-18.

Volatility 3 Documentation. How to write a simple plugin. https://volatility3.readthedocs.io/en/latest/simple-plugin.html. Zugriff: 01. Juli 2024.

Volatility Foundation. Pull request 854: Enhancements to dwarf parsing in dwarf.py, 2023a. URL https://github.com/volatilityfoundation/volatility/pull/854. Zugriff: 01. Juli 2024.

Volatility Foundation. Update linux dtb scanner to handle newer linux kernel versions. GitHub, 2023b. https://github.com/volatilityfoundation/volatility/pull/852.

Volatility Foundation. Volatility 3 repository. https://github.com/volatilityfoundation/volatility3, 2024. Zugriff: 17. Juni 2024.

Volatility Project. Migration von volatility 2 zu volatility 3, 2024. URL https://volatility3.readthedocs.io/en/latest/vol2to3.html. Zugriff: 17. Juni 2024.

A. Anhang

A.1. Volatility3 Installation

```
git clone https://github.com/volatilityfoundation/volatility3.git
cd volatility3
pip install -r requirements.txt
```

A.2. Volatility2 Installation

```
# Python, PIP Installation
$ sudo apt install python2
$ sudo apt-get install python-dev
$ sudo apt install curl
$ curl https://bootstrap.pypa.io/pip/2.7/get-pip.py --output get-pip.py
$ sudo python2 get-pip.py

# Volatility Installation
$ git clone https://github.com/volatilityfoundation/volatility.git
$ cd volatility
$ sudo python2 setup.py install

# Distorm Installation
$ git clone https://github.com/gdabah/distorm.git
$ cd distorm
$ python setup.py build
$ sudo python setup.py build install

$ sudo apt-get install yara -y
$ wget https://ftp.dlitz.net/pub/dlitz/crypto/pycrypto/pycrypto-2.6.1.
    tar.gz
$ tar -xvzf pycrypto-2.6.1.tar.gz
$ cd pycrypto-2.6.1
$ python setup.py build
$ sudo python setup.py build install

https://github.com/volatilityfoundation/volatility/issues/837
```

A.3. AVML Installation

```
$ wget https://github.com/microsoft/avml/releases/download/v0.14.0/avml
$ chmod +x avml
$ sudo ./avml memory.lime
```

A.4. Symboltabelle

```
git clone https://github.com/volatilityfoundation/dwarf2json.git
cd dwarf2json/
go build
sudo apt install linux-image-amd64 linux-image-amd64-dbg
```

```
sudo ./dwarf2json linux --elf /usr/lib/debug/boot/vmlinux-6.8.11-amd64
    --system-map /boot/System.map-6.8.11-amd64 > symbol_table_kali.json
mv symbol_table_kali.json ../volatility3/volatility3/symbols
```

A.5. Vol2 - Linux Profil erstellen

```
#path: volatility/tools/linux/

$ sudo apt install dwarfdump
$ sudo apt install build-essential
$ sudo apt install linux-headers-$(uname -r)
$ vim modules.c
# MODULE_LICENSE("GPL"); # append to modules.c

$ make


$ zip $(lsb_release -i -s)_$(uname -r)_profile.zip ./module.dwarf /boot/
    System.map-$(uname -r)
```

A.6. Vol2 - volatility/dwarf.py

```
#path: volatility/dwarf.py
tp2vol = {
        ...
        '__int128': 'long long',
        '__int128 unsigned': 'unsigned long long',
}
```

A.7. Vol2 - Dwarfv5 Error

```
No suitable address space mapping found
Tried to open image as:
MachOAddressSpace: mac: need base
LimeAddressSpace: lime: need base
WindowsHiberFileSpace32: No base Address Space
WindowsCrashDumpSpace64BitMap: No base Address Space
VMWareMetaAddressSpace: No base Address Space
WindowsCrashDumpSpace64: No base Address Space
HPAKAddressSpace: No base Address Space
VirtualBoxCoreDumpElf64: No base Address Space
QemuCoreDumpElf: No base Address Space
VMWareAddressSpace: No base Address Space
WindowsCrashDumpSpace32: No base Address Space
SkipDuplicatesAMD64PagedMemory: No base Address Space
WindowsAMD64PagedMemory: No base Address Space
LinuxAMD64PagedMemory: No base Address Space
AMD64PagedMemory: No base Address Space
IA32PagedMemoryPae: No base Address Space
IA32PagedMemory: No base Address Space
OSXPmemELF: No base Address Space
MachOAddressSpace: MachO Header signature invalid
```

```
22          MachOAddressSpace: MachO Header signature invalid
23          LimeAddressSpace: Invalid Lime header signature
24          WindowsHiberFileSpace32: PO_MEMORY_IMAGE is not available in profile
25          WindowsCrashDumpSpace64BitMap: Header signature invalid
26          VMWareMetaAddressSpace: VMware metadata file is not available
27          WindowsCrashDumpSpace64: Header signature invalid
28          HPAKAddressSpace: Invalid magic found
29          VirtualBoxCoreDumpElf64: ELF Header signature invalid
30          QemuCoreDumpElf: ELF Header signature invalid
31          VMWareAddressSpace: Invalid VMware signature: -
32          WindowsCrashDumpSpace32: Header signature invalid
33          SkipDuplicatesAMD64PagedMemory: Incompatible profile LinuxUbuntu_20_04_5
                _15x64 selected
34          WindowsAMD64PagedMemory: Incompatible profile LinuxUbuntu_20_04_5_15x64
                selected
35          LinuxAMD64PagedMemory - EXCEPTION: 'state'
36          AMD64PagedMemory - EXCEPTION: 'state'
37          IA32PagedMemoryPae: Incompatible profile LinuxUbuntu_20_04_5_15x64
                selected
38          IA32PagedMemory: Incompatible profile LinuxUbuntu_20_04_5_15x64 selected
39          OSXPmemELF: ELF Header signature invalid
40          FileAddressSpace: Must be first Address Space
41          ArmAddressSpace - EXCEPTION: 'state'
```

