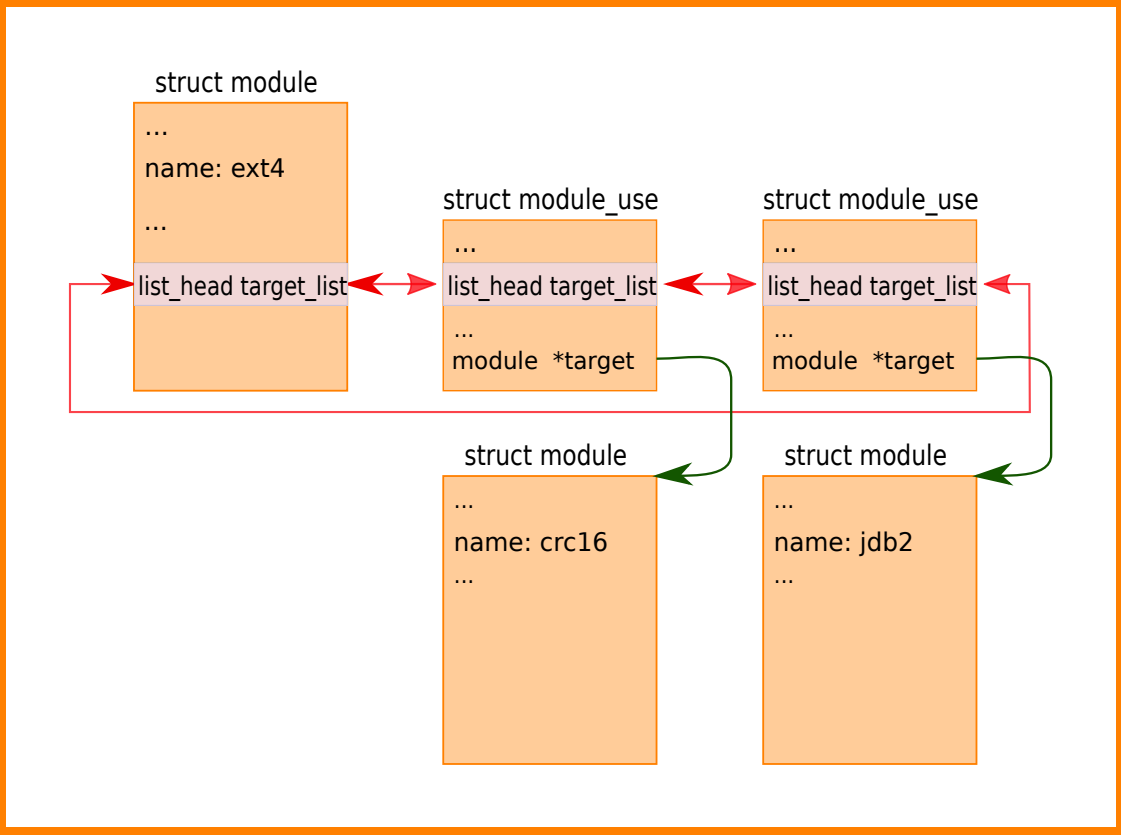
Open C3S
Open Competence Center for Cyber Security

UNIVERSITÄT PASSAU
Fakultät für Informatik und Mathematik



Mikromodul 8004: Virtual Machine Introspection

Autoren:
Prof. Dr. Hans P. Reiser
Noëlle Rakotondravony
Johannes Köstler

Universität Passau

Mikromodul 8004: Virtual Machine Introspection

Autoren:

Prof. Dr. Hans P. Reiser

Noëlle Rakotondravony

Johannes Köstler

---

1. Auflage

Universität Passau

GEFÖRDERT VOM
Bundesministerium
für Bildung
und Forschung

AUFSTIEG DURCH BILDUNG >>
OFFENE HOCHSCHULEN

Universität Passau
Fakultät für Informatik und Mathematik
Innstraße 43
94034 Passau

1. Auflage (4. Mai 2017)

Um die Lesbarkeit zu vereinfachen, wird auf die zusätzliche Formulierung der weiblichen Form bei Personenbezeichnungen verzichtet. Wir weisen deshalb darauf hin, dass die Verwendung der männlichen Form explizit als geschlechtsunabhängig verstanden werden soll.

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen 16OH12025 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

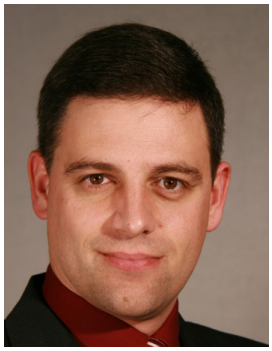
Inhaltsverzeichnis

Einleitung

I. Abkürzungen der Randsymbole und Farbkodierungen

Beispiel	B
Definition	D
Quelltext	Q
Übung	Ü

II. Zu den Autoren



Hans P. Reiser ist Juniorprofessor für Sicherheit in Informationssystemen an der Universität Passau. Schwerpunkte seiner Arbeitsgruppe sind die Weiterentwicklung von Konzepten und Systemen aus dem Bereich der fehler- und einbruchstoleranten Replikation, die frühzeitliche und umfassende Erkennung von Sicherheitsproblemen und Sicherheitsvorfällen in Cloud-Umgebungen sowie die Erforschung neuartiger Sicherheitskonzepte auf Hypervisorebene.

Noëlle Rakotondravony ist seit September 2015 wissenschaftliche Mitarbeiterin in der Arbeitsgruppe von Prof. Hans P. Reiser an der Universität Passau.

Johannes Köstler ist seit Mai 2015 wissenschaftlicher Mitarbeiter in der Arbeitsgruppe von Prof. Hans P. Reiser an der Universität Passau.

Mikromodul: Virtual Machine Introspection

1 Lernziele

Nach Bearbeitung dieses Mikromoduls sind Sie mit Theorie und Praxis von Virtual Machine Introspection (VMI) vertraut. Sie kennen die grundlegende Funktionsweise und die verschiedenen Anwendungsmöglichkeiten von VMI. Sie kennen den aktuellen Stand der Forschung zu den zwei Ausprägungen des Problems der semantischen Lücke (weak semantic gap, strong semantic gap) und können auf dieser Grundlage die Grenzen und Risiken einer Fehlinterpretation bei Virtual Machine Introspection einschätzen. Darüber hinaus sind Sie mit den wesentlichen Datenstrukturen eines Betriebssystemkerns am Beispiel von Linux vertraut. Sie können das Werkzeug Volatility zusammen mit VMI einsetzen, um laufende Systeme in virtuellen Maschinen mit existierenden Analyse-Plugins zu untersuchen.

2 Virtual Machine Introspection

2.1 Begriff

Der Begriff der Introspektion virtueller Maschinen (Virtual Machine Introspection – VMI) geht zurück auf einen Artikel (Garfinkel und Rosenblum, 2003), der VMI als sehr vorteilhafte Technik zur Einbruchserkennung (Intrusion Detection) vorschlägt. Bei VMI greift ein untersuchendes System (das *Analysesystem*) von außen auf den internen Zustand eines untersuchten Systems (das *Zielsystem*) zu.

VMI-basierte Einbruchserkennung verbindet die Vorteile von netzbasierter Einbruchserkennung und hostbasierter Einbruchserkennung. Während erstere zwar eine starke Isolation zwischen Analysesystem und Zielsystem gewährleistet, bietet die Beobachtung des Netzverkehrs nur eingeschränkten Einblick in interne Vorgänge im Zielsystem. Auf der anderen Seite bieten hostbasierte Systeme potentiell sehr guten Einblick in den internen Systemzustand, sind aber als Teil des Zielsystems nicht von diesem isoliert und können dementsprechend leicht von einem Angreifer manipuliert werden.

Diese Vorteile werden durch Ausnutzung von drei grundlegenden Eigenschaften von virtuellen Maschinen ermöglicht: Isolation, Inspektion und Interposition.

Isolation bedeutet, dass das Analysesystem im Idealfall vollständig vom Zielsystem getrennt ist. Dies wird dadurch erreicht, dass das Zielsystem in einer virtuellen Maschine ausgeführt wird. Das Analysesystem befindet sich außerhalb dieser virtuellen Maschine. Eine mögliche Variante ist es, das Analysesystem direkt in den Hypervisor zu integrieren. In der Praxis oft üblich ist bei einem Bare-Metal-Hypervisor die Ausführung in einer privilegierten virtuellen Maschine wie *Dom0* beim Hypervisor Xen und bei einem Hosted-Hypervisor als Prozess des Host-Betriebssystems. Eine weitere vorteilhafte Variante ist die Verwendung einer separaten virtuellen Maschine (*Monitoring Virtual Machine*), die sowohl vom Zielsystem als auch von allen anderen Systemkomponenten isoliert ist.

