Статья Вредонос под наблюдением. Как работают сендбоксы и как их обойти

Один из способов детектировать малварь — запустить ее в «песочнице», то есть изолированной среде, где можно следить за поведением вредоноса. В этой статье мы посмотрим, как устроены сендбоксы, и изучим приемы уклонения от детекта — и широко освещенные в интернете, и принципиально новые, упоминания о которых не удалось найти ни в специализированной литературе, ни на просторах Сети.
Борьба вирусописателей и производителей антивирусов — это борьба технологий: с появлением новых методов защиты сразу же появляются методы противодействия. Поэтому многие вредоносные программы первым делом проверяют, не запущены ли они в виртуальной машине или песочнице. Если вредоносу кажется, что он выполняется не на реальном «железе», то он, скорее всего, завершит работу.

Технику обхода песочниц часто называют термином Sandbox Evasion. Эту технику используют многие кибергруппировки, например PlugX или Cozy Bear (APT29). Так, нашумевшая группировка Anunak (авторы известного трояна Carbanak) использовала встроенное в документ изображение, которое активирует полезную нагрузку, когда пользователь дважды щелкает по нему.

Проще всего понять, как работают механизмы обнаружения виртуальных сред, с использованием возможностей PowerShell и WMI.

Пример построения песочницы
Традиционный враг любого вируса — специально подготовленная виртуальная машина, в которой исследователь запускает образец вредоносного ПО. Иногда вирусные аналитики используют сложные автоматизированные системы для анализа объектов в изолированной среде на потоке с минимальным участием человека. Такого рода ПО или программно-аппаратные комплексы поставляют разные компании, в частности FireEye (Network/Email Security), McAfee (Advanced Threat Defense), Palo Alto (Wildfire), Check Point (SandBlast), Kaspersky (Anti-Targeted Attack Platform), Group-IB, TDS. Также существуют опенсорсные песочницы, например Cuckoo Sandbox.

В России этот сегмент рынка относительно молод и начал активно развиваться примерно в 2016 году. Технологии песочницы, как ты видишь, используют практически все топовые производители средств информационной безопасности.

Начнем с базы и кратко пробежимся по внутреннему устройству виртуальной машины (ВМ). Для начала запомни, что ВМ — это абстракция от физического железа (аппаратной составляющей). Этот тезис понадобится, чтобы понимать, откуда у нас,

собственно, возникли некоторые из приемов обнаружения изолированных сред.

Сердце ВМ — это гипервизор. Гипервизор сам по себе в некотором роде минимальная операционная система (микроядро или наноядро). Он предоставляет запущенным под его управлением операционным системам сервис виртуальной машины, то есть виртуализирует или эмулирует реальное (физическое) аппаратное обеспечение. Гипервизор также управляет этими виртуальными машинами, выделяет и освобождает ресурсы для них.

Схема работы виртуалки

Напомним некоторые особенности виртуальных машин. Во-первых, центральный процессор гипервизора никогда не виртуализируется, а распределяет свои ресурсы между виртуальными машинами. То есть гипервизор не может выделить, например, половинку от ядра для какой-либо ВМ. Во-вторых, аппаратные компоненты при виртуализации фактически представляют собой файлы и не могут обладать физическими характеристиками HDD, такими, например, как износ секторов диска и другие параметры из SMART.

Как правило, виртуальные машины внутри песочниц разворачиваются чистыми (без специализированных средств анализа вредоносных программ внутри ВМ). Весь анализ выполняется гипервизором, в противоположность «ручным» исследовательским лабораториям, где на виртуальную машину уже установлен весь джентельменский набор программ для отслеживания активности вредоноса (OllyDbg, IDA, Wireshark, RegShot, Mon, Ripper, Process Monitor/Explorer и прочие). В этих прогах каждый созданный вирусописателями модуль защиты или проверка на наличие виртуального окружения обходится правкой кода вируса в дизассемблированном виде. Ниже схематически изображен анализ в одной из коммерческих песочниц — данные взяты из патента на продукт. Название мы нарочно оставим за кадром, так как нам интересен не конкретный производитель, а сама идея и подход к решению задачи.

Работа песочницы

Большинство коммерческих систем динамического анализа объектов построены по схожей схеме. Кратко поясним идею алгоритма.

1. Файлы, поступающие на анализ, попадают в очередь на проверку.

