Санкт-Петербургский государственный Университет

Математико-механический факультет

Кафедра системного программирования

Восстановление адресного пространства процесса из образа памяти с использованием файла подкачки на платформе Linux

Курсовая работа студента 445 группы

Свидерского Павла Юрьевича

Научный руководитель ст. преп. Губанов Ю.А.

Санкт-Петербург

2012

Содержание

1 Введение

Цифровой криминалистический анализ (Digital Forensic Analysis) — это комплекс экспертных мероприятий, направленных на сбор, хранение и анализ информации, находящейся на компьютерах и компьютерных устройствах. Полученная в результате экспертизы устройства цифровая информация, например, может быть предоставлена в суде как потенциальное доказательство, в случае если данное устройство имело отношение к рассматриваемому судебному делу. Другой немаловажной целью проведения экспертизы является поиск причин отказа или некорректной работы компьютера. Одной из частых причин некорректной работы может являться заражение компьютера вредоносной программой. Как правило, цифровой криминалистический анализ проводится для выяснения, что произошло с компьютером или компьютерным устройством, когда это произошло, как это случилось и кто был вовлечен.

До недавнего времени процедура создания точной и достоверной копии данных с компьютера в ходе цифрового криминалистического анализа ограничивалась только устройством хранения информации, таким как жесткий диск. Это означает, что процесс анализа полагался только на файловую систему накопителя. Многие распространенные файловые системы (NTFS, FAT, ext), ввиду особенностей их реализации, имеют возможность восстановления ранее удаленных файлов, что может быть полезно для выяснения цепочки действий, произведённых с файловой системой. У такого подхода есть свои плюсы. Во-первых, даже с отключенным питанием данные на жестком диске остаются долгое время неизменными. Процедура снятия образа жесткого диска достаточно проста и безопасна, и не влечет изменений содержимого, хранящегося на нём. Во-вторых, уже существует огромное количество утилит и продуктов для анализа файловых систем, что значительно облегчает экспертизу. Однако, есть и большой минус: файловая система, снятая с жесткого диска компьютера, может быть зашифрована. Если неизвестен секретный ключ, процедура расшифровки файловой системы становится практически невозможной.

В наше время популярными становятся твердотельные накопители (SSD, Solid-state drive). Особый принцип работы таких устройств заключается в том, что операционная система может уведомлять твердотельный накопитель о том, какие блоки данных больше не используются и могут быть очищены самим накопителем (см. TRIM). Эта особенность повышает производительность SSD-диска, однако сильно затрудняет или даже делает невозможным процесс восстановления с него удаленных данных.

С развитием облачных хранилищ данных, вся информация, с которой работает пользователем, может храниться и обрабатываться на удалённых серверах в сети Интернет. Как правило, это распределенные системы серверов, получение данных с которых может быть достаточно сложным. При взаимодействии с облачными хранилищами компьютер пользователя не загружает данные целиком на свой жесткий диск, а оперирует только с их небольшими частями в оперативной памяти. Более того, компьютер пользователя может быть и вовсе бездисковой терминальной станцией.

Стоит отметить, что современные компьютеры обладают весьма большим объёмом оперативной памяти (несколько гигабайт). Это позволяет оперировать с большими данными прямо в памяти, не загружая их на жесткий диск. В оперативной памяти также хранится актуальная на данный момент информация о работающей на компьютере операционной системе, запущенных приложениях (в том числе вредоносных), установленных сетевых соединениях с другими компьютерами или веб-узлами, расшифрованных данных и т.д. Более того, может храниться и ранее используемая вышеописанная информация, например список недавно работающих приложений. Полезным применением такой информации может быть восстановление ранее не сохранённого на жёсткий диск документа после краха текстового редактора. Также, к примеру, если компьютер был взломан, типичной задачей анализа его памяти является выяснение списка выполненных злоумышленником действий или команд на взломанной системе.

Таким образом, во время цифрового криминалистического анализа невозможно выяснить полную историю происходящего, основываясь только на анализе устройств хранения. Анализ оперативной памяти компьютера, используемый в дополнение к анализу устройства хранения, помогает решить эту проблему. Образ памяти снимается с рабочей системы и впоследствии анализируется. Стоит отметить, что наиболее интересная информация может быть найдена, если анализируемая система не перезагружалась.

2 Постановка задачи

Образ оперативной памяти содержит данные и процессы (приложения), которые работали на системе во время создания этого образа. Также в образе может содержаться информация о ранее работавших процессах. Одна из ключевых целей анализа памяти — находить и идентифицировать такие процессы, включая потенциально скрытые процессы. Каждый процесс имеет свое виртуальное адресное пространство, которое отображается в оперативную память, не обязательно прямолинейно. Если какой-либо процесс

в системе требует больше оперативной памяти чем доступно, большинство современных операционных систем (Windows, Mac OS X, Linux), для решения этой задачи, используют технологию “подкачка” (swapping). Суть этой технологии заключается в том, что некоторый объем данных (который не “помещается” в оперативную память) временно хранится на жестком диске, в то время как другая часть данных обрабатывается.

На данный момент ни одна из существующих утилит для работы с образом памяти не поддерживает совместную работу с файлом подкачки. Поэтому задачей данной курсовой работы ставится разработка алгоритмов проведения анализа памяти совместно с файлом или разделом подкачки операционной системы Linux, работающей на x86 архитектуре. Будем считать, что образ оперативной памяти и образ раздела подкачки получены в одно время (являются согласованными). Совместный анализ позволит более полно восстановить адресное пространство как работающих процессов на момент снятия образов, так и процессов, умышленно скрытых вредоносными программами путем вмешательства в логику работы ядра операционной системы (руткит[1]). Описанный подход позволит также восстановить адресное пространство недавно завершенных процессов, так как соответствующие структуры, описывающие их, могут долго оставаться в памяти и не перезаписываться.