Diese Isolation muss nicht immer perfekt sein. Unwahrscheinlich, aber nicht undenkbar ist, dass ein Angreifer, der das Zielsystem kontrolliert, von diesem aus Schwachstellen in der Schnittstelle zum Hypervisor ausnutzt und so zunächst die Kontrolle über den Hypervisor und in der Folge auch über das Analysesystem erlangt. Häufiger sind in der Praxis Auswirkungen auf die Performance des Gastsystems, die von diesem bemerkt werden können. Auf mögliche Wechselwirkungen zwischen virtuellen Maschinen geht das Mikromodul Seitenkanäle genauer ein.

Inspektion bedeutet, dass das Analysesystem mit Hilfe des Hypervisors Zugriff auf den Zustand des Zielsystems hat. Den wichtigsten Teil dieses Zustands stellt der virtuelle Arbeitsspeicher des Zielsystems dar. Er umfasst aber auch den Zustand der CPU-Register sowie der I/O-Geräte. Trotz Isolation kann so das untersuchende System ein sehr detailliertes Bild des Zielsystems erlangen.

Interposition bedeutet, dass der Hypervisor in der Lage ist, zahlreiche Operationen innerhalb des Zielsystems (wie das Ausführen von privilegierten CPU-Operationen und den Zugriff auf I/O-Geräte) abzufangen. Hier lässt sich ein Hypervisor leicht so erweitern, dass ein Analysesystem über derartige Ereignisse automatisch informiert wird.

Bei der VMI-basierten Untersuchung eines Zielsystems lassen sich *aktive* und *passive* Methoden unterscheiden:

- Bei *aktiver VMI* erzeugt der Hypervisor selbst Benachrichtigungen über Ereignisse, die dem Analysesystem zugestellt werden; hierzu kann beispielsweise der Zugriff auf eine bestimmte Speicherseite oder die Ausführung einer privilegierten CPU-Instruktion gezählt werden.
- Bei *passiver VMI* stellt der Hypervisor lediglich eine Schnittstelle bereit, über die das Analysesystem z. B. periodisch den Zustand des Zielsystems überprüfen kann.

Viele Anwendungen von VMI beschränken sich, wie die ursprüngliche Arbeit von Garfinkel und Rosenblum (2003), auf das Lesen des Zustands (also Hauptspeicher, CPU-Register, I/O-Geräte) der virtuellen Maschine. In ähnlicher Form ist es aber auch möglich, das Zielsystem zu verändern. Während dies einerseits als Missbrauchsmöglichkeit betrachtet werden kann, lassen sich durchaus auch nützliche Vorteile aus dieser Technik erzielen. So kann beispielsweise das gezielte Injizieren von Haltepunkten verwendet werden, um dadurch aktive Benachrichtigung zu bestimmten Vorgängen im Zielsystem (beispielsweise das Ausführen von Systemaufrufen (*system calls*) oder bestimmten Funktionen) zu erzeugen (Deng et al., 2013). Es ist auch denkbar, dadurch gezielt bestimmte Fehler oder Schwachstellen von außen zu beheben oder sogar einzelne Prozesse z. B. mit Monitoringaufgaben in das Zielsystem von außen einzubringen (Gu et al., 2011).

2.2 Anwendungsbeispiele

Seitdem VMI erstmals als nützliche Technik beschrieben wurde, haben Forschungsarbeiten und praktische Systeme gezeigt, dass sich diese grundlegende Technik für viele unterschiedliche Anwendungszwecke einsetzen lässt.

Intrusion Detection ist dabei der ursprüngliche Zweck, der von Garfinkel und Rosenblum (2003) betrachtet wurde. Hierbei besteht der Vorteil gegenüber bereits etablierten hostbasierten Verfahren in der Isolation zwischen Zielsystem und Analysesystem.

Ein damit verwandter Anwendungszweck sind *Virus-Scanner*, die ebenfalls ein Zielsystem detailliert analysieren können, ohne dass die Analysesoftware durch die Schadsoftware manipuliert oder deaktiviert werden kann (Jiang et al., 2007).

Im Bereich der *digitalen Forensik* ergeben sich ähnliche Vorteile. Für eine statische Untersuchung von Speicherabbildern werden herkömmlich Werkzeuge verwendet, die auf dem Zielsystem ausgeführt werden und daher immer eine Interaktion und folglich eine mögliche Veränderung des Zielsystems bedeuten. Diese Speicherabbilder lassen sich aber auch extern über VMI erzeugen. Darüber hinaus ermöglicht

VMI eine dynamische Analyse eines laufenden Systems (Poore et al., 2013). Hierbei wird statt eines statischen Schnappschusses zu einem einzelnen Zeitpunkt das Verhalten über einen längeren Zeitraum hinweg beobachtet.

Eine verwandte, aber dennoch andersartige Verwendung ist der Einsatz zur *Analyse von Malware*. Ein bekanntes System für Windows-Gastsysteme, die auf dem Xen-Hypervisor ausgeführt werden, ist DRAKVUF (Lengyel et al., 2014).

Auch lässt sich VMI zum Zwecke des Compliance-Monitoring einsetzen. Beispielsweise zeigt die Arbeit von Kienzle et al. (2010), wie sich mittels VMI Eigenschaften des Zielsystems wie verwendetes Betriebssystem, Softwareversionen und Konfigurationen feststellen und damit die Übereinstimmung mit vorhandenen Sicherheits-Policies überprüfen lassen.

Ein anders gearteter Nutzen von VMI zielt auf eine Ressourcen-Optimierung: Mittels VMI lassen sich identische Speicherseiten in mehreren virtuellen Maschinen erkennen und diese dann ressourcensparend im physischen Speicher deduplizieren. Ebenso lässt sich vor einer Migration einer virtuellen Maschine von außen prüfen, welche Speicherseiten der virtuellen Maschine nicht in Verwendung sind und entsprechend auch nicht bei der Migration übertragen werden müssen (Chiang et al., 2013).

2.3 Herausforderungen

Die gerade genannten Anwendungsmöglichkeiten zeugen von den vielfältigen Vorteilen, die VMI haben kann. Diesen Vorteilen stehen in der Praxis aber auch einige Probleme gegenüber, die sich im Wesentlichen in mangelnde Unterstützung auf existierenden Plattformen, Performance-Probleme sowie die sogenannte „semantische Lücke“ (semantic gap) untergliedern lassen.

Bei der *Plattformunterstützung* stellt derzeit *LibVMI* die Bibliothek mit der größten Verbreitung dar. *LibVMI* wird als Open-Source-Projekt in zahlreichen VMI-Projekten und -Werkzeugen eingesetzt und unterstützt als Virtualisierungsumgebung Xen, KVM und QEMU. Auf anderen Hypervisoren sind meist keine oder nur proprietäre, nicht öffentlich dokumentierte Schnittstellen für VMI vorhanden.

Ein weiterer Aspekt sind *Performance*-Einflüsse von VMI. Hierzu ist es notwendig, jeden Einzelfall individuell zu betrachten, weil diese Einflüsse über einen sehr großen Bereich variieren können. Auch liegen mehrere mögliche Ursachen zugrunde. Zum einen kann es für komplexere Untersuchungen notwendig sein, sicherzustellen, dass sich die betrachteten Datenstrukturen während der Untersuchung nicht verändern, was in der Praxis heißt, dass das laufende Zielsystem angehalten werden muss. In diesem Zusammenhang werden in einzelnen Forschungsarbeiten Copy-on-Write-Techniken diskutiert, mit denen sich dieser Nachteil reduzieren lässt. Allerdings werden solche Techniken derzeit von praktischen Systemen nicht durchgehend unterstützt. Des Weiteren können sich zusätzliche Kontextwechsel zwischen Gastsystem und Hypervisor z. B. bei Zugriff auf Speicherseiten auf die Ausführungsgeschwindigkeit des Gastes auswirken. Ein noch gravierenderer Einfluss ist zu erwarten, wenn für das Tracing von Systemaufrufen oder anderen Funktionen solche Kontextwechsel mit hoher Frequenz verursacht werden. Eine genauere Diskussion von verschiedenen VMI-Techniken und eine Evaluation des Einflusses der VMI-basierten Untersuchung auf den Ressourcenverbrauch und die Beeinflussung der Performance der überwachten virtuellen Maschine findet sich u. a. bei Hebbal et al. (2015).

Das Problem mit der größten Komplexität stellt allerdings die semantische Lücke dar. Dieses Problem besteht darin, dass letztendlich über VMI der Hauptspeicher des Zielsystems nur als opake Menge an Bytes sichtbar ist, deren sinnvolle Interpretation das Wissen erfordert, welche Datenstrukturen innerhalb des Zielsystems verwendet werden und wo sich diese im Speicher befinden.