A.8. create_ram_pattern.c

```
1           #include <stdio.h>
2           #include <stdlib.h>
3           #include <unistd.h>
4           #include <sys/mman.h>
5           #include <fcntl.h>
6           #include <stdint.h>
7
8           // Funktion zum Anzeigen von Fehlern und Beenden des Programms
9           void error(const char *msg) {
10                  perror(msg);
11                  exit(1);
12          }
13
14          int main() {
15                  // Groesse des zu schreibenden Musters
16                  size_t pattern_size = 4096; // 4KB Seite
17                  uint32_t *pattern;
18
19                  // Allokieren von speicher mit mmap
20                  pattern = mmap(NULL, pattern_size, PROT_READ | PROT_WRITE,
                        MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
21                  if (pattern == MAP_FAILED) {
22                          error("mmap failed");
23                  }
24
25                  // Schreiben eines Musters in den Speicher
26                  *pattern = 0xDEADBEEF;
27
28                  // Ausgabe der virtuellen Adresse des Musters
29                  printf("Virtuelle Adresse des Musters: %p\n", (void *)pattern);
30
```

```
31                  // Physische Adresse kann ueber /proc/PID/pagemap herausgefunden
                        werden
32                  // Beispielhaft hier dargestellt, aber nicht direkt auslesbar in
                        einem einfachen C-Programm
33
34                  // PID des aktuellen Prozesses
35                  pid_t pid = getpid();
36                  printf("PID: %d\n", pid);
37
38                  // Ausgabe der pagemap-Adresse
39                  char pagemap_file[64];
40                  snprintf(pagemap_file, sizeof(pagemap_file), "/proc/%d/pagemap",
                        pid);
41                  printf("Pagemap-Datei: %s\n", pagemap_file);
42
43                  // Hinweis zum Beenden des Programms
44                  printf("Druecken Sie Enter zum Beenden des Programms...\n");
45                  getchar();
46
47                  // Speicher freigeben
48                  if (munmap(pattern, pattern_size) == -1) {
49                          error("munmap failed");
50                  }
51
52                  return 0;
53          }
```

A.9. Memmap - get_requirements()

```
1           @classmethod
2           def get_requirements(cls) -> List[interfaces.configuration.
                RequirementInterface]:
3
4                   return [
5                           requirements.ModuleRequirement(
6                                   name="kernel",
7                                   description="Linux kernel",
8                                   architectures=["Intel32", "Intel64"]
9                           ),
10                          requirements.PluginRequirement(
11                                  name="pslist", plugin=linux_pslist.PsList,
                                        version=(2, 0, 0)
12                          ),
13                          requirements.IntRequirement(
14                                  name="pid",
15                                  description="Process ID to include (all other
                                        processes are excluded)",
16                                  optional=True
17                          ),
18                          requirements.BooleanRequirement(
19                                  name="dump",
20                                  description="Extract listed memory segments",
21                                  default=False,
22                                  optional=True
23                          ),
24                          requirements.BooleanRequirement(
25                                  name="split",
```

```
26                                  description="Split memory segments into 4096-
                                        byte sections",
27                                  default=False,
28                                  optional=True
29                          ),
30                  ]
```

A.10. Memmap - get_requirements()

```
1           def run(self):
2           filter_func = linux_pslist.PsList.create_pid_filter([self.config.get("
                pid", None)])
3           kernel = self.context.modules[self.config["kernel"]]
4
5           return renderers.TreeGrid(
6                   [
7                           ("PID", int),
8                           ("Virtual", format_hints.Hex),
9                           ("Physical", format_hints.Hex),
10                          ("Size", format_hints.Hex),
11                          ("Offset in File", format_hints.Hex),
12                          ("File output", str),
13                  ],
14                  self._generator(
15                          linux_pslist.PsList.list_tasks(
16                                  context=self.context,
17                                  vmlinux_module_name=self.config["kernel"],
18                                  filter_func=filter_func,
19                                  include_threads=False,
20                          )
21                  ),
22          )
```