2. Приложение, отвечающее за безопасность, вместе с анализатором выполняет предварительные операции: сигнатурную, эвристическую и иные проверки. Проверка объекта в ВМ сама по себе очень затратна по времени и ресурсам. В итоге определяется, нужно ли отправлять образец на анализ в песочницу, и конкретные параметры виртуальной машины (включая версию и разрядность ОС), а также метод анализа объекта.
3. Объект помещается в ВМ и исполняется. В ВМ нет никакого специализированного ПО — только стандартный набор программ обычного офисного сотрудника типичной компании; также имеется доступ в интернет (выявить сетевую активность вредоноса).
4. Все действия исследуемой программы внутри ВМ через гипервизор попадают в анализатор, который должен понять, происходит что-то вредоносное или нет, опираясь на вшитые в него шаблоны поведения.
5. Контекст и результаты анализа заносятся в базу данных для хранения, дальнейшего использования и обучения анализатора.
   Отсюда становится ясной основная фишка автоматизированных систем анализа объектов. Они действуют за областью виртуальной машины — на стороне гипервизора — и анализируют всю поступающую на него информацию. Именно поэтому вредоносной программе нецелесообразно искать специальные инструменты анализа на стороне ВМ, но мы в статье будем это делать, чтобы более широко охватить тему.

На основе каких гипервизоров строятся песочницы? Обычно это Open Source и коммерческие гипервизоры на основе KVM. У того и у другого подхода есть свои плюсы и минусы, которые мы свели в табличку.

Плюсы и минусы гипервизоров

Очень часто, чтобы скрыть, что программа работает в виртуальной среде, в гипервизорах с открытым исходным кодом используется метод под названием харденинг. В этом случае часть доступных для определения параметров (например, идентификаторы производителей устройств) пробрасывается в виртуальную машину или заглушается сгенерированными данными.

Исследуя программу в песочнице, эмулируют действия пользователя (движение мыши, запуск приложений, работа в интернет-браузере). Проверяют работу образца с правами администратора, а также сдвигают время, чтобы выявить специфические условия активации вредоносной составляющей (некоторые из них начинают действовать после определенного периода «сна») и логических бомб.

Определение работы в песочнице
Мы написали небольшую программку на Python с использованием PowerShell, оценивающую «реальность» машины, на которой она запущена. В зависимости от условий большинству использованных в программе методов достаточно прав пользователя, однако некоторым для достоверного результата нужны привилегии администратора. Сразу оговоримся, что тесты мы проводили на Windows-платформах — они популярнее всего, и их чаще заражают. Для простоты мы использовали запросы к WMI (Windows Management Instrumentation, инструментарий управления Windows, то есть технология для обращения к объектам в системе). Запросы к WMI для унификации и наглядности мы выполняли с использованием PowerShell.

Исследовав в общей сложности порядка 70 машин, мы выделили ряд методов, позволяющих выявить средства виртуализации. Условно все методы определения работы в виртуальной среде мы разделили на четыре уровня, начиная с простого поиска строковых признаков виртуализации и заканчивая определением физических параметров системы, от которых будет сложно избавиться на виртуальной машине.

Level 1 — поиск характерных строковых идентификаторов
Самое простое, что может сделать программа, запустившись на новой машине, — проверить, что это за компьютер. Если название модели (или производителя некоторых составляющих) содержит virt либо где-то присутствуют ссылки на известных вендоров технологий виртуализации, это уже первый тревожный звоночек.

Параметры производителя и модель компьютера (Manufacturer и Model) легко поддаются правке, поэтому их значения остаются на совести создателей песочниц. Вот как можно отправить запрос к WMI через командную строку:
Code:

```
wmic computersystem get model,manufacturer
```

А вот так выглядит запрос в PowerShell:
Code:

```
Get-WmiObject Win32_ComputerSystem | Select-Object -Property Model,Manufacturer
```

В результате обработки запроса мы получим следующий отчет — сравни первые три вывода команды для виртуальных машин с показаниями реальной:
Code:

```
Manufacturer: innotek GmbH
Model: VirtualBox

Manufacturer: Parallels Software International Inc.
Model: Parallels Virtual Platform

Manufacturer: VMware, Inc.
Model: VMware Virtual Platform

Manufacturer: ASUSTeK Computer Inc.
Model: K53SV (реальная машина. – Прим. авт.)
```

В официальной документации к гипервизору может встречаться указание на список доступных для подмены параметров. Так, для гипервизора VirtualBox можно поменять ряд параметров. Перечислим некоторые:

- DmiBIOSVendor (производитель BIOS'а);
- DmiBIOSVersion (его версия);
- DmiBIOSReleaseDate (и его дата релиза);
- DmiSystemVendor (производитель ОС);
- DmiSystemProduct (название ОС);
- DmiSystemVersion (версия ОС);
- DmiSystemSerial (серийный номер установки ОС);
- DmiSystemFamily (семейство ОС);
- Disk SerialNumber (серийник HDD);
- Disk FirmwareRevision (его номер прошивки);
- Disk ModelNumber (модель HDD).