3 Обзор существующих решений

Предположим, что существует метод снятия образа всей физической памяти работающего компьютера. Такой образ представляет собой мегабайты или даже гигабайты “сырых” данных. Самый простой способ анализа этих данных — это поиск строковых шаблонов. Данный подход относительно надёжный, но, к сожалению, очень медленный при работе с большими образами. Для некоторых задач использование поиска строковых шаблонов в образе памяти может быть очень сложным, например, в задаче вывода списка запущенных процессов операционной системы (ОС). Если версия и производитель операционной системы известны, можно составить шаблоны, которые будут подходить конкретно к данной версии ОС. Однако это невозможно в некоторых случаях, например, для самостоятельно скомпилированного ядра Linux. В данной работе будут описаны альтернативные подходы анализа образа физической памяти.

[1]Руткит — программа или набор программ для скрытия следов присутствия злоумышленника или вредоносной программы в системе.

3.1 Volatility Framework

Volatility Framework представляет собой набор утилит для разностороннего анализа образа физической памяти. Сейчас Volatility Framework поддерживается компанией Volatile Systems. Данный фреймворк является одним из самых функциональных открытых инструментов для анализа образа памяти. Он представляет также платформу для дальнейших исследований в этой области. Volatility Framework обладает следующими возможностями: извлечение информации о запущенных процессах, открытых сетевых соединениях, открытых файлах и загруженных библиотеках для каждого процесса, извлечение виртуального адресного пространства процесса и исполняемого файла из него, и многое другое. Поддерживается как сигнатурный поиск структур, так и поиск на основе отладочной информации (KDBG) и управляющих структур (KPCR). На момент написания данной работы Volatility Framework предоставлял поддержку образов памяти операционных систем семейства Windows (XP, 2003 Server, Vista, 2008 Server, 7). Поддержка ОС Linux появилась в феврале 2011 как отдельная ветка проекта. Возможность анализа различных версий ядра Linux достигается использованием профилей. Профили — это текстовые документы, описывающие содержимое и расположение различных структур ядра в памяти. На данный момент Volatility Framework предоставляет профили всего для нескольких версий ядер операционных систем Debian, Ubuntu и CentOS. Так как для поддержки конкретного ядра требуется иметь соответствующий профиль, необходимо поддерживать большую базу профилей. Естественно, для скомпилированного самостоятельно ядра невозможно заранее создать соответствующий ему профиль. В настоящий момент поддержка ОС Linux не развивается.

3.2 Draugr

Проект Draugr был представлен в 2007 году, но был сделан только один релиз, и, вероятно, инструмент больше не поддерживается. Согласно сайту проекта Draugr, данный инструмент может выводить список запущенных процессов сигнатурным методом и методом списков, разрешать символьные адреса без использования System.map и идентифицировать секции (.text, .data, ...) внутри процессов. Согласно описанию, инструмент draugr имеет встроенный дизассемблер. Вышеперечисленная функциональность показана в демонстративных видео роликах, но на практике же draugr не показал себя работающим, по крайней мере автору этой работы.

4 Механизм виртуальной памяти в архитектуре x86

4.1 Выделение виртуального адресного пространства процесса

Каждый пользовательский процесс в системе должен работать в своём адресном пространстве, определённом набором соответствующих структур управления страницами памяти. Следовательно, одним из методов нахождения всех процессов в образе памяти является нахождение в образе вышеупомянутых структур. Анализируя образ памяти, следует понимать семантику необработанных данных находящихся в образе. В первую очередь она зависит от работающей системы (аппаратное обеспечения, операционная система). Можно выделить два подхода к поиску структур управления страницами памяти: используя особенности реализации ядра операционной системы и используя особенности реализации аппаратного обеспечения.

Первый подход опирается на работу ядра операционной системы. Так как ядро обслуживает все структуры управления страницами памяти, можно найти их расположение проанализировав структуры данных ядра. Однако минус данного подхода заключается в возможности получения недостоверной информации в случае скомпрометированного ядра и его структур данных.

Второй подход является более приемлемым. Он опирается на определённые особенности программно-аппаратного взаимодействия, которым должно удовлетворять даже вредоносное программное обеспечение. В работе [5] описан алгоритм использующий данный подход. Этот алгоритм находит структуры данных, которые используются аппаратным обеспечением для поддержания функционирования процессов ОС. В данной работе для нахождения структур управления страницами памяти будет использоваться второй подход.

4.2 Аппаратное управление страницами памяти

На компьютерах с x86 архитектурой задача разделения адресного пространства процессов решается использованием механизма страничной адресации памяти, предоставляемого аппаратным обеспечением. Аппаратный блок управления страницами преобразует виртуальные адреса каждого процесса в физические (адреса на физической микросхеме памяти). Блок управления страницами считает, что вся оперативная память разбита на страничные кадры фиксированной длины (физические страницы). Каждый страничный кадр содержит одну страницу, то есть его длина равна длине страницы.

Страничный кадр является составной частью оперативной памяти и, следовательно, областью для хранения данных. Необходимо проводить четкое различие между страницей и страничным кадром. Страница — это всего лишь блок данных, который может храниться в любом страничном кадре или на диске.

Структуры данных, отображающие виртуальные адреса в физические, называются таблицами страниц. Они также хранятся в памяти и должны быть должным образом проинициализированы ядром до включения блока управления страницами [1].

Механизм страничной адресации памяти даёт целый ряд преимуществ для процессов: отдельные адресные пространства, виртуальное представление памяти, возможность хранения страниц на диске (область подкачки), разделяемая память.