Jain et al. (2014) teilen dieses Problem sinnvollerweise in zwei Teilprobleme auf: die schwache und die starke semantische Lücke.

Viele theoretische und praktische Arbeiten zur semantischen Lücke basieren auf der Annahme, dass in der Ziel-VM ein vertrauenswürdiges, gutmütiges Gast-Betriebssystem ausgeführt wird. *Gutmütig* bedeutet hierbei, dass dieses System nicht gezielt bösartig versucht, das VMI-basierte Analysesystem zu stören. Dieses Problem wird als die *schwache semantische Lücke (weak semantic gap)* bezeichnet. Die Lösung dieses Problems erfordert es, herauszufinden, wie das verwendete Gast-Betriebssystem Datenstrukturen im Hauptspeicher ablegt. Dazu existieren bereits zahlreiche erfolgreiche Lösungsansätze, angefangen von einer manuellen Analyse und Spezifikation der Datenstrukturen bis hin zu automatischen Analysen des Gastsystems.

Bei der *starken semantischen Lücke (strong semantic gap)* geht man dagegen davon aus, dass ein Angreifer das Gastsystem kontrolliert, und damit dem Gast-Betriebssystem und den verwendeten Datenstrukturen im Speicher nicht vertraut werden kann. Unter anderem beschreiben Jain et al. (2014) mehrere Möglichkeiten, wie eine solche Modifikation aussehen kann. Beispielsweise könnte ein Angreifer das Gast-Betriebssystem so ändern, dass die tatsächlich verwendeten Datenstrukturen an andere Positionen im Speicher verlagert werden, während an den ursprünglichen Positionen im Speicher (die von einem nicht manipulierten Gastsystem verwendet werden und auf die ein VMI-basiertes Analysetool in der Regel zugreifen würde) verfälschte Informationen abgelegt werden, mit denen der Angreifer dem Analysetool gezielt den von ihm gewünschten Zustand vortäuschen kann.

3 LibVMI

LibVMI ist aus XenAccess, einer Bibliothek für VMI auf Xen-basierten Systemen, hervorgegangen und als Open-Source-Projekt veröffentlicht[1]. Ziel von LibVMI ist es, eine hypervisorübergreifende Schnittstelle für VMI bereitzustellen, wobei derzeit sowohl Xen als auch KVM unterstützt werden. Des Weiteren kann statt auf eine laufende virtuelle Maschine auch auf ein statisches Speicherabbild zugegriffen werden. Beispielsweise kann damit ein VMware-Speichersnapshot auf gleiche Weise (zumindest lesend) untersucht werden wie ein laufendes Xen- oder KVM-System.

Die Funktionalität von LibVMI umfasst sowohl lesenden als auch schreibenden Zugriff auf den Zustand einer virtuellen Maschine. Neben passiver VMI wird auch aktive VMI in Form einer Event-Schnittstelle unterstützt. Neben einer C-API bietet LibVMI auch eine darauf aufbauende Python-API.

LibVMI bietet dabei Unterstützung, um die semantische Lücke zu überbrücken. Die Abläufe dafür sind in Abbildung 1 dargestellt. Darin möchte eine VMI-Anwendung den Wert eines Kernel-Symbols des Zielsystems lesen (Schritt 1). Dazu ermittelt in Schritt 2 LibVMI zunächst zu diesem Symbol die entsprechende virtuelle Adresse. Als nächstes wird über Seitentabellen des Gastsystems die zu dieser virtuellen Adresse (GVA) gehörende physische Adresse (GPA) ermittelt (Schritt 3 und 4).

[1] http://libvmi.com/

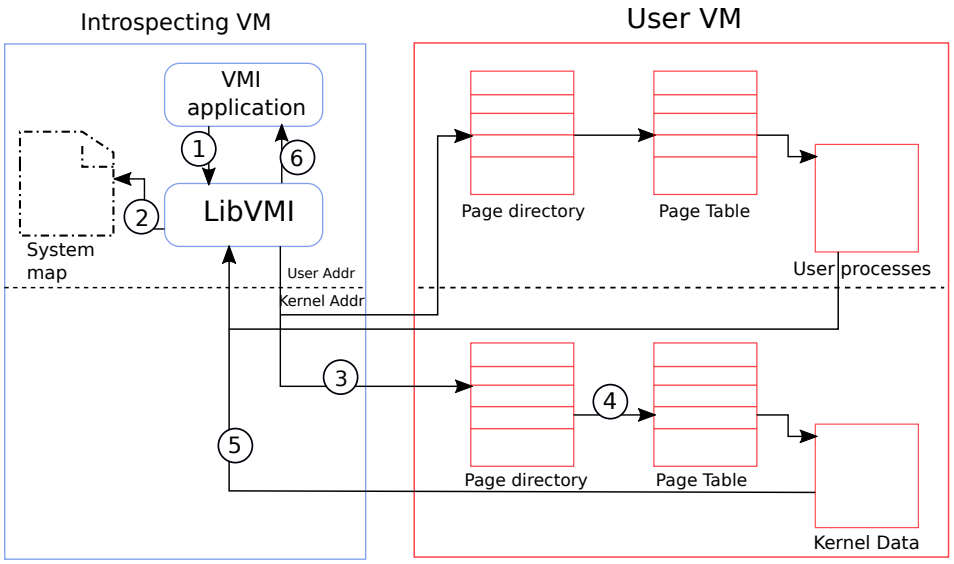


Abb. 1: Grundlegende Funktionsweise von LibVMI

Schließlich wird der entsprechende Wert aus dem Speicher gelesen und an die VMI-Anwendung zurückgegeben (Schritt 5 und 6). Neben dem beschriebenen Zugriff auf den virtuellen Speicher des Gast-Betriebssystems ist in ähnlicher Form auch der Zugriff auf den virtuellen Speicher einzelner Prozesse im Gastsystem möglich.

D

Definition 1: Virtuelle und physische Adressen bei virtuellen Maschinen

- HPA (Host Physical Address): tatsächliche Adressen im physischen Hauptspeicher des Rechners.
- GPA (Guest Physical Address): Adressen in dem virtuellen Speicher, den der Hypervisor dem Gastsystem bereitstellt. Aus Sicht des Gastes entspricht dieser virtuelle Speicher dem physischen Hauptspeicher eines normalen Rechners.
- GVA (Guest Virtual Address): Adressen in einem virtuellen Adressraum des Gastsystems. Dies sind die Adressen, die im ausgeführten Programmcode tatsächlich verwendet werden. Bei einem Betriebssystem mit virtueller Speicherverwaltung, wie sie in nahezu allen modernen Betriebssystemen zum Einsatz kommt, gibt es hier separate virtuelle Adressräume für den Betriebssystemkern sowie für jeden einzelnen Anwendungsprozess.

Wie wir in Abschnitt 4 noch vertiefen werden, sind also für Virtual Machine Introspection drei Fragen von zentraler Bedeutung: Wie erfolgt die Abbildung von virtueller (GVA) auf physische (GPA) Adresse? An welchen Speicheradressen finden sich Kernel-Datenstrukturen? An welchen Offsets innerhalb der Kernel-Datenstrukturen finden sich welche Informationen? Die erforderlichen Informationen zu diesen Fragen werden bei LibVMI durch eine Konfigurationsdatei bereitgestellt (siehe Quelltext 1).

Q

Quelltext 1: libvmi.conf

```
winxpsp2 {
    ostype = "Windows";
    win_tasks   = 0x88;
    win_pdbase  = 0x18;
    win_pid     = 0x84;
    win_kdvb    = 0x80544ce0;
}

[one-1108] {
        ostype = "Linux";
        sysmap = "/root/System.map-3.16.0-4-amd64\_debian\_hvm"
        linux_name = 0x4f0;
        linux_tasks = 0x280;
        linux_mm = 0x2d0;
        linux_pid = 0x334;
        linux_pgd = 0x40;
}
```

Wie die erforderlichen Informationen für diese Konfigurationsdatei in Erfahrung gebracht werden, hängt vom Gastsystem ab. Für praktische Übungen zu diesem Studienbrief wird eine vorbereitete Konfiguration bereitgestellt. Grundsätzlich können die erforderlichen Informationen wie folgt ermittelt werden:

Für Linux-Gastsysteme stellt LibVMI den Quellcode eines Kernelmoduls bereit (`findoffsets.c` unter `libvmi/tools/linux-offset-finder`), das im Gastsystem kompiliert werden muss. Das dabei entstehende Modul `findoffsets.ko` kann mit `insmod` in den Kernel geladen werden und erzeugt dabei eine Ausgabe, die direkt in die Datei `/etc/libvmi.conf` kopiert werden kann. Zudem muss vom Gastsystem die Datei `Sysmap.map` in das Analysesystem kopiert und mit dem Eintrag `system` referenziert werden.