DMI (Desktop Management Interface) — программный интерфейс (API), позволяющий программному обеспечению собирать данные о характеристиках аппаратуры компьютера.

Для начала можно проверить серийный номер BIOS. Это срабатывает часто, ~~поскольку при создании ВМ никого не волнует тонкая настройка базовой подсистемы ввода-вывода~~, поэтому, если нам удастся получить правдоподобные сведения, скорее всего, машина реальная. Попробуй набить данную команду в PowerShell или даже PowerShell ISE (кстати, дальше мы развлекаемся только на PowerShell и не понимаем, почему ты еще не вошел в консоль):
Code:

```
Get-WMIObject Win32_Bios
```

На реальной машине ты увидишь что-то вроде
Code:

```
SMBIOSBIOSVersion : K53SV.320
Manufacturer : American Megatrends Inc.
Name : BIOS Date: 11/11/11 14:59:25 Ver: 04.06.03
SerialNumber : B5N0AS16677319F
Version : ASUS - 6222004
(реальная машина)
```

А на виртуальной тебе могут дополнительно подсказать в поле SerialNumber, какая здесь принята среда виртуализации:
Code:

```
SBIOSVersion : 6.00
Manufacturer : Phoenix Technologies LTD
Name : Default System BIOS
SerialNumber : VMware-56 4d 0c d4 94 e7 56 70-45 0e 17 fa d8 6b a4 99
Version : PhoenixBIOS 4.0 Release 6.0
```

Если серийник BIOS недоступен или его длина превышает 25 символов (проверено на практике), это скажет тебе, что либо машина скрывает свои параметры, либо они были целенаправленно изменены.

Теперь обратим внимание на список запущенных процессов. Как мы помним, для коммерческих песочниц этот способ вряд ли окажется эффективен. К тому же попытка получить список процессов наверняка будет воспринята как подозрительная. Тем не менее для широты охвата опишем и этот подход.

Присутствие таких процессов, как vmtoolsd.exe (а также vboxtray, vboxservice, vmacthlp, kqemu), уже само по себе способно заставить задуматься. Добавь к этому список из утилит вроде sysinternals, перечисленных в начале статьи, wireshark.exe, утилит для отладки (ollydbg, x64dbg) — и сразу станет понятно, анализирует ли кто-то твою программу. Финишной прямой будет обнаружение «песочных» (с программной изоляцией исполнения) процессов sandboxierpcss, sandboxiedcomlaunch и таких прог, как Sandboxie, Shade, BufferZone.

Просто берешь и получаешь список процессов:
Code:

```
Get-Process
```

Не сомневаемся, что в списке запущенных процессов ты без труда найдешь наших подозреваемых:
Code:

```
Handles NPM(K) PM(K) WS(K) VM(M) CPU(s) Id ProcessName
------- ------ ----- ----- ----- ------ -- -----------
443 47 217408 201748 459 72,87 8472 AcroRd32
89 9 2744 1500 32 0,09 1784 AsLdrSrv
46 8 2668 516 56 0,09 2084 ATKOSD
98 13 5488 788 90 0,66 2384 ATKOSD2
369 13 24208 23660 67 8776 audiodg
...
91 6 1144 5228 53 1.63 944 vmacthlp
319 22 5720 17536 136 96.78 1260 vmtoolsd
327 22 4700 14360 94 149.00 1924 vmtoolsd
...
```

Интересно, а если программа определит, что она внутри песочницы, и запустит бесконечный цикл, который заморозит ВМ, сможет ли система песочницы достать артефакты?

Еще один интересный способ — проверить собственное имя исполняемой программы. Зачастую при анализе в песочницах образец вредоноса переименовывается на примерно следующее: sample.exe, virus.exe, runme.exe, artifact.exe — или задается имя, равное контрольной сумме образца. Думаем, нормальные люди редко переименовывают файлы перед запуском.

VMware предоставляет официальные документированные способы обнаружить виртуальные машины. Первый способ основан на использовании порта гипервизора 0x5658 (VX) и гипервизора магического 32-битного значения DWORD 0x564D5868, который обозначается VMXh. Этот метод очень давно известен и широко используется вредоносным ПО. Его легко отключить: перенастроить файл конфига %vmname%.vmx или с помощью дополнительной фильтрации. Однако этот бэкдор требуется для нормальной работы VMware Tools.