4.3 Обычное управление страницами

Начиная с процессоров 80386, блок управления страницами в процессорах Intel обрабатывает страницы размером 4 Кбайт. Преобразование 32-битного виртуального адреса происходит посредством двух таблиц: каталог страниц (Page Directory) и таблица страниц (Page Table). Тридцать два бита адреса разделяются на три поля: каталог Directory (старшие 10 бит), таблица Table (средние 10 бит), смещение Offset (младшие 12 бит).

С каждым процессом связан свой каталог страниц. Физический адрес его хранится в управляющем регистре cr3. Структура каталог страниц занимает ровно одну страницу (4 Кбайт) и содержит до 1024 записи (Page Directory Entry) размером 4 байт, указывающие на структуры таблица страниц. Структура таблица страниц также занимает одну страницу и содержит до 1024 записи (Page Table Entry) размером 4 байт, указывающие на физические страницы памяти.

Преобразование виртуального адреса в физический происходит следующим образом: поле Directory виртуального адреса определяет номер записи в каталоге страниц, указывающей на соответствующую таблицу страниц, поле Table определяет номер записи в этой таблице страниц, содержащей физический адрес страничного кадра, а поле Offset определяет относительное смещение внутри страничного кадра (Рис. 1). Таким образом, каталог страниц может адресовать до $1024 \times 1024 \times 4096$ байт $= 4 \times 2^{30} =$ $= 4$ Гб физической памяти.

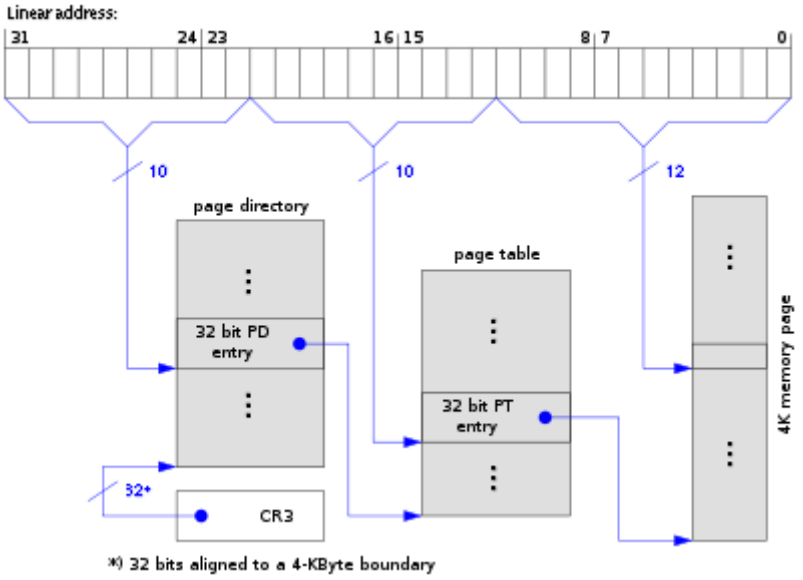

Рис. 1: Трансляция виртуального адреса (обычное управление страницами)

4.4 Механизм расширения физических адресов (PAE)

В старых моделях процессоров Intel, от 80386 до Pentium, использовались 32-разрядные физические адреса. Теоретически в таких системах можно адресовать не более 4 Гбайт оперативной памяти, однако большие высоконагруженные серверы нуждаются в большем объеме памяти. Это привело к разработке другого механизма управления страницами памяти. Одновременно с процессором Pentium Pro компания Intel представила механизм, названный расширением физических адресов, или PAE (Physical Address Extension). С помощью данного механизма стало возможным адресовать до $2^{36} = 64$ Гбайт оперативной памяти (32-битовые виртуальные адреса преобразуются в 36-битовые физические). Расширение физических адресов активизируется с помощью флага PAE в управляющем регистре cr4 процессора.

С использованием механизма PAE память по прежнему разбивается на кадры размером 4 Кбайт. В отличие от обычного управления страницами, преобразование 32-битного виртуального адреса в PAE режиме происходит посредством трёх таблиц. Добавлен третий уровень, названный таблицей указателей на каталог страниц PDPT (Page Directory Pointer Table). Её физический адрес для каждого процесса хранится в управляющем регистре cr3. Эта таблица состоит из четырёх 64-битовых записей. С включенным режимом PAE таблицы каталог страниц и таблица страниц состоят из 512 записей 8 байт каждая.

Преобразование виртуального адреса в физический происходит по такому же принципу, как и в обычном управлении страницами. Виртуальный адрес делится на четыре

поля: старшие 2 бита определяют одну из четырёх записей таблицы PDPT, следующие 9 бит указывают на одну из 512 записей каталога страниц, следующий 9 бит — на одну из 512 записей таблицы страниц и оставшиеся младшие 12 байт определяют относительное смещение внутри страничного кадра (Рис. 2).

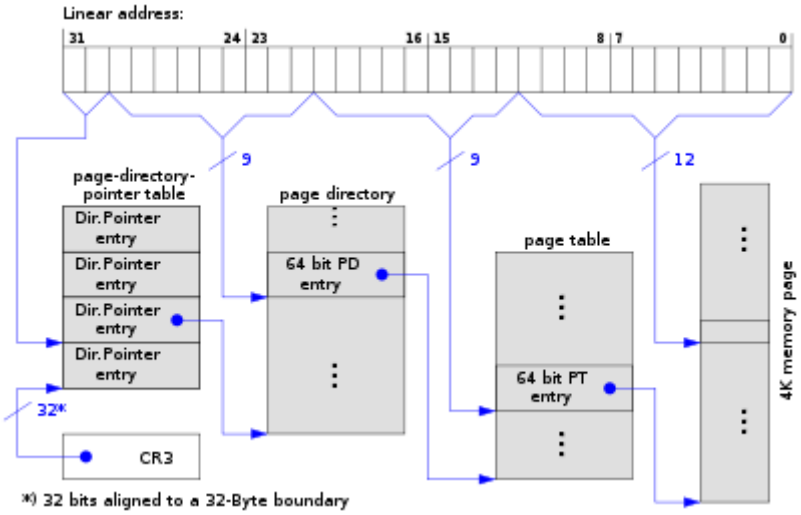

Рис. 2: Трансляция виртуального адреса (механизм расширения физических адресов PAE)