A.11. Memmap - _generator()

```
1           def _generator(self, procs):
2
3           for proc in procs:
4           pid = "Unknown"
5
6           try:
7           pid = proc.pid
8           proc_layer_name = proc.add_process_layer()
9           proc_layer = self.context.layers[proc_layer_name]
10          except exceptions.InvalidAddressException as excp:
11          vollog.debug(
12          "Process {}: invalid address {} in layer {}".format(
13          pid, excp.invalid_address, excp.layer_name
14          )
15          )
16          continue
17          except KeyError:
18          vollog.error(f"Process {pid}: layer {proc_layer_name} not found in
                context.layers")
19          continue
```

```
20
21          if self.config["dump"]:
22          file_handle = self.open(f"pid.{pid}.dmp")
23          else:
24          file_handle = contextlib.ExitStack()
25
26          with file_handle as file_data:
27          file_offset = 0
28          for mapval in proc_layer.mapping(
29          0x0, proc_layer.maximum_address, ignore_errors=True
30          ):
31          offset, size, mapped_offset, mapped_size, maplayer = mapval
32
33          if self.config["split"]:
34          section_size = 4096
35          while size > 0:
36          chunk_size = min(section_size, size)
37          file_output = "Disabled"
38          if self.config["dump"]:
39          try:
40          data = proc_layer.read(offset, chunk_size, pad=True)
41          file_data.write(data)
42          file_output = file_handle.preferred_filename
43          except exceptions.InvalidAddressException:
44          file_output = "Error outputting to file"
45          vollog.debug(
46          "Unable to write {}'s address {} to {}".format(
47          proc_layer_name,
48          offset,
49          file_handle.preferred_filename,
50          )
51          )
52
53          yield (
54          0,
55          (
56          pid,
57          format_hints.Hex(offset),
58          format_hints.Hex(mapped_offset),
59          format_hints.Hex(chunk_size),
60          format_hints.Hex(file_offset),
61          file_output,
62          ),
63          )
64
65          file_offset += chunk_size
66          offset += chunk_size
67          size -= chunk_size
68          mapped_offset += chunk_size
69          else:
70          file_output = "Disabled"
71          if self.config["dump"]:
72          try:
73          data = proc_layer.read(offset, size, pad=True)
74          file_data.write(data)
75          file_output = file_handle.preferred_filename
76          except exceptions.InvalidAddressException:
77          file_output = "Error outputting to file"
78          vollog.debug(
79          "Unable to write {}'s address {} to {}".format(
```

```
80          proc_layer_name,
81          offset,
82          file_handle.preferred_filename,
83          )
84          )
85
86          yield (
87          0,
88          (
89          pid,
90          format_hints.Hex(offset),
91          format_hints.Hex(mapped_offset),
92          format_hints.Hex(mapped_size),
93          format_hints.Hex(file_offset),
94          file_output,
95          ),
96          )
97
98          file_offset += mapped_size
99          offset += mapped_size
```

A.12. Ubuntu 20.04 Kernel 5.4 symbol table und dbgsymbols laden

```
1   $ echo "deb http://ddebs.ubuntu.com $(lsb_release -cs) main restricted
        universe multiverse
2   deb http://ddebs.ubuntu.com $(lsb_release -cs)-updates main restricted
        universe multiverse
3   deb http://ddebs.ubuntu.com $(lsb_release -cs)-proposed main restricted
        universe multiverse" | \
4   sudo tee -a /etc/apt/sources.list.d/ddebs.list
5
6   $ sudo apt install ubuntu-dbgsym-keyring
7
8   $ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys F2EDC64
        DC5AEE1F6B9C621F0C8CAB6595FDFF622
9
10  $ sudo apt-get update
11
12  $sudo apt install linux-image-$(uname -r)-dbgsym
13
14  $./dwarf2json linux --elf /usr/lib/debug/boot/vmlinux-5.4.0-1126-aws --
        system-map /boot/System.map-5.4.0-1126-aws > symbol_table_ubuntu_20_
        4_kernel_5_4.json
```