Für Windows-Gastsysteme bietet LibVMI die Möglichkeit, die erforderlichen Information aus einem Speicherabbild zu ermitteln. Hierzu müssen im Analysesystem mehrere Tools installiert werden. Diese Tools extrahieren Daten aus dem Speicherabbild (u. a. die OS-GUID), laden von einem Microsoft-Server dazu passende Debug-Informationen, werten diese aus und erzeugen daraus mittels des Tools `createConfig.py` unter `libvmi/tools/windows-offset-finder/` die passenden Einträge für `/etc/libvmi.conf`.

Für zusätzliche Details hierzu wird an dieser Stelle ergänzend auf die unter https://github.com/libvmi/libvmi verfügbare Dokumentation zu LibVMI verwiesen.

4 Volatility und Kernel-Datenstrukturen: Beispiel Linux

In diesem Abschnitt betrachten wir am Beispiel eines Linux-Gastsystems, wie die wichtigsten Kernel-Datenstrukturen aufgebaut sind und wie sich diese komfortabel mit Volatility untersuchen lassen. Das Volatility-Framework ist ein Open-Source-Werkzeug, das ursprünglich zur Analyse von statischen Speicherabbildern entwickelt wurde[2]. Vertiefte Informationen zur Verwendung von Volatility finden sich in dem Buch „The Art of Memory Forensics“ (Ligh et al., 2014).

[2] http://www.volatilityfoundation.org

4.1 Volatility-Grundlagen

Volatility liest bei der Initialisierung zwei wichtige Informationen aus einem *Profile*: den Aufbau von Kernel-Datenstrukturen und die Adressen von Kernel-Symbolen im Speicher. Unter Linux findet sich letztere Informationen in der Datei `System.map`, während erstere Information aus dem Quellcode des Betriebssystem-Kernels generiert werden kann. Beide Informationen hängen stark sowohl von der jeweils verwendeten Kernel-Version als auch von der spezifischen Kernel-Konfiguration ab. Bei einem individuell erzeugten Linux-Kernel ist daher auch ein dafür maßgeschneidertes Profil zu erstellen. Für die meisten Standard-Linux-Distributionen sind dagegen bereits fertige Profile verfügbar.

Für die in diesem Studienbrief beschriebenen Beispiele und Übungsaufgaben stellen wir Ihnen als Begleitmaterial eine vorkonfigurierte Volatility-Installation sowie einen Hauptspeicher-Abzug eines Mustersystems bereit. Die vorkonfigurierte Installation können Sie nutzen, indem sie sich zu unserer Übungsumgebung per VPN verbinden und per SSH wie im Begleitmaterial beschrieben einloggen. Bei allen Beispielen, die mit dem statischen Hauptspeicher-Abzug arbeiten, können Sie diesen problemlos auf ein eigenes System kopieren und eine mit passendem Profil konfigurierte eigene Volatility-Installation verwenden. Alle Beispiele zur Live-Analyse dagegen beruhen direkt auf unserer Übungsumgebung und sind nur mit größerem Installationsaufwand auf ein eigenes System übertragbar.

In Volatility stellt der *Adressraum (addrspace)* das zentrale Basiselement für den Zugriff auf den Speicher einer virtuellen Maschine dar, wobei gewöhnlich eine Hierarchie von mehreren Adressraum-Instanzen erzeugt wird. Beispiel 1 zeigt für unseren Muster-Speicherdump diesen Aufbau: Die oberste Schicht (Schritt [1]) ist für die Abbildung von virtuellen Adressen auf physische Adressen für die Intel-x86-Architektur zuständig, die letzte Schicht (Schritt [3]) bildet schließlich den Speicherzugriff auf Lese-/Schreiboperationen auf eine Datei ab, entsprechend dem Dateiformat des Speicherabbilds.

Beispiel 1: Adressraum-Hierarchie in Volatility

```
In [1]:  addrspace()
Out[1]:  <volatility.plugins.addrspaces.amd64.AMD64PagedMemory
         at 0x7f37d8d5c610>

In [2]:  addrspace().base
Out[2]:  <volatility.plugins.addrspaces.lime.LimeAddressSpace
                 at 0x7f37da268f90>

In [3]:  addrspace().base.base
Out[3]:  <volatility.plugins.addrspaces.standard.FileAddressSpace
                 at 0x7f37da268f10>
```

B

In der interaktiven Volatility-Shell ist es mit Hilfe des Adressraums möglich, auf Speicher zuzugreifen, ähnlich wie dies mit Hilfe von LibVMI direkt möglich ist. In der Praxis wird man in vielen Fällen diese grundlegenden Methoden nicht direkt verwenden, sondern stattdessen auf übergeordnete Funktionen zurückgreifen, die diese intern verwenden. Darauf gehen wir ab Abschnitt 4.2 genauer ein.

Beispiel 2 illustriert die Verwendung der grundlegenden Funktionen eines Adressraums. Die einzelnen Funktionen erfüllen folgende Aufgaben:

- `addrspace`: gibt ein Objekt zurück, das den Adressraum repräsentiert;

- `vtop`: bildet eine virtuelle Adresse (GVA) auf eine physische Adresse (GPA) ab;
- `read_*_phys`: liest Daten von der angegebenen physischen Adresse.

In dem Beispiel wird also die virtuelle Adresse 0xffffffff8181a460L in der Variablen `virt` gespeichert und in eine physische Adresse umgerechnet [3]. Dann wird zunächst der Inhalt der beiden 8-byte-Speicherworte (Typ `long long`) ab dieser Adresse ermittelt [5, 6]. Danach werden zwei 8-byte-Speicherworte ab der Adresse `virt+0x4f0` gelesen und diese mittels der Python-Funktion „pack“ in ein 16 Zeichen langes byte-Array konvertiert. Welche Bedeutung die gelesenen Speicherworte haben, wird im nächsten Abschnitt deutlich werden.

B

Beispiel 2: Direkter Speicherzugriff in der Volatility-Shell

```
In [1]:  a = addrspace()
In [2]:  virt = 0xffffffff8181a460L
In [3]:  phys = a.vtop(virt)

In [4]:  hex(phys)
Out[4]:  0x181a460L

In [5]:  hex(a.read_long_long_phys(phys))
Out[5]:  '0x0'

In [6]:  hex(a.read_long_long_phys(phys+8))
Out[6]:  '0xffffffff81800000L'

In [7]:  a.read(virt+0x4f0, 16)
Out[7]: 'swapper/0\x00\x00\x00\x00\x00\x00\x00'
```

Linux verwendet eine Identitätsabbildung für den Kernel-Adressraum. Das bedeutet, dass alle virtuellen Adressen im Kernel-Kontext ein konstantes Offset von den entsprechenden physischen Adressen haben. Beispielsweise sind für ein 32-bit-Linux die virtuellen Adressen der Speicherseiten des Betriebssystems gleich ihrer physischen Adressen plus ein Offset von 0xc0000000. Für 64-bit-Linux beträgt dieser Offset, wie in Beispiel 2 zu sehen ist, 0xffffffff80000000.

4.2 Einfache Kernel-Symbole

In Volatility werden mit der Kommandozeilenoption `--profile` die zu verwendende Profile-Informationen angegeben. Für Linux ist dies ein zip-Archiv, welches die Datei `System.map` und eine `DWARF-Datei` enthält. `System.map` enthält die Adressen der statischen Kernel-Symbole (d. h. die *Position* der Kernel-Datenstrukturen im Speicher). Die DWARF-Datei enthält Informationen über den internen *Aufbau* der Kernel-Datenstrukturen (die man auf der Grundlage von Kernel-Konfiguration und Kernel-Quellcode generieren kann).

In Volatility sind insbesondere folgenden Kommandos verfügbar, um Informationen aus einem Profile abzufragen:

- `get_obj_size`: ermittelt die Größe einer Datenstruktur (in byte).
- `get_obj_offset`: ermittelt das Offset eines Strukturelementes innerhalb einer Struktur.
- `vtypes`: listet alle Elementen einer Datenstruktur auf.

- `get_symbol`: ermittelt die Adresse einer statischen Kernel-Variable.

Das Beispiel 3 illustriert die Verwendung dieser Funktionen. Wie man am Ende sieht, ist die Adresse `virt` aus Beispiel 2 die Adresse des Kernel-Symbols `init_task`. An dieser Speicheradresse befindet sich eine Datenstruktur vom Typ `task_struct`, welche Informationen zu einem Prozess enthält.