4.5 Расширенное управление страницами

Начиная с модели Pentium, микропроцессоры 80x86 могут использовать расширенное управление страницами, которое позволяет страничным кадрам иметь размер не 4 Кбайт, а 4 Мбайт или 2 Мбайт при включенном механизме PAE. Применение расширенного управления страницами может быть полезно при отображении большого интервала смежных виртуальных адресов в соответствующие интервалы физических адресов. В таком случае экономится память, так как не используются промежуточные структуры таблицы страниц.

На Рис. 3 изображено преобразование виртуального адреса с использованием расширенного управления страницами и выключенным механизмом расширения физических адресов PAE. На Рис. 4 изображено то же самое, но с включенным механизмом PAE. Отметим еще раз, что в первом случае страничные кадры имеют размер 4 Мбайт, а во втором — 2 Мбайт.

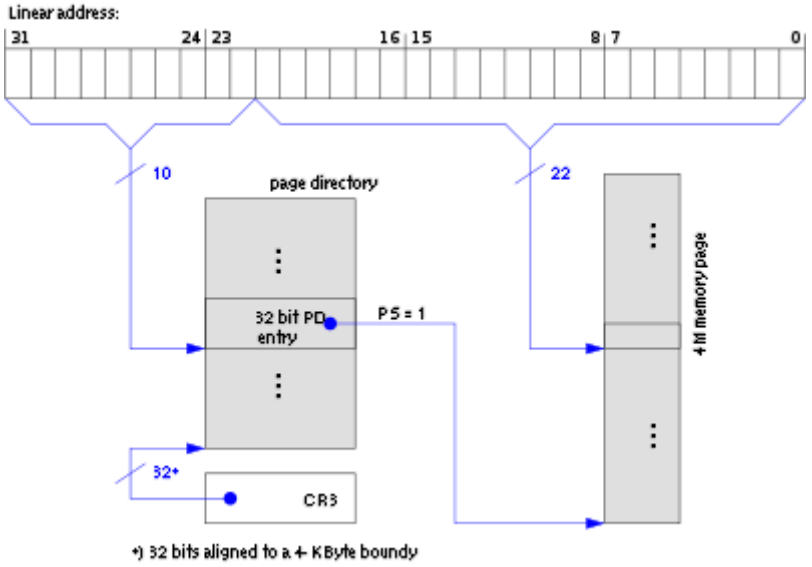

Рис. 3: Трансляция виртуального адреса (расширенное управление страницами)

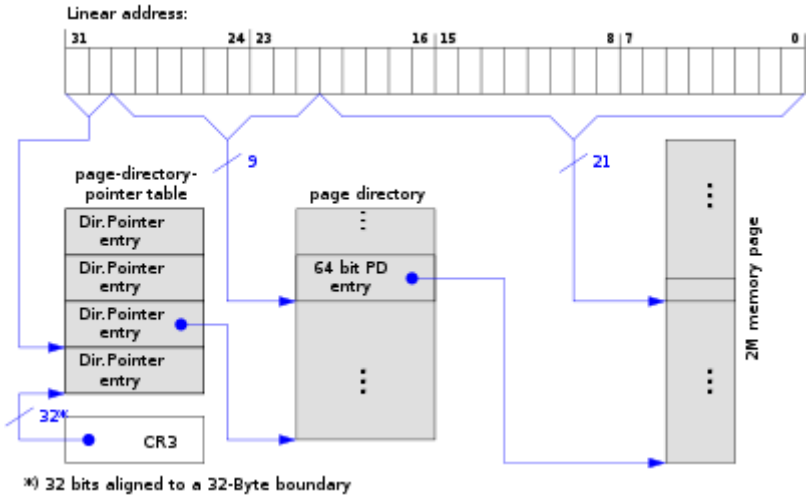

Рис. 4: Трансляция виртуального адреса (расширенное управление страницами с включенным механизмом PAE)

4.6 Записи таблиц страниц

Таблицы страниц состоят из 4-байтовых или 8-байтовых (в случае PAE) записей. Старшие биты записи представляют собой адрес, указывающий на страничный кадр, содержащий следующую в иерархии таблицу страниц преобразования виртуального адреса, или на страничный кадр, содержащий непосредственно страницу (в случае Page Table Entry). Младшие биты — это флаги, характеризующие различные состояния страницы. Записи таблиц страниц имеют определённый формат для каждого конкретного

случая в зависимости от включенного механизма PAE и размера отображаемой страницы (4 Кбайт, 2 Мбайт или 4 Мбайт). Подробности будут описаны в следующих разделах.

5 Решение

В рамках данной курсовой работы исследования будут касаться операционной системы GNU/Linux, работающей на архитектуре x86 как с включённым механизмом расширения физических адресов (PAE) так и без него. Диапазон исследуемых версий ядер ОС GNU/Linux: 2.6.11 – 3.3. Предполагается, что имеются файл образа памяти и файл образа области подкачки, снятые с целевой системы.

Поставленные в данной работе задачи восстановления адресного пространства процессов будут решаться в несколько этапов:

1. Воспользуемся описанным в работе [5] алгоритмом для нахождения в памяти табличных структур каталог страниц (Page Directory) или, при включенном механизме PAE, таблиц указателей на каталог страниц (Page Directory Pointer Table). Данный алгоритм использует особенности программно-аппаратного взаимодействия архитектуры x86 и не зависит от работающей на анализируемой системе версии ядра ОС GNU/Linux. Фактически, для каждого активного процесса в системе мы найдём физический адрес, хранящийся в управляющем регистре cr3 процессора.