В процессорах Intel и AMD бит 31 регистра ECX (EAX= 1,) инструкции CPUID (EAX=40000000h) 0x1 зарезервирован как бит присутствия гипервизора (HV). Гипервизоры, чтобы указать свое присутствие в гостевой операционной системе, устанавливают этот бит в единицу, а во всех существующих и будущих физических процессорах этот бит устанавливается в ноль. Гостевые операционные системы могут просмотреть этот 31-й бит, чтобы определить, работают они внутри виртуальной машины или нет.

Корпорация Microsoft в своей официальной документации пишет следующее: «Прежде чем использовать какие-либо функции интерфейса гипервизора, программное обеспечение должно сначала определить, работает ли он внутри виртуальной среды. На платформах x64 ПО проверяет, что оно работает в виртуальной среде, выполняя инструкцию CPUID со входом (регистр EAX), равным 1. Когда

выполняется инструкция CPUID, код должен проверять бит 31 регистра ECX. Бит 31 — бит присутствующего гипервизора. Если этот бит установлен, то присутствует гипервизор, иначе ПО работает в виртуальной среде».

Если подтверждено присутствие бита HV, то хорошо знать, какой это тип HV. Intel и AMD также зарезервировали в инструкции CPUID адреса в диапазоне 0x40000000–0x400000FF. Гипервизоры могут использовать эти адреса, предоставляя интерфейс для передачи информации из гипервизора в гостевую операционную систему, запущенную внутри виртуальной машины. VMware определяет адрес 0x40000000 как информационный адрес CPUID гипервизора. Код, работающий на гипервизоре VMware, может протестировать этот информационный адрес CPUID для проверки подписи гипервизора. VMware хранит строку VMwareVMware в регистрах EBX, ECX, EDX инструкции CPUID 0x40000000.

Упомянутые методы CPUID используются во вредоносных программах Win32/Winwebsec. Исходные коды этой малвари легко найти на просторах интернета.

Использование техники CPUID вызовет подозрения у песочницы, как и поиск запущенных процессов. Технику обнаружения по флагам процессора мы использовали на ранних стадиях исследовательской работы, но после множества тестов решили от этого модуля отказаться из-за явной компрометации своего ПО и большого количества ложноположительных срабатываний.

Резюмируем: Level 1 — это поиск характерных признаков виртуальных машин из песочниц, в основном строковых параметров внутри ОС. Сюда можно отнести:

- серийные номера BIOS, HDD, CPU;
- MAC-адреса;
- имена платформы, машины, домена, процессов, файлов, ключей реестра;
- инструменты анализа объекта, находящиеся на самой ВМ (OllyDbg, Wireshark, RegShot, Mon, Ripper, Process Monitor/Explorer);
- обнаружение CPUID;
- измененное имя исполняемого файла.

Level 2 — интерактив и окружение системы

В этой части мы ищем признаки, указывающие, что машина — «рабочая лошадка», а не чистая виртуалка, запущенная только для анализа нашей программы. Помимо этого, мы дополнительно исследуем выделенные для ВМ мощности.

Первым делом интересно посмотреть, когда система была запущена. Любители консоли (это в последний раз, обещаю) используют такую команду:
Code:

```
wmic path Win32_OperatingSystem get LastBootUpTime
```

и получат такой вывод:
Code:

```
LastBootUpTime
20200505015539.735711+180
```

Немного попотев, можно на PowerShell составить команду вида
Code:

```
Get-WmiObject win32_operatingsystem | select csname, @{LABEL='LastBootUpTime';
EXPRESSION={$_.ConverttoDateTime($_.lastbootuptime)}}
```

И оформить результат в человекопонятном формате:
Code:

```
csname LastBootUpTime
------ --------------
Mycomp 04.05.2020 13:50:47
```

Если система работает меньше десяти минут (uptime), это может означать, что она была запущена целенаправленно для исследования нашего ПО (скорее всего, это ручной анализ) или из снимка виртуальной машины. Каждый раз при запуске ВМ из одного и того же снапшота значение времени работы ОС постоянно. Если виртуалка запущена из снимка в песочницах, в некоторых из них можно найти примерно одинаковое значение аптайма. Чтобы никого не скомпрометировать, скажем, что это значение было 2000 с ± 100 с. Также важно отслеживать количество дней, прошедших с последнего запуска машины, поскольку реальные компьютеры (имеются в виду корпоративные) включаются и выключаются в среднем раз в день. Если текущее время отличается от данной метки больше чем на месяц, такая машина подозрительна. Этот способ рассчитан на недосмотр разработчиков песочниц.