Beispiel 3: Zugriff auf Kernel-Symbole und -Datenstrukturen

```
In [10]:  p = a.profile
In [11]:  p.get_obj_size("task_struct")
Out[11]:  2384

In [12]:  p.get_obj_offset("task_struct", "tasks")
Out[12]:  640

In [13]:  p.vtypes["task_struct"]
Out[13]:
[1784
        'acct_rss_mem1': [1824, ['unsigned long long']],
        'acct_timexpd': [1840, ['unsigned long']],
        [...]

In [14]:  p.get_symbol("init_task")
Out[14]: : 18446744071587341408L

In [15]:  hex(p.get_symbol("init_task"))
Out[15]:  '0xffffffff8181a460L'
```

B

Um den Zugriff auf Kernel-Datenstrukturen zu vereinfachen, können diese in Volatility auf Python-Objekte abgebildet werden. Dies geschieht, wie in Beispiel 4 dargestellt, mit Hilfe der Funktion `obj.Object`. Diese benötigt drei Argumente: einen String, der die betrachtete Kernel-Datenstruktur angibt, einen Adressraum, sowie die virtuelle Adresse des Kernel-Symbols. Wie im Beispiel zu sehen ist, können über die „Punktnotation“ die Werte einzelner Elemente der Struktur direkt abgefragt werden.

Beispiel 4: Abbildung von Kernel-Datenstrukturen auf Python-Objekte

```
In [16]:  i = p.get_symbol("init_task")
In [17]:  init = obj.Object("task_struct", vm=a, offset=i)

In [18]:  init.pid
Out[18]:  [int]: 0

In [19]:  init.comm
Out[19]:  [String comm] @ 0xFFFFFFFF8181A950

In [20]:  init.comm.v()
Out[20]:  'swapper/0\x00\x00\x00\x00\x00\x00\x00\x00'
```

B

Neben dem direkten Abfragen der Elemente einer Struktur mit Hilfe des Feldes `vtypes` eines Profils (siehe Beispiel 3) gibt es in der Volatility-Shell auch eine universelle Komfortfunktion (`dt`), mit der sich Kernelstrukturen und Kernelobjekte übersichtlich ausgeben lassen (siehe Beispiel 5).

B

Beispiel 5: Volatility-Funktion `dt`

```
In [21]:  dt("task_struct")
Out[21]:  'task_struct' (2384 bytes)
0x0   : state                          ['long']
0x8   : stack                          ['pointer', ['void']]
0x10  : usage                          ['__unnamed_0x38e']
[...]

In [22]:  dt(init)
Out[22]:  [task_struct task_struct] @ 0xFFFFFFFF8181A460
0x0   : state                             0
0x8   : stack                          18446744071587233792
0x10  : usage                          18446744071587341424
[...]
```

Die Datenstruktur `task_struct` enthält unter anderem die folgenden wichtigen Informationen:

- `pid`: Prozess-ID;
- `state`: Prozess-Zustand (laufend/bereit/blockiert/Zombie);
- `start_time`: Zeitpunkt, zu dem der Prozess gestartet wurde (relativ zum Systemstart);
- `parent`: Eltern-Prozess (i. d. R. derjenige Prozess, der den aktuellen Prozess gestartet hat);
- `children`: Information über die Prozesse, die vom aktuellen Prozess initiiert wurden;
- `start_code/end_code`: Anfang und Ende des Speichers mit ausführbaren Code;
- `start_data/end_data`: Anfang und Ende des Heap-Speichers des Prozesses;
- `start_stack`: Anfang des Stack-Speichers des Prozesses;
- `arg_start/arg_end`: Anfang und Ende der Kommandozeilenargumente im Speicher;
- `env_start/env_end`: Anfang und Ende der Umgebungsvariablen im Speicher;
- `mmap` und `mm_rb`: virtuelle Speicherbereiche des Prozesses, in zwei verschiedenen Formaten (verkettete Liste, rot-schwarz Baum);
- `pgd`: „Page Global Directory“, oberste Datenstruktur zur Verwaltung des virtuellen Adressraums (Abbildung von virtuellen auf physische Adressen, je ein Verzeichnis pro Prozess).

4.3 Prozesse

Dieser Abschnitt geht nun genauer darauf ein, wie man Informationen über alle existierenden Prozesse (tasks) in einem System ermitteln kann. Dazu muss man wissen, dass im Linux-Kernel die Menge aller Prozesse (unter anderem) durch eine doppelt verkettete Liste von `task_struct`-Datenstrukturen dargestellt wird.

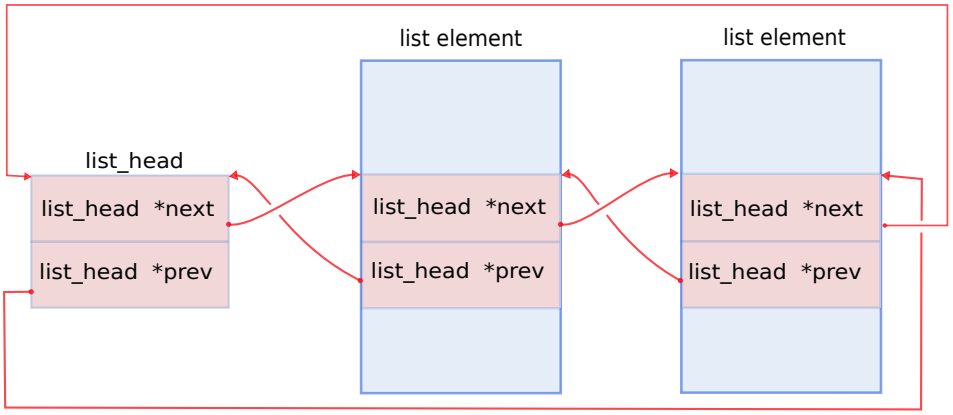


Abb. 2: Doppelt verkettete Liste in Linux-Kernel-Datenstrukturen

Doppelt verkettete Listen

Doppelt verkettete Listen sind eine sehr wichtige Datenstruktur im Linux-Kernel. Viele Datenstrukturen, wie beispielsweise die Prozessliste und die Kernelmodulliste, verwenden doppelt verkettete Listen.

D

Definition 2: Doppelt verkettete Listen im Linux-Kernel

Zur Darstellung einer doppelt verketteten Liste im Linux-Kernel werden sowohl der Anfang der Liste als auch alle Elemente dieser Liste durch eine Datenstruktur vom Typ `list_head` beschrieben. Diese Datenstruktur enthält die beiden Zeiger `next` und `prev`, die auf das nächste und das vorherige Element der Liste verweisen (siehe Abbildung 2).

Die Struktur `list_head` ist meist in eine andere Datenstruktur („Container“) eingebettet. Hierbei ist zu beachten, dass die Vorwärts- und Rückwärtszeiger nicht auf den Container verweisen, sondern auf den im Container eingebetteten `list_head`.

Wir werden später z. B. bei Kernel-Modulen noch Beispiele kennen lernen, bei denen die eigentlichen Listenelemente in einem Container eingebettet sind, während dies bei der Variable, die den Listenanfang darstellt, nicht der Fall ist bzw. wo diese in eine andere, abweichende Datenstruktur eingebettet ist.

Prozessliste

Die Prozessliste ist in Linux eine doppelt verkettete Liste, die alle Prozessbeschreibungen in Form von `task_struct`-Strukturen verbindet. Wenn ein neuer Prozess gestartet wird, wird eine neue Datenstruktur vom Typ `task_struct` erstellt und in der verketteten Liste eingetragen. Jede Instanz von `task_struct` enthält im Feld `tasks` eine `list_head`-Datenstruktur, deren Felder `next` und `prev` auf das nächste und vorherige Element in der Liste verweisen.

In Abbildung 3 ist diese verkettete Liste von Prozess-Datenstrukturen dargestellt. Im Beispiel-Dump befindet sich das Element `tasks` an Offset 0x280 einer `task_struct`.

Der Anfang der Prozessliste ist in der bereits zuvor betrachteten Kernel-Variablen `init_task` (ebenfalls vom Typ `task_struct`) gespeichert. Im Beispiel 6 ist dargestellt, wie man manuell von der `task_struct` in der Variablen `init` zum Nachfolger in der verketteten Liste kommt. Die Adresse des Containers kann errechnet werden,

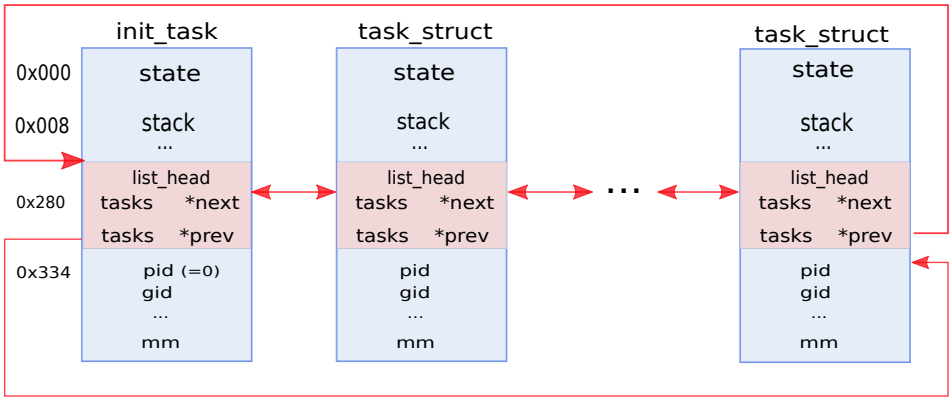


Abb. 3: Linux-Kernel-Datenstrukturen zur Repräsentation einer Prozessliste

indem von der Adresse des Elements `list_head tasks` das Offset dieses Elements innerhalb des Containers `task_struct` subtrahiert wird.