2. Для каждой структуры, полученной на предыдущем этапе, выполним обход в глубину её записей (тем самым получим перебор виртуальных адресов процесса в порядке возрастания). Каждая запись PDE в каталоге страниц или PTE в таблице страниц может иметь три состояния:

   - Младший бит записи равен 1 — страница находится в оперативной памяти, и, следовательно, присутствует в снятом образе.
   - Не все биты записи, кроме младшего, равны нулю — страница выгружена в область подкачки.
   - Запись состоит из одних нулей — страница не принадлежит адресному пространству процесса, либо соответствующий страничный кадр еще не был назначен процессу (выделение страниц по требованию).

3. Найдём структуры ядра, описывающие процессы в системе. С помощью них сможем восстановить имена процессов, адресные пространства которых были получены на предыдущих этапах. Также сможем определить работающие в системе процессы скрытые руткитами.

5.1 Алгоритм нахождения потенциальных PGD и PDPT

Каждому процессу в ОС GNU/Linux выделяется 4Гб виртуального адресного пространства. Первые 3 Гбайт являются адресным пространством режима пользователя, последний 1 Гбайт — адресным пространством режима ядра.

Сначала мы найдём потенциальные каталоги страниц, удовлетворяющие условиям отображения адресного пространства режима ядра. Если объём физической памяти в системе менее 896 Мбайт, ядро ОС GNU/Linux отображает линейно всю физическую память в виртуальное адресное пространство процесса, начиная с виртуального адреса `0xC0000000` (четвертый гигабайт). Если объём памяти превосходит 896 Мбайт, то таким образом отображаются её первые 896 Мбайт. В ядре ОС GNU/Linux для отображения адресного пространства режима ядра используется расширенное управление страницами. Страничные кадры, в которые отображается последний гигабайт виртуального адресного пространства, имеют размер 4 Мбайт или 2 Мбайт при включенном режиме PAE. Данный алгоритм для нахождения потенциальных PGD и PDPT в образе памяти фокусируется на аппаратных флагах, выставленных в записях каталога страниц. Последние значащие биты (флаги) записей, отображающие виртуальное адресное пространство процесса режима ядра, должны равняться шаблону `0x1E3`, как изображено на Рис. 5.

Опишем алгоритм для случая адресации с выключенным механизмом PAE. Каталог страниц в памяти занимает 4 Кб и выровнен по границе в 4 Кб. Каталог страниц может содержать до 1024 (0 – 1023) записи. Таким образом, перебираем в образе памяти с шагом в 4 Кб блок данных размером 4 Кб.

Для каждого такого потенциального каталога страниц считаем число записей, удовлетворяющих вышеописанному шаблону и находящихся в последней четверти каталога страниц (записи с номерами 768 – 1023). Эти записи должны соответствовать виртуальным адресам процесса режима ядра. Далее сравниваем полученное число с пороговым значением, выставленным немного меньшим $1/4 \cdot min$(объём физической памяти, 896). Множитель 1/4 обусловлен тем, что размер страничных кадров, в которые отображают-

ся виртуальные адреса, равен 4 Мбайт. Если полученное значение меньше порогового, то данный блок данных исключаем из рассмотрения и переходим к следующему блоку размера 4 Кб.

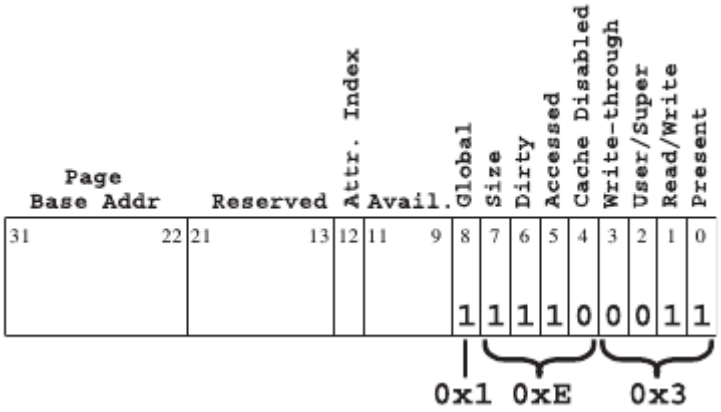

Рис. 5: Запись каталога страниц с выключенным механизмом расширения физических адресов (PAE)

Если для потенциального каталога страниц удовлетворено вышеописанное пороговое условие, переходим к следующему этапу проверки корректности записей, отображающих виртуальное адресное пространство процесса режима пользователя (записи с номерами 0 – 767). Если младший бит записи равен 0, переходим к следующей записи. Младший бит записи равный 1 означает присутствие страницы в оперативной памяти. Проверяем, что поле Page Base Address записи, указывающее на номер страничного кадра оперативной памяти, в котором хранится соответствующая страница, попадает в допустимый диапазон (не превосходит объём оперативной памяти).

Если рассматриваемый блок (потенциальный каталог страниц) удовлетворяет всем описанным выше условиям, то мы помечаем его как действительный каталог страниц. Этот каталог страниц идентифицирует очередной процесс в системе.

Подробнее о пороговых значениях, тестировании и расширении алгоритма для работы в PAE режиме описано в работе [5].

5.2 Получение адресного пространства процесса

Рассмотрим алгоритм получения полного адресного пространства процесса для случая с использованием механизма расширения физических адресов (PAE). Вариант без PAE является частным случаем данного.