При наличии админских прав (заметим, что для максимального раскрытия возможностей исследуемых программ у большинства песочниц объекты запускаются с правами администратора) попробуем посмотреть количество интерактивных входов в систему. Для этого используем логи безопасности (если логирование событий входа в систему включено).

Выведем все события с EventId=7001 и 7002 (события входа и выхода из системы winlogon) за последние семь дней:
Code:

```
$logs = get-eventlog system -ComputerName yourCompName
-source Microsoft-Windows-Winlogon
-After (Get-Date).AddDays(-7);
$res = @();
ForEach ($log in $logs) {
if($log.instanceid -eq 7001) {
$type = "Logon"}
Elseif ($log.instanceid -eq 7002){
$type="Logoff"}
Else {Continue}
$res += New-Object PSObject
-Property @{Time = $log.TimeWritten;
"Event" = $type;
User = (New-Object System.Security.Principal.SecurityIdentifier
$Log.ReplacementStrings[1]).Translate(
[System.Security.Principal.NTAccount])}};
$res
```

Вывод будет примерно такой:
Code:

```
Event User Time
----- ---- ----
Logon yourCompName 02.05.2020 10:34:09
Logoff yourCompName 01.05.2020 21:58:19
Logon yourCompName 30.04.2020 17:13:53
Logoff yourCompName 29.04.2020 23:34:46
```

Когда мы видим, что в систему заходили достаточно часто (скажем, не менее 20 раз за пару месяцев) и регулярно (например, между недавними входами нет перерыва более месяца), есть все основания предполагать, что машина реальна.

Интересный параметр — количество общих сетевых папок (так называемые админские шары). Обычно в Windows их четыре — C$, D$ (при наличии раздела), ADMIN$ и IPC$. Как их посмотреть? Вот так:
Code:

```
Get-WmiObject -Class win32_share
```

Шары выводятся наглядно, да еще и с описанием — красота!
Code:

```
Name Path Description
---- ---- -----------
ADMIN$ C:\Windows Удаленный Admin
C$ C:\ Стандартный общий ресурс
D$ D:\ Стандартный общий ресурс
IPC$ Удаленный IPC
```

Также можно использовать аналогичную команду (однако она работает не на всех версиях PowerShell):
Code:

```
Get-SmbShare
```

Если расшаренных папок нет или не больше двух, вероятно, с машиной работал матерый пользователь и дорабатывал ее, чтобы ограничить неиспользуемые сетевые ресурсы. Естественно, это не стопроцентный признак виртуальной машины, но как одна из зацепок — очень даже.

Идем дальше. На всякий случай проверим такие простые показатели, как размер жесткого диска и оперативки. Сложно представить, что на реальной машине будет установлен винчестер объемом менее 100 Гбайт и 2 Гбайт оперативной памяти. Спросить у машины, сколько у нее гигабайтов оперативки, можно так:
Code:

```
$cs = get-wmiobject -class "Win32_ComputerSystem"
 [math]::Ceiling($cs.TotalPhysicalMemory / 1024 / 1024 / 1024)
```

В ответ она тебе выведет только число. Это удобно для быстрой прикидки, но, если ты дотошен, давай возьмем от команды все:
Code:

```
$cs = get-wmiobject -class "Win32_ComputerSystem"
$cs
```

Более подробный вывод команды, несомненно, порадует тебя:
Code:

```
Domain : WORKGROUP
Manufacturer : ASUSTeK Computer Inc.
Model : K53SV
Name : mysupuerdupercomp
PrimaryOwnerName : author
TotalPhysicalMemory : 8497418240 (8 Гбайт ОЗУ, реальная машина)
```

И, возможно, несколько огорчит в виртуалке:
Code:

```
Domain : WORKGROUP
Manufacturer : VMware, Inc.
Model : VMware Virtual Platform
Name : IE11WIN10_virtual
PrimaryOwnerName :
TotalPhysicalMemory : 1671300608 (< 2 Гбайт ОЗУ)
```

Получить доступные тома, их метку и размер просто:
Code:

```
Get-WmiObject -Class Win32_logicaldisk
```

Сравни вывод команды в реальной машине:
Code:

```
DeviceID : C:
DriveType : 3
ProviderName :
FreeSpace : 29173989376
Size : 256051458048 (256 Гбайт – норм)
VolumeName : OS
```

И виртуальных средах:
Code:

```
DeviceID : C:
DriveType : 3
ProviderName :
FreeSpace : 30396342272
Size : 42947571712 (42 Гбайт – прям как в симуляциях)
VolumeName : Windows 10_virtual
```

Еще один признак реальности системы — общее количество подключаемых USB-устройств. Посчитать их — как нечего делать:
Code:

```
$poo=Get-WmiObject -Class Win32_USBControllerDevice
$poo.count
```

Вывод команды — общее число подключаемых устройств к системе. Если за все время работы системы к ней подключалось не больше пары устройств, это подозрительно. На реальных машинах ожидаемо увидеть клавиатуру, мышь, пару флешек. На чистой, только что развернутой виртуалке количество зарегистрированных в системе USB-устройств обычно не превышает 15.