B

Beispiel 6: Manuelle Navigation in der Task-Liste

```
In [25]:  init
Out[25]:  [task_struct task_struct] @ 0xFFFFFFFF8181A460

In [26]:  init.tasks
Out[26]:  [list_head tasks] @ 0xFFFFFFFF8181A6E0

In [27]:  init.tasks.next
Out[27]:  <list_head pointer to [0xFFFF88000C933530]>

In [28]:  n1 = obj.Object("task_struct", vm=a,
                  offset=0xFFFF88000C933530-0x280)
In [29]:  n1.pid
Out[29]:  [int]: 1
.
.
.
```

Volatilitys Listen-Iterator

In Volatility gibt es eine einfachere Art, über verkettete Listen zu iterieren: Die Methode `list_of_type` von einem `list_head`-Objekt erzeugt einen *Generator*, der über alle Elemente der verketteten Liste iteriert. Die Methode `list_of_type` benötigt als Argumente den Typ der Container-Datenstruktur (zum Beispiel `task_struct`) und den Namen des Feldes, unter dem der Container die `list_head`-Datenstruktur enthält (zum Beispiel das Feld `tasks` in einer `task_struct`).

B

Beispiel 7: Iterieren über die Prozessliste

```
In [1]:  a = addrspace()
In [2]:  i = a.profile.get_symbol("init_task")
In [3]:  init = obj.Object("task_struct", vm=a, offset=i)
In [4]:  g = init.tasks.list_of_type("task_struct", "tasks")

In [5]:  for t in g:
...:       print t.pid, t.comm
...:
Out[5]:  1 init
```

```
[...]
77811 bash
77881 insmod
```

Auch gibt es in Volatility eine einfache Möglichkeit, die `task_struct` zu einer bestimmten Prozess-ID (pid) zu ermitteln: Mittels `cc` (change context) lässt sich ein Prozess auswählen, dessen Task-Struktur sich danach als Objekt durch Aufruf von `proc()` ermitteln lässt. Im Beispiel 8 wird auf diesem Weg vom Prozess „sshd“ (PID 464) der Startzeitpunkt (Sekunden nach dem Booten des Systems) ermittelt.

Beispiel 8: Verwendung von cc(pid=...) und proc()

```
In [1]:  cc(pid=464)
Current context: process sshd, pid=464 DTB=0x1ea2000

In [2]:  p = proc()
In [3]:  p
Out[3]:  [task_struct task_struct] @ 0xFFFF880009C32050
In [4]:  dt(p.start_time)
[timespec start_time] @ 0xFFFF88001A9957D8
0x0   : tv_sec                          9
0x8   : tv_nsec                         723438222
```

B

4.4 Kernelmodule

Zu der Verwaltung von Kernelmodulen verwendet Linux eine verkettete Liste von `module`-Strukturen, die Informationen über die geladenen Kernelmodule im Speicher enthält.
Einige Elemente einer `module`-Datenstruktur sind:

- `name`: der Name des Kernelmoduls;
- `kp` (*kernel parameters*): ein Zeiger auf die Parameter, die an das Modul zur Ladezeit übergeben wurden;
- `module_init`: Zeiger auf den Initialisierungscode des Moduls.

Das Element `list` in der `module`-Datenstruktur enthält den `list_head` der verketteten Liste. Die globale Variable `modules` zeigt auf den `list_head` innerhalb des ersten Moduls (Datenstruktur `module`) in der Modulliste (siehe Abbildung 4).

Das Element `target_list` verweist auf eine Liste von Modulen, von denen das betrachtete Modul abhängt. Auch hier ist wieder zu beachten, dass der Verweis auf den Anfang der Liste der abhängigen Module in die Struktur `module` eingebettet

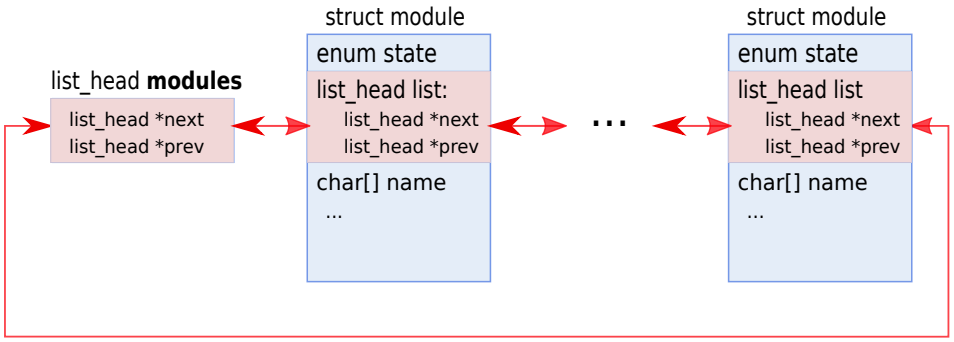


Abb. 4: Linus-Kernel-Datenstrukturen für geladene Kernelmodule

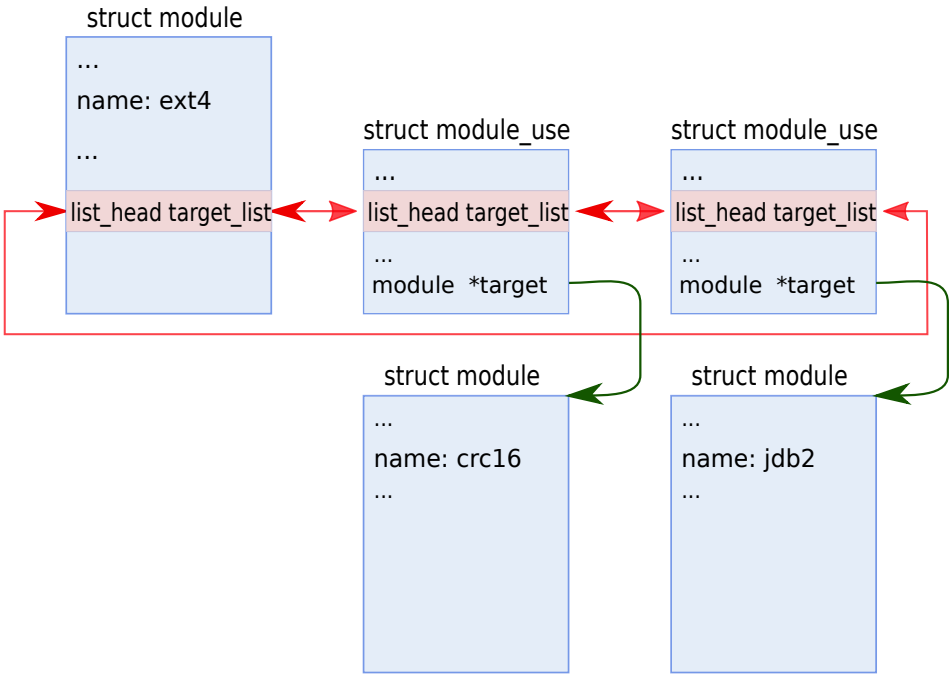


Abb. 5: Abhängigkeiten zwischen Kernelmodule in der Kernelmodul-Datenstruktur

ist. Die einzelnen Elemente der Liste dagegen sind in Datenstrukturen vom Typ `module_use` gespeichert, die intern wiederum auf einzelne Module verweisen (siehe auch Abbildung 5).

4.5 Dateien und Kommunikationsverbindungen

Unter Linux werden viele grundlegende Abstraktionen (Dateien, Sockets, Pipes usw.) auf eine einheitliche Art und Weise behandelt. In einem Anwendungsprogramm werden all diese Elemente durch einen *Dateideskriptor*, eine einfache ganze Zahl, referenziert. Diese Zahl entspricht einem Index in einer prozessspezifischen Tabelle, die im Betriebssystem-Kern gespeichert wird.

Die Adresse der Dateideskriptortabelle (Datentyp: `struct files_struct`) findet sich im Feld `files` der `task_struct`. Die Tabelle enthält insbesondere ein Feld `fd_array`, das aus einem Array aus Zeigern auf `file`-Datenstrukturen besteht. Ein Dateideskriptor entspricht dem Index eines Eintrags in diesem Array.