Для начала необходимо выбрать процесс, виртуальное адресное пространство кото-

рого собираемся получать. Каждая найденная описанным выше алгоритмом структура PDPT (Page Directory Pointer Table) определяет отдельный процесс в системе (его адресное пространство). Выберем нужную PDPT и выполним обход в глубину связанных с ней табличных структур следующим образом:

1. Перебираем по порядку записи структуры PDPT. Первые три записи отображают адресное пространство режима пользователя, четвертая запись — адресное пространство режима ядра. Выбрав очередную запись, извлекаем из неё поле Page Directory Address (12 – 35 биты записи). Это индекс страницы со структурой каталог страниц (Page Directory) на которую ссылается данная запись. Вычисляем расположение данной страницы в образе памяти (смещение от начала файла) как Page Directory Addres · 4096 байт. Переходим к пункту 2.

2. Получив адрес структуры каталог страниц (Page Directory), перебираем её PDE записи. В зависимости от того, какой тип адресного пространства отображает данный каталог страниц, возможны два варианта:

   - Каталог страниц отображает адресное пространство режима пользователя. Если младший бит записи PDE равняется 1 (флаг Present), получаем из записи индекс страницы со структурой таблица страниц (Page Table) — поле Page Table Address (12 – 35 биты). Смещение этой страницы в файле образа равняется Page Table Address · 4096 байт. Переходим к пункту 3.
   - Каталог страниц отображает адресное пространство режима ядра. Если младший бит записи равняется 1 (флаг Present), получаем из записи индекс соответствующей ей страницы — поле Page Base Address (21 – 35 биты). Так как страницы в данном случае имеют размер 2 Мбайт, смещение этой страницы в файле образа будет равняться Page Base Address · 2 · 2^{20} байт.

3. Перебираем PTE записи полученной таблицы страниц (Page Table). Каждая PTE может иметь три состояния:

   - Младший бит записи равен 1 (флаг Present) — запись ссылает на страницу находящуюся в оперативной памяти, а, следовательно, данная страница присутствует в снятом образе памяти. Индекс этой страницы равняется полю Page Frame Address (12 – 35 биты), а смещение в файле образа — Page Frame Address · 4096 байт.

- Не все биты записи, кроме младшего, равны нулю — страница выгружена в область подкачки. Старшие 32 бита PTE являются идентификатором выгруженной страницы. Идентификатор выгруженной страницы состоит из двух полей: индекс страничного слота (старшие 27 бит) и номер области подкачки (младшие 5 бит). В системе ОС GNU/Linux может использоваться одновременно до 32 областей подкачки. Смещение выгруженной страницы в файле подкачки равняется "индекс страничного слота" $\cdot$ 4096 байт.
- Запись состоит из одних нулей — страница не принадлежит адресному пространству процесса, либо соответствующий страничный кадр еще не был назначен процессу (выделение страниц по требованию). Для получения информации об этой странице необходимо смотреть на структуру ядра `vm_area_struct`, относящуюся к данному процессу.

Для каждой полученной из адресного пространства режима пользователя страницы известен номер записей, по которым совершался переход в структурах PDPT, Page Directory и Page Table, следовательно, можно вычислить соответствующий ей виртуальный адрес. Пусть $pdpte\_index$ = "номер записи (начиная с 0) в структуре PDPT", pde_index = "номер записи PDE (начиная с 0) в каталоге страниц", pte_index = "номер записи PTE (начиная с 0) в таблице страниц". Тогда виртуальный адрес страницы равняется $pdpte\_index << 30 + pde\_index << 21 + pte\_index << 12$, где знаком "<<" обозначена операция побитового сдвига влево. Аналогично для страниц из адресного пространства режима ядра, виртуальный адрес страницы равняется $pdpte\_index << 30 + pde\_index << 21$.

Таким образом, для каждого процесса может быть создан отдельный дамп файл, в который будут последовательно копироваться найденные описанным алгоритмом страницы. Для каждого такого файла необходимо иметь индексный файл, в котором будет определено соотношение виртуального адреса и смещения соответствующей страницы в дамп файле.

5.3 Нахождение структур ядра, описывающие процессы

Чтобы восстановить имена процессов, адресные пространства которых были получены описанным в предыдущем разделе способом, необходимо обратиться к связанным с процессами структурам ядра ОС.

Прежде чем описывать, как ядро отслеживает процессы в системе, стоит отметить важность определённых структур, реализующих двунаправленные списки. В Linux определена структура `list_head`, два поля которой, `next` и `prev`, содержат указатели на следующий и предыдущий элементы двунаправленного списка общего назначения. При этом важно отметить, что указатели содержат адреса других структур `list_head`, а не адреса структур, включающих в себя структуру `list_head` (Рис. 6).