Косвенный признак — типы и количество сетевых адаптеров. Если в системе присутствует Wi-Fi-адаптер, с большой вероятностью можно считать, что перед нами реальная машина. Просмотреть сетевые адаптеры можно командой Get-WmiObject -Class Win32_NetworkAdapter.

Одна из важных характеристик в аппаратной конфигурации любого компьютера — это параметры процессора. Банальным подсчетом количества ядер (если меньше двух — это подозрительно) не стоит ограничиваться. Интересно еще и определить саму модель CPU, особенно если она присутствует в списке предустановленных моделей процессора в KVM. Если ввести указанную ниже команду в командной строке на гипервизоре KVM, то можно увидеть следующее (жирным выделены модели, которые встречались нам в некоторых песочницах):
Code:

```
## kvm -cpu ?
```

Тем самым мы получаем список поддерживаемых моделей процессоров для гостевых систем гипервизора:
Code:

```
x86 qemu64 QEMU Virtual CPU version 2.4.0
x86 phenom AMD Phenom(tm) 9550 Quad-Core Processor
x86 core2duo Intel(R) Core(TM)2 Duo CPU T7700 @ 2.40GHz
x86 kvm64 Common KVM processor
x86 qemu32 QEMU Virtual CPU version 2.4.0
x86 kvm32 Common 32-bit KVM processor
x86 coreduo Genuine Intel(R) CPU T2600 @ 2.16GHz
x86 486
x86 pentium
x86 pentium2
x86 pentium3
x86 athlon QEMU Virtual CPU version 2.4.0
x86 n270 Intel(R) Atom(TM) CPU N270 @ 1.60GHz
x86 Conroe Intel Celeron_4x0 (Conroe/Merom Class Core 2)
x86 Penryn Intel Core 2 Duo P9xxx (Penryn Class Core 2)
x86 Nehalem Intel Core i7 9xx (Nehalem Class Core i7)
x86 Westmere Westmere E56xx/L56xx/X56xx (Nehalem-C)
x86 SandyBridge Intel Xeon E312xx (Sandy Bridge)
x86 IvyBridge Intel Xeon E3-12xx v2 (Ivy Bridge)
x86 Haswell-noTSX Intel Core Processor (Haswell, no TSX)
x86 Haswell Intel Core Processor (Haswell)
x86 Broadwell-noTSX Intel Core Processor (Broadwell, no TSX)
x86 Broadwell Intel Core Processor (Broadwell)
x86 Opteron_G1 AMD Opteron 240 (Gen 1 Class Opteron)
x86 Opteron_G2 AMD Opteron 22xx (Gen 2 Class Opteron)
x86 Opteron_G3 AMD Opteron 23xx (Gen 3 Class Opteron)
x86 Opteron_G4 AMD Opteron 62xx class CPU
x86 Opteron_G5 AMD Opteron 63xx class CPU
x86 host KVM processor with all supported host features (only available in KVM mode)
```

Забавный факт: в некоторых системах динамического анализа можно заметить несовместимость физических компонентов, таких как модель материнской платы и процессора (чипсет разный), или, например, несоответствие количества ядер модели CPU.

Очень полезно узнать, установлены ли в системе антивирусные продукты (в некоторых песочницах и в их ВМ были замечены антивирусы). Нелишне проверить актуальность сигнатурных баз (поле productState, подробную расшифровку значений можно подглядеть тут). Для такого рода запроса нужно немного напрячься:
Code:

```
Get-WmiObject -Namespace "root\SecurityCenter2"
-Class AntiVirusProduct  -ComputerName yourCompName
```

Вывод команды достаточно подробен (мы оставили только ключевые поля). Не все программы прописываются так качественно, но теперь ты знаешь, где их искать:
Code:

```
__GENUS : 2
__CLASS : AntiVirusProduct
__SUPERCLASS :
__DYNASTY : AntiVirusProduct
…
displayName : Trend Micro Titanium Internet
…
pathToSignedProductExe : C:\Program Files\Trend Micro\Titanium\wschandler.exe
pathToSignedReportingExe : C:\Program Files\Trend
Micro\UniClient\UiFrmwrk\WSCStatusController.exe
productState : 266240
```

Существует пара очень хитрых способов проверить машину на виртуальность. Например, можно периодически запрашивать положение курсора мыши. В случае анализа без имитации пользовательских действий указатель мыши будет неподвижен значительную часть времени. Что необходимо моделировать движения мыши и нажатия клавиш в песочнице, писала «Лаборатория Касперского» (на основе файлов WikiLeaks) здесь.