A.13. Comparsion-Script

```
1   import matplotlib.pyplot as plt
2
3
4   def hex_to_int(hex_str):
5   # Entfernt '0x' falls vorhanden und konvertiert die Hex-Zeichenkette in
        eine Ganzzahl
6   return int(hex_str, 16)
7
8   def int_to_hex(integer):
```

```
9           # Konvertiert eine Ganzzahl in eine Hex-Zeichenkette mit fuehrenden
                Nullen fuer 64-Bit Adressen und fuegt '0x' hinzu
10          return f"0x{integer:016x}"
11
12          def parse_output_file(filename):
13          with open(filename, 'r') as file:
14          data = file.readlines()
15
16          result = []
17          for line in data:
18          parts = line.split()
19          if len(parts) >= 6 and parts[0].isdigit():
20          pid = parts[0]
21          virtual = hex_to_int(parts[1])
22          physical = hex_to_int(parts[2])
23          size = int(parts[3], 16)
24          result.append((pid, virtual, physical, size))
25          return result
26
27          def parse_reference_file(filename):
28          with open(filename, 'r') as file:
29          data = file.readlines()
30
31          result = {}
32          for line in data:
33          parts = line.split()
34          if len(parts) >= 5 and parts[1].isdigit():
35          pid = parts[1]
36          virtual = hex_to_int(parts[2])
37          physical = hex_to_int(parts[3])
38          size = int(parts[4], 16)
39          if virtual not in result:
40          result[virtual] = []
41          result[virtual].append((pid, physical, size))
42          return result
43
44          def check_memory_maps(output_data, reference_data):
45          total_checked = 0
46          total_passed = 0
47          total_failed = 0
48
49          for pid, virtual, physical, size in output_data:
50          virtual_hex = int_to_hex(virtual)
51          print(f"Checking: {virtual_hex} against reference data")
52          if virtual in reference_data:
53          found = False
54          for ref_pid, ref_physical, ref_size in reference_data[virtual]:
55          if ref_physical == physical:
56          found = True
57          total_passed += 1
58          print(f"Check passed for Virtual Address {virtual_hex} with Physical
                Address {int_to_hex(physical)}")
59          break
60          if not found:
61          total_failed += 1
62          print(f"No matching physical address found for Virtual Address {
                virtual_hex}")
63          else:
64          total_failed += 1
65          print(f"Virtual Address {virtual_hex} not found in reference.")
```

```
        total_checked += 1

        # Ausgabe der Statistik
        print("\n--- Memory Map Check Statistics ---")
        print(f"Total Checked: {total_checked}")
        print(f"Total Passed: {total_passed}")
        print(f"Total Failed: {total_failed}")
        print(f"Success Rate: {total_passed / total_checked * 100:.2f}%")

        plot_results(total_passed, total_failed)



        def plot_results(passed, failed):
        labels = ['Passed', 'Failed']
        values = [passed, failed]

        fig, ax = plt.subplots()
        ax.bar(labels, values, color=['green', 'red'])
        ax.set_ylabel('Counts')
        ax.set_title('Results of Memory Map Checks')
        ax.set_ylim(0, max(passed, failed) + 10)

        plt.show()


        output_data = parse_output_file('output.txt')
        reference_data = parse_reference_file('reference_pycharm_dump.txt')
        check_memory_maps(output_data, reference_data)
```

```
        $ gcc create_ram_pattern.c -o create_ram_pattern
        $ ./create_ram_pattern
```

```
        Das hier spaeter in den Fliesstext einfuegen: (Beispielausgabe)

        Virtuelle Adresse des Musters: 0x7fa263b5b000
        PID: 46416
        Pagemap-Datei: /proc/46416/pagemap
        Druecken Sie Enter zum Beenden des Programms...
```

A.14. Ubuntu 20.04 auf Kernel 5.4 zurücksetzen von Kernel 5.15

```
        sudo apt update
        sudo apt install -y linux-aws-lts-20.04

        sudo grep 'menuentry \|submenu ' /boot/grub/grub.cfg | cut -f2 -d "'"
            -> Ausgabe fuer den Befehl: Ubuntu
                    Advanced options for Ubuntu
                    Ubuntu, with Linux 5.15.0-107-generic
                    Ubuntu, with Linux 5.15.0-107-generic (recovery mode)
                    Ubuntu, with Linux 5.15.0-67-generic
                    Ubuntu, with Linux 5.15.0-67-generic (recovery mode)
                    Ubuntu, with Linux 5.4.0-1126-aws
                    Ubuntu, with Linux 5.4.0-1126-aws (recovery mode)
                    Memory test (memtest86+)
                    Memory test (memtest86+, serial console 115200)
```

```
15
16
17          sudo cp /etc/default/grub /etc/default/grub.bak                  ->
                BACKUP
18
19          sudo vim /etc/default/grub
20                  -> Anpassung: GRUB_DEFAULT='Advanced options for Ubuntu>Ubuntu,
                        with Linux 5.4.0-1126-aws'
21
22
23
24
25
26
27
28          -------
29
30
31          Fuer die Symbolfiles musste fuer dwarf2json auf Ubuntu 20.04 mit Kernel
                5.4 go geupdated werden
32          -> Voraussetzung mindestens go 1.18
```