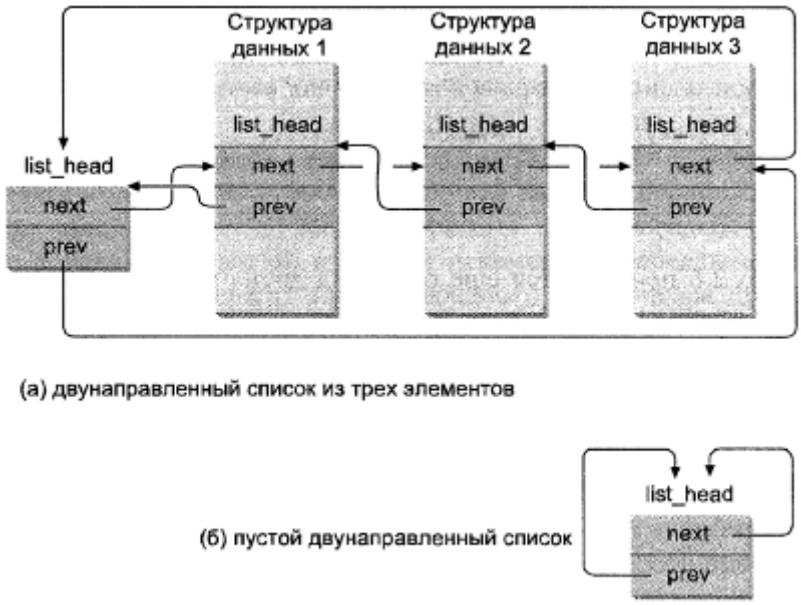

Рис. 6: Двунаправленные списки, построенные из структур listhead

В Linux основной структурой, ассоциированной с каждым процессов, является `task_struct`. Данная структура (дескриптор процесса) описывает процесс для ядра ОС и содержит указатели на другие, связанные с процессом, структуры. Каждая структура `task_struct` включает в себя поле список процессов `tasks` типа `list_head`. Поля `tasks.next` и `tasks.prev` указывают соответственно на следующий и предыдущий процессы в связном списке по отношению к текущему. Голова списка процессов представляет собой дескриптор `init_task` типа `task_struct`. Это дескриптор так называемого нулевого (swapper) процесса, всегда присутствующего в системе.

Имя процесса определяется в структуре `task_struct` полем comm типа `char[16]`. Дескриптор каждого пользовательский процесса в системе содержит указатель `mm` на структуру `mm_struct`. Данная структура содержит всю информацию об адресном пространстве, относящемся к процессу. В рамках поставленной в этой работе задачи, интерес представляет поле `pgd` структуры `mm_struct`. Для каждого процесса оно содержит адрес структуры каталог страниц (Page Directory) или таблицы указателей на каталог страниц (Page Directory Pointer Table) в случае включенного механизма расширения

физических адресов (PAE). Описанная связь структур изображена на Рис. 7.

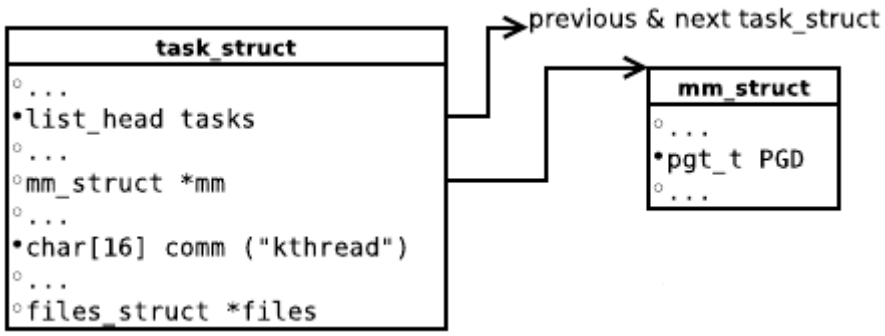

Рис. 7: Дескриптор процесса структура taskstruct

Сложность поиска структуры `task_struct` в образе памяти обусловлена тем, что для общего случая нет устойчивых шаблонов (сигнатур) для её идентификации. Порядок полей и их смещения внутри структур полностью зависят от версии и конфигурации ядра Linux, а также от выбранных опций компиляции. Опишем далее возможные подходы для нахождения структур `task_struct` и вычисления смещений полей `tasks`, `mm`, `comm` и `pgd`.

Для получения адреса дескриптора `init_task` процесса swapper будет использоваться таблица символов ядра. При подключении модуля ядра в Linux все ссылки на глобальные символы ядра (переменные и функции) в объектном коде модуля должны быть заменены на соответствующие адреса. Для этого ядро использует специальную таблицу символов. Эта таблица хранится в секции `__ksymtab` сегмента кода ядра, а следовательно, в первых мегабайтах физической памяти, куда загружается ядро. Данная область памяти никогда не выгружается в область подкачки. В таблице символов ядра присутствует символ с именем `init_task`, адрес которого равняется адресу структуры `task_struct` процесса swapper.

Таблица символов ядра состоит из подряд идущих записей размера 8 байт. Каждая запись состоит из двух адресов: адрес структуры, на которую ссылается символ, (старшие 4 байт) и адрес строковой константы — имя символа (младшие 4 байт). После таблицы символов в памяти располагаются строковые константы символов ядра. Таким образом, чтобы получить адрес символа `init_task`, необходимо найти строковую константу `init_task\0` в начале образа памяти (её адрес будет равняться смещение от начала файла). После чего найти соответствующую запись в таблице символов и извлечь из неё адрес символа (старшие 4 байт).

Получив адрес структуры `task_struct` процесса swapper, можно вычислить смеще-

ние поля comm внутри структуры, так как известно, что оно равняется `swapper\0`.

Для получения смещения поля `tasks` структуры `task_struct` может быть применён метод списков. Данный метод заключается в переборе смещения для головы списка (структуры `list_head`) и проверки связности и замкнутости полученного списка. Смещение для поля `tasks` перебирается в диапозоне от 0 до смещения поля `comm`. Так как структура `task_struct` содержит различные связные списки, такие как список дочерних процессов или процессов "братьев" (имеющих того же родителя), список максимальной длины определяет смещение поля `tasks`.

Осталось определить смещения полей `mm` в структуре `task_struct` и `pgd` в структуре `mm_struct`. Заметим, что во всех рассматриваемых в работе версиях ядра, вне зависимости от выбранной конфигурации, выполняется следующее: по нулевому смещению структуры `mm_struct` находится указатель на структуру `vm_area_struct`, а по нулевому смещению структуры `vm_area_struct` — указатель обратно на ту же структуру `mm_struct`. Проанализировав исходные коды структуры `task_struct`, были получены следующие возможные смещения поля `mm` относительно поля `tasks`:

- 16 байт в версиях ядер v2.6.11 – v2.6.26.8
- 0 байт в версиях v2.6.27 – v2.6.29.6
- 20 байт в версиях v2.6.30 – v2.6.37.6
- 0 или 20 байт в версиях v2.6.38 – v3.3

Перебор перечисленных смещений с проверкой описанного выше условия целостности определяет смещение поля `mm` в структуре `task_struct`.