Более хитроумный способ — подключиться к базе данных браузера и просмотреть историю серфинга, логины и пароли. Например, это можно сделать с помощью Python-модуля browserhistory.

Резюмируем: Level 2 — исследовать базовые аппаратные характеристики ВМ, наработку и интерактивность. Вот предлагаемые параметры для наблюдения:

- размер HDD, RAM, видеопамяти (об этом и некоторых других параметрах см. Level 4) и количество ядер процессора;
- последний запуск компьютера, аптайм, логи Windows (например, количество интерактивных входов с EventID == 4624), параллельно с этим по WMI можно проверить, когда устанавливалась ОС (командой $system = Get-WmiObject -Class Win32_OperatingSystem $system.Installdate);
- движения мыши;
- история браузера;
- разрешение монитора, есть ли в системе кулер, подключенные принтеры, батарея (помогает определить работу на реальном ноутбуке на раз-два: а когда ты видел виртуалку, запущенную на сервере, у которого есть аккумулятор? Для определения батареи используй команду Get-WmiObject Win32_Battery), адаптер Wi-Fi и так далее.

Level 3 — временные задержки

Анализируем время выполнения ПО и отслеживаем динамически изменяемое время,

исходя из предпосылки, что средства динамического анализа вносят задержки в выполнение программы.

В простейшем случае нужно запомнить время запуска программы (берется локальное время системы и время с внешнего NTP — для сопоставления). Одновременно с запуском программы необходимо вести собственный отсчет времени (например, количество тактов процессора), чтобы сравнить общее время работы программы. Для вычисления можно использовать следующую формулу:

время финиша работы/функции/модуля – время старта = количество времени внутреннего счетчика

Если изменилась дата или время (явный показатель работы в песочнице), это будет сразу заметно, и программа может перейти к выполнению невредоносного участка кода. Ниже показан отчет о времени работы ПО внутри одной из песочниц.

Фрагмент отчета из песочницы

Скриншот отчета говорит о том, что в системе динамического анализа подменялось как локальное время, так и сетевое на разное количество секунд. Время выполнения программы составило 33 с, а локальное время увеличилось на 6 с, при этом значение времени по NTP изменилось на 13 с.

Интересный момент: в рамках тестов ранней версии нашей программы на языке Python мы отправили исполняемый файл на анализ в VirusTotal. Спустя пару дней нам пришел отчет из Канады, где был запущен экземпляр программы:

Лог FTP-сервера
Code:

```
103.208.85.223 UNKNOWN wrtftpuser [03/Apr/2017:17:26:07 +0000] "STOR file03042020_223856.txt" 226 1025
BORROS-PC.lan UNKNOWN ftpuzer [03/Apr/2017:17:29:34 +0000] "RETR file03042020_223856.txt" 226 1025
```

Фрагмент отчета
Code:

```
Is Admin? > 1
System uptime of 113 seconds
UPTIME LESS THAN 10 MIN !!!!!
total log in: 9777
machine most likely REAL (more than 15 log in)
CAN'T GET VALUE OF HDD TEMPERATURE SENSOR
CAN'T GET VALUE OF CPU TEMPERATURE SENSOR
-----------------------------------------------
start time: 6.07619124777e-06
fin time: 59.5664095576
time difference (start - fin): 59.5664034814
TIME_DELAY: 60
-----------------------------------------------
start LOCAL time: ['2017', '04', '03', '22', '37', '50']
start NTP time: ['Mon', 'Apr', '03', '22', '55', '01', '2017']
fin LOCAL time: ['2017', '04', '03', '22', '38', '53']
fin NTP time: ['Mon', 'Apr', '03', '22', '56', '04', '2017']
-----------------------------------------------
LOCAL DATE NOT CHANGED
LOCAL TIME NOT CHANGED
LOCAL TIME NOT CHANGED (NTP)
-----------------------------------------------
W_CpuID 1 W_HDD_CPU_TEMP 1.4 W_VMware 1 W_UPTIME 0.2 (весовые коэффициенты для различных
проверок)
===============================================
W_TOTAL -> 0.9 (вероятность того, что машина виртуальна)
===============================================
```

Обычное выполнение всей программы занимает до 20 с, а в данном случае 60, что кричит нам об использовании инструментов анализа исполняемого файла. Помимо прочего, проверка системного журнала показала 9777 интерактивных входов в систему. Сама программа была запущена с правами администратора, а система запустилась 113 с назад. Все это сигнализирует о ручной проверке образца. На машине отработал даже метод CPUID!

В данном случае была запущена виртуальная машина, на которой исследователь тестировал образцы предположительно вредоносного ПО с использованием подготовленной ВМ. VirusTotal еще в 2017 году использовал собственную динамическую проверку объектов. О значениях температурных сенсоров мы поговорим позже — когда будем обсуждать Level 4.

Level 4 — аппаратные показатели (это физика, детка!)
Наиболее достоверным признаком, что перед нами реальная, а не виртуальная машина, можно считать возможность достучаться до железа. При харденинге некоторые производители песочниц задают имя процессора, не соответствующее его архитектуре. Аналогичная проблема касается и других параметров, которые не

полностью переносятся в виртуальную среду или скрываются либо вовсе не могут быть переданы в ВМ. Когда в системе только одно ядро, маловероятно, что это сравнительно современный компьютер — скорее пожилой системник, древний нетбук, планшет или виртуальная машина.

Для таких запросов о процессоре есть свой специальный класс в WMI:
Code:

```
Get-WmiObject -Class Win32_Processor
```

Он выводит подробную информацию о свойствах процессора:
Code:

```
…
Caption : Intel64 Family 6 Model 42 Stepping 7
…
Description : Intel64 Family 6 Model 42 Stepping 7
…
L2CacheSize : 256 (размер кеша L2)
L2CacheSpeed :
L3CacheSize : 6144 (размер кеша L3)
L3CacheSpeed : 0
…
Manufacturer : GenuineIntel
…
Name : Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz
NumberOfCores : 4
NumberOfLogicalProcessors : 8
…
ProcessorId : BFEBFBFF000206A7
…
```

Обрати внимание на размеры кешей L2 и L3. Вот тебе для сравнения размеры кешей в виртуальных машинах:
Code:

```
Vbox:
L2CacheSize: None
L3CacheSize: 0

VMware:
L2CacheSize: 0
L3CacheSize: 0

Paralles:
L2CacheSize: None
L3CacheSize: 0
```

~~И тут мне все стало яс...~~ Почему кеши второго и третьего уровня равны нулю внутри ВМ? Этим вопросом мы задались на ранних этапах разработки ПО. Посмотрим на картинку.

Проброс кешей в виртуалки 

Процессоры имеют кеш инструкций и кеш данных (это два подуровня кеша L1).
Кеш инструкций (L1i) используется для чтения инструкций исполняемой программы. Процессор и кеш L1 минимально необходимы для того, чтобы ОС и программы смогли работать на железе, в том числе и на ВМ.

Кеш L2 используется для больших блоков данных, по умолчанию находится на гипервизоре. Однако гипервизор позволяет настроить более производительную виртуальную машину, и в этом случае кеш, скорее всего, пробросится в ВМ.

Кеш L3 обычно представляет собой кеш-слой, который располагается поверх основной памяти. Таким образом, кеш L3 используется всеми процессорами материнской платы. Всякий раз, когда ЦП запрашивает данные из основной памяти, больший блок данных считывается в кеш L3. Если CPU требуется еще больше данных, сохраняемых последовательно после запрошенной ранее памяти, есть вероятность, что память может быть найдена в кеше L3, который находится на стороне гипервизора.

Тем не менее некоторые параметры CPU могут быть проброшены в виртуалку либо подменены, например Caption, DeviceID, Manufacturer, MaxClockSpeed, Name, SocketDesignation.

В результате гипервизор выдает виртуальной машине L1-кеш, но вот достучаться из-под виртуалки до кеша L2 можно не всегда, а до L3 — нельзя практически никогда. Сыграем на этом: если размеры кешей нулевые или None (не удалось получить значение), то перед нами виртуалка.

Много интересного о том, как виртуализируется аппаратная часть, можно прочитать в статье на сайте VMware.

Занятно, но некоторые ИБ-вендоры уже пофиксили баг с несуществующим кулером либо тщательно мониторят запрос на его наличие, хотя и на реальных машинах он корректно обрабатывается далеко не всегда. Чтобы получить информацию о кулерах, можно воспользоваться следующим скриптом PowerShell:
Code:

```
$colItems = Get-WmiObject Win32_Fan -Namespace "root\cimv2"
foreach ($objItem in $colItems)