Über die Liste der Dateideskriptoren lassen sich detaillierte Informationen über alle geöffneten Dateien und Kommunikationsverbindungen eines Prozesses extrahieren. Je nach Typ des Dateideskriptors werden intern verschiedene Arten von Datenstrukturen verwendet, die auch abhängig von der verwendeten Linux-Version unterschiedlich strukturiert sein können. Daher verzichten wir an dieser Stelle auf eine explizite Darstellung, die doch nur unvollständig sein kann. In der Praxis bietet es sich an, auf bereits existierende Volatility-Plugins wie `linux_lsof` zurückzugreifen, welche die relevanten Datenstrukturen aus dem Kernel extrahieren und aufbereiten. Der Quellcode dieser Plugins bietet auch eine gute Grundlage für die Entwicklung eigener Werkzeuge.

4.6 Volatility-Plugins

Für viele Kernel-Datenstrukturen sind Funktionen zum Auslesen von Informationen bereits als fertiges *Volatility-Plugin* implementiert. Sie können ein Plugin nutzen, wenn Sie statt `linux_volshell` den Namen des Plugins verwenden. Durch

den Aufruf von `volatility -info` erhalten Sie eine Liste von allen verfügbaren Volatility-Plugins. Einige wichtige Plugins sind in Beispiel 9 aufgeführt.

B

Beispiel 9: Volatility-Plugins

- `linux_pslist`: listet alle Prozesse in der Prozessliste auf (ausgehend von der Variablen `init_task`).
- `linux_pstree`: gibt die Eltern/Kinder-Beziehung aller Prozesse aus.
- `linux_proc_maps`: zeigt die virtuellen Speicherbereiche aller Prozesse (auf Basis des Felds `mm.map` der `task_struct`). Die Verwendung von „`-p <pid>`“ zeigt die Zuordnung nur vom Prozess `<pid>`.
- `linux_psaux`: gibt die Prozessliste mit der Kommandozeile von jedem Prozess aus.
- `linux_psenv`: gibt die Prozessliste mit den Umgebungsvariablen der Prozesse aus.
- `linux_lsof`: gibt die Liste der geöffneten Dateien aus.

5 Volatility und VMI

5.1 Live-Analyse mit Hilfe von LibVMI

Wird eine laufende virtuelle Maschine untersucht, besteht eine mögliche Vorgehensweise darin, zunächst einen Hauptspeicherabzug zu erstellen und dann diesen wie beschrieben zu analysieren. Es ist daneben ebenfalls möglich, bei der Analyse direkt auf den Hauptspeicher der laufenden virtuellen Maschine zuzugreifen. Hierfür bietet Volatility aktuell zwei grundlegende Möglichkeiten: Zum einen kann mit Hilfe des LibVMI-Adressraums Volatility direkt mittels LibVMI auf den Speicher der virtuellen Maschine zugreifen. Zum anderen kann mit Hilfe von VMIFS ein virtuelles Dateisystem bereitgestellt werden, das aus Sicht von Volatility wie ein Hauptspeicher-Dump aussieht, das aber intern alle Lesezugriffe direkt an den Hauptspeicher der virtuellen Maschine weiterleitet. Wir werden im Folgenden die zweite Option nutzen. Auch VMIFS basiert intern auf LibVMI.

Für die folgenden Aufgaben benötigen Sie ein Xen-basiertes System, auf dem LibVMI, VMIFS und Volatility installiert ist. VMIFS muss in einer Domäne ausgeführt werden, welcher der Zugriff auf andere virtuelle Maschinen mittels VMI gestattet ist. Im einfachsten Fall benötigen Sie also root-Zugriff auf die Dom0 des Systems. In Beispiel 10 ist dargestellt, wie VMI in der bereitgestellten Übungsumgebung mittels besonders berechtigten *Monitoring Virtual Machines (MVM)* möglich ist.

B

Beispiel 10: Infrastruktur-Verwendung für VMI-Live-Analyse

Sie benötigen für alle folgenden Übungen zwei virtuelle Maschinen, die Sie im OpenNebula-Cluster anlegen können: Eine *Production VM (PVM)* (Maschine die untersucht wird, Template *Wordpress*) und eine *Monitoring VM (MVM)* (die Maschine, die Sie verwenden, um andere virtuelle Maschinen zu untersuchen, Template *Wordpress_Monitor*). Notieren Sie sich die von OpenNebula zugeteilte ID der PVM („one-*xxx*“), sowie die IP-Adressen beider virtueller Maschinen.

- Loggen Sie sich mittels SSH in der MVM ein.

- Bearbeiten Sie die Datei `/etc/libvmi.conf`. Ersetzen Sie in der ersten Zeile `one-901` durch die ID Ihrer PVM.
- Nutzen Sie die Kommandozeile `/usr/src/libvmi/tools/vmifs/vmifs name <VM-id> /mnt` um das virtuelle VMIFS zu aktivieren. Als Ergebnis steht der Speicher der laufenden PVM als virtuelle Datei unter `/mnt/mem` bereit.

Sie können mit folgender Kommandozeile testen, ob alles funktioniert:

```
volatility -f /mnt/mem linux_pslist -profile LinuxDebian8x64
```

Als Ergebnis sollten Sie als Ausgabe eine Liste aller laufenden Prozesse in der Ziel-VM erhalten.

Verwenden Sie dies nun zur Durchführung der Experimente, die in Abschnitt 6.3 beschrieben sind.

5.2 Modifikation virtueller Maschinen mit VMI

Volatility ist ein Werkzeug, das vor allem für die Analyse von statischen Speicherdumps entwickelt wurde. Die Bibliothek LibVMI dagegen erlaubt auch einen schreibenden Zugriff auf den Speicher einer VM. Sie können dafür die C-Schnittstelle von LibVMI oder den Python-Wrapper pyVMI verwenden. Letztere Möglichkeit besteht sogar innerhalb einer interaktiven Volatility-Sitzung.

Modifikation des Linux-Banners

Die Kommandozeile `cat /proc/version` gibt die Kernel-Version und andere Kernelinformationen aus. Die dabei ausgegebenen Werte werden beim Generieren eines Linux-Kernels ermittelt und sind im laufenden Betriebssystem in einer Variablen namens `linux_banner` abgelegt. Im folgenden Beispiel werden Volatility und LibVMI kombiniert verwendet, um den Wert dieser Variablen und somit den Inhalt des Linux-Banners zu ändern.

B

Beispiel 11: Modifikation des Linux-Banners

Verwenden Sie das Volatility-Plugin `linux_banner` und betrachten Sie so das Banner Ihrer untersuchten Maschine. Ebenso können Sie in der laufenden virtuellen Maschine auf der Kommandozeile den Befehl `cat /proc/version` ausführen.

```
volatility -f /mnt/mem linux_banner --profile LinuxDebian8x64
```

Starten Sie eine interaktive Volatility-Shell und folgen Sie diesem Beispiel, um eine Modifikation mittels VMI durchzuführen:

```
volatility -f /mnt/mem linux_volshell --profile LinuxDebian8x64
Volatility Foundation Volatility Framework 2.5
Current context: process systemd, pid=1 DTB=0xbe76000
Welcome to volshell! Current memory image is: file:///mnt/mem
To get help, type 'hh()'


In [1]:  a = addrspace()
```

```
In [2]:  p = a.profile
In [3]:  banner_addr = p.get_symbol("linux_banner")

In [4]:  banner = obj.Object("String", vm=a, offset=banner_addr,
          length=256)
In [5]:  print banner
Out[5]:  Linux version 3.16.0-4-amd64 (debian-kernel@lists.debian.
          org) (gcc version 4.8.4 (Debian 4.8.4-1) ) #1 SMP
          Debian 3.16.7-ckt9-3 (2015-04-23)

In [6]:  banner_va = banner.obj_offset
In [7]:  banner_pa = a.vtop(banner_va)

In [8]:  import pyvmi
In [9]:  vmi=pyvmi.init("one-1108", "complete")
Out[9]:  []<pyvmi_instance for one-1108>

In [10]:  vmi.write_64_pa(banner_pa, 123456789)
```

Sie können nun das Volatility-Plugin `linux_banner` nochmals verwenden, um zu prüfen, ob sich die Ausgabe geändert hat.

Erarbeiten Sie nun auf dieser Grundlage eine Lösung für die Aufgabe 12.

6 Übungsaufgaben

6.1 Grundlagen

Übung 1: VM-Eigenschaften und VMI

Erläutern Sie die Bedeutung der Eigenschaften Isolation, Inspektion und Interposition eines Hypervisors für Virtual Machine Introspection. Beschreiben Sie den Unterschied zwischen aktiver und passiver VMI sowie zwischen lesender und schreibender VMI.

Ü

Übung 2: Anwendungsmöglichkeiten und Probleme von VMI

Nennen und erläutern Sie die wichtigsten Anwendungsmöglichkeiten von Virtual Machine Introspection sowie die wichtigsten Probleme, mit denen ein VMI-Werkzeug konfrontiert sein kann.

Ü

6.2 Analyse des statischen Hauptspeicher-Abbilds

Übung 3: Datenfelder der Struktur `task_struct`

Können Sie herausfinden, welche Felder der Datenstruktur `task_struct` in Ausgabenzeile [5] und [7] von Beispiel 2 (Seite 14) ausgegeben wurden?

Ü

Ü

Übung 4: Kernel-Zeit von Speicherabbild

Können Sie herausfinden, zu welcher Zeit das Speicherabbild erstellt wurde?

Tipp 1: Die Kernelvariable `timekeeper` vom Typ `struct timekeeper` enthält Informationen über die aktuelle Uhrzeit. Das Feld `xtime_sec` enthält dabei die Sekunden seit Beginn der „Epoche“ (epoch), d. h. seit dem 1.1.1970 00:00 UTC[3].

Tipp 2: In Python können Sie, nachdem Sie das Zeitmodul durch `import time` geladen haben, die Funktion `time.gmtime(epoch seconds)` verwenden, um die Epoch-Sekunden in die übliche Darstellung mit Datum und Uhrzeit (UTC) umzuwandeln.

Ü

Übung 5: Manuelle Iteration über Prozessliste

Können Sie ausgehend von Beispiel 6 auf Seite 18 den Prozessnamen (Element `comm` einer `task_struct`) für die nächsten drei Prozesse in der Prozessliste auf diesem Weg ermitteln? Welche Python-Anweisungen haben Sie dafür verwendet?

Ü

Übung 6: Ausgabe von Kindprozessen mit automatischer Iteration über Prozessliste

Bei der Ausgabe von Kindprozessen ist eine Besonderheit zu beachten: Der Verweis auf den Anfang der Liste der Kindprozesse befindet sich im Elternprozess im Feld `children`. Die verkettete Liste der Kindprozesse selbst befindet sich dagegen eingebettet innerhalb der `task_struct` im Feld `sibling`.

Modifizieren Sie das Beispiel 7 auf Seite 18 so, dass alle Kindprozesse des Prozesses mit der PID 864 (apache2) ausgegeben werden!

Ü

Übung 7: Zugriff auf Kernelmodule

Ermitteln Sie für das Beispielhauptspeicherabbild folgende Informationen zu Kernelmodulen:

- An welchem Offset in der Modul-Datenstruktur befindet sich der Name des Moduls?
- Verwenden Sie in einer interaktiven Volatility-Shell den Iterator (Methode `list_of_type`), um die Namen von allen geladenen Kernel-Modulen auszugeben.

[3] siehe z. B. http://en.wikipedia.org/wiki/Unix_time für ausführlichere Erläuterungen

Übung 8: Verwendung fertiger Volatility-Plugins

Ü

In den Übungen 3–7 wurden Ergebnisse mit Hilfe der interaktiven Volatility-Shell ermitteln. Versuchen Sie, diese Ergebnisse soweit möglich auch mit fertigen Volatility-Plugins aus dem Hauptspeicherabbild zu gewinnen.

6.3 Live-Analyse einer laufenden virtuellen Maschine

Übung 9: Auslesen der Kernel-Uhrzeit

Ü

Lesen Sie mittels der interaktiven Volatility-Shell die aktuelle Systemzeit der PVM aus (vergleiche dazu auch Übung 4). Wiederholen Sie das Lesen der Zeit mehrfach innerhalb der selben Volatility-Shell. Danach wiederholen Sie dieses Experiment durch Beenden und Neustarten der Volatility-Shell. Was beobachten Sie? Können Sie das Verhalten erklären?

Übung 10: Angriffserkennung mit VMI auf Grundlage der Prozessliste

Ü

Verwenden Sie das Volatility-Plugin `linux_pslist`, um die Liste von laufenden Prozessen auszugeben. Speichern Sie diese in einer Textdatei ab. Ebenso können Sie in der PVM die aktuelle Liste von Prozessen auf der Kommandozeile mit dem Kommando `ps` ausgeben lassen.

Starten Sie nun wie im Begleitmaterial beschrieben einen simulierten Angriff auf die PVM. Dieser Angriff nutzt eine Schwachstelle in der PVM aus und führt Aktionen durch, die typisch für einen realen Angriff sein können.

Extrahieren Sie nun die Liste der laufenden Prozesse erneut mittels des Volatility-Plugins `linux_pslist` und speichern Sie diese ebenfalls ab. Zusätzlich können Sie mit dem Kommando `ps` erneut die Prozessliste innerhalb der PVM ermitteln.

Vergleichen Sie die beiden Liste vor und nach dem Angriff und erklären Sie das Ergebnis. Diskutieren Sie auch Unterschiede zwischen den Ausgaben von `linux_pslist` und `ps`.

Übung 11: Untersuchung der Kernel-Modulliste und der Netzwerkverbindungen

Ü

Wiederholen Sie die Untersuchung von Übung 10, betrachten Sie nun aber Informationen zu geladenen Kernel-Modulen und offenen Netzwerkverbindungen. Welche Erkenntnisse können Sie aus den Ergebnissen dieser und der vorangegangen Übung über den Angriff ziehen?

Ü

Übung 12: Modifizierender Zugriff mittels VMI

Schreiben Sie ein C- oder Python-Programm, das zwei Argumente (`name, pid`) entgegennimmt und die Berechtigungen (*credentials*) des Prozesses mit PID `<pid>` in der untersuchten virtuellen Maschine mit der VM-ID `<name>` zu root-Rechten verändert.

Dazu können Sie zuerst die Liste der laufenden Prozesse in der virtuellen Maschine mit dem Namen `<name>` ermitteln und dann über die Prozessliste iterieren, um den Prozess mit PID `<pid>` zu finden. Die Berechtigungen befinden sich im Element `cred` der Prozess-Datenstruktur.

Dabei ist zu beachten, dass `cred` ein Zeiger auf eine Datenstruktur vom Typ `cred` enthält, die wiederum Informationen zu Berechtigungen enthält. Eine direkte Änderung z. B. des Werts der User-ID (uid) auf 0 (= root) in der Datenstruktur kann dazu führen, dass die Berechtigungen von mehreren Prozessen geändert werden, da mehrere Prozesse auf dieselbe `cred`-Datenstruktur verweisen können. Eine gute Variante besteht daher darin, stattdessen nur den `cred`-Zeiger in der Prozessdatenstruktur auf die `cred`-Struktur eines anderen Prozesses zu ändern, der bereits über root-Rechte verfügt.

Verzeichnisse

I. Abbildungen

II. Beispiele

III. Definitionen

IV. Literatur

Jui-Hao Chiang, Han-Lin Li, und Tzi-cker Chiueh. Introspection-based Memory De-duplication and Migration. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, S. 51–62, New York, NY, USA, 2013. ACM.

Zhui Deng, Xiangyu Zhang, und Dongyan Xu. SPIDER: Stealthy Binary Program Instrumentation and Debugging via Hardware Virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, S. 289–298, New York, NY, USA, 2013. ACM.

Tal Garfinkel und Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, S. 191–206, 2003.

Zhongshu Gu, Zhui Deng, Dongyan Xu, und Xuxian Jiang. Process Implanting: A New Active Introspection Framework for Virtualization. In *Proceedings of the 2011 IEEE 30th International Symposium on Reliable Distributed Systems*, SRDS '11, S. 147–156, Washington, DC, USA, 2011. IEEE Computer Society.

Yacine Hebbal, Sylvie Laniepce, und Jean-Marc Menaud. Virtual Machine Introspection: Techniques and Applications. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, S. 676–685. IEEE, 2015.

Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, und Radu Sion. SoK: Introspections on Trust and the Semantic Gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, S. 605–620, Washington, DC, USA, 2014. IEEE Computer Society.

Xuxian Jiang, Xinyuan Wang, und Dongyan Xu. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, S. 128–138, New York, NY, USA, 2007. ACM.

Darrell Kienzle, Ryan Persaud, und Matthew Elder. Endpoint Configuration Compliance Monitoring via Virtual Machine Introspection. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, S. 1–10, Jan 2010.

Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, und Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Proceedings of the 30th Annual Computer Security Applications Conference*, Dezember 2014.

Michael Hale Ligh, Andrew Case, Jamie Levy, und Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 1st Aufl., 2014. ISBN 1118825098, 9781118825099.

James Poore, Juan Carlos Flores, und Travis Atkison. Evolution of Digital Forensics in Virtualization by Using Virtual Machine Introspection. In *Proceedings of the 51st ACM Southeast Conference*, ACMSE '13, S. 30:1–30:6, New York, NY, USA, 2013. ACM.