Имея в распоряжении дескриптор процесса swapper и известное смещение поля `tasks` в структуре `task_struct`, можно получить список дескрипторов всех процессов в системе. Для каждого перебираемого смещение поля `pgd` в структуре `mm_struct`, можно из данного построить новый список. Каждый элемент `task` исходного списка заменить на значение `task->mm->pgd` или исключить элемент из списка, если тот не содержит адрес на структуру `mm_struct` в поле `mm` (например, процессы режима ядра). Для корректно определённого смещения поля `pgd`, новый список должен состоять по большей части из адресов, полученных алгоритмом нахождения потенциальных PGD и PDPT.

Таким образом, используя найденные смещения в структурах, можно для каждого файла дампа адресного пространства процесса (из предыдущего раздела) поставить в

соответствие имя процесса. Некоторые дампы могут не получить имени. Эти файлы будут являться дампами адресных пространств недавно завершенных процессов, а также процессов, потенциально скрытых руткитами, так как их дескрипторы не содержатся в списке дескрипторов активных процессов в структурах ядра Linux.

6 Тестирование

Для упрощения тестирование изложенных в работе алгоритмов использовались виртуальные машины различных конфигураций, работающие под управлением различных версий ОС GNU/Linux. Для проверки корректности восстановления адресного пространства процессов, в рамках работы была разработана программная реализация описанных алгоритмов.

Программа принимает на вход путь к файлу образа памяти и путь к файлу образа области подкачки. Для каждого найденного в образе памяти процесса создает отдельный дамп файл с именем процесса, если удалось получить имя, или с именем равным шестнадцатеричному представлению адреса связанного с процессом каталога страниц. Для каждого дамп файла создается файл индексов. Файл индексов определяет соотношение виртуального адреса и смещения соответствующей ему страницы в дамп файле.

Для тестирования корректности восстановления адресного пространства совместно с областью подкачки была также написана вспомогательная программа. Она выделяет несколько участков памяти, выровненных по границе в 4 Кбайт и по размеру в сумме заведомо больших объёма оперативной памяти. После выделения очередного участка памяти, вспомогательная программа заполняет его определённым образом: первые 4 Кбайт участка заполняет байтами со значением `0x00`, следующие 4 Кбайт байтами со значением 0x01 и так далее. После заполнения очередной области размера 4 Кбайт байтами со значением `0xFF`, процедура повторяется снова, начиная со значения 0x00. Таким образом, каждая страницы, относящаяся к такому участку памяти, будет заполнена байтами с одинаковыми значениями. Выделив и заполнив все участки памяти, запущенный экземпляр вспомогательной программы (процесс) переходит в режим ожидания пользовательского ввода и остаётся доступным в списке активных процессов ОС. Далее можно произвести снятие образов памяти и области подкачки, например, описанным в работе [3] способом.

После применения программной реализации описанных в работе алгоритмов к снятым образам, легко проверяется корректность восстановления всех страниц, соответ-

ствующих выделенным специальным образом участкам памяти в адресном пространстве процесса вспомогательной программы.

7 Заключение

В данной курсовой работы был описан алгоритм нахождения в образе памяти страничных структур архитектуры x86, не зависящих от версии работающей операционной системы GNU/Linux. Рассмотрен алгоритм получения виртуального адресного пространства каждого процесса из файла образа памяти совместно с файлом образа области подкачки, а также описаны преимущества использования совместного подхода. Были описаны сложности поиска структур ядра ОС GNU/Linux, связанных с управлением процессами в системе и предложены возможные решения этой задачи.

Рассмотренные в работе алгоритмы могут применяться для выявления в системе процессов, потенциально скрытых путём вмешательства вредоносных программ в логику работы ядра ОС. Полученное с помощью данных алгоритмов адресное пространство процессов является достаточно полным и может быть полезным для дальнейшего анализа и восстановления из него такой информации, как изображения, текстовые документы, исполняемые файлы, зашифрованные или запакованные вредоносные программы.

В рамках проведённых исследований была разработана программная реализация, доказывающая работоспособность и корректность методов решения поставленной задачи.

8 Дальнейшее развитие

Дальнейшим развитием темы работы может являться исследование способов извлечения из образа памяти работоспособных исполняемых файлов, а также рассмотрение различных средств снятия образов оперативной памяти и области подкачки, например DMA (Direct Memory Access) с помощью hardware-средств, таких как Firewire или PCI.

С развитием технологий виртуализации, всё чаще активная деятельность пользователей начинает происходить внутри виртуальной машины. Поэтому, для цифрового криминалистического анализа интересной становится задача определения в образе памяти областей, относящихся к гипервизору и виртуальной машине, а также задача восстановления адресного пространства процессов виртуальной машины.

В данной работе был проведён анализ операционной системы Linux, работающей на архитектуре x86, поэтому дальнейшей ход исследований может быть нацелен на адаптацию рассмотренных алгоритмов к системам, работающим на архитектурах x86_64 и ARM, а также на разработку устойчивых методов, не зависящих от версии ядра ОС.

Список литературы

[1] DP Bovet, M. Cesati Understanding the Linux kernel. 3rd ed. Sebastopol, CA: O'Reilly Media, Inc.; 2006.

[2] M. Burdach Physical memory forensics. USA: Black Hat; August 2006.

[3] I. Kollar Forensic RAM dump image analyser. Charles University in Prague; 2010.

[4] M. Gorman Understanding the Linux Virtual Memory Manager.

[5] K. Saur, J. B. Grizzard Locating x86 paging structures in memory images. Digital Investigation Journal; August 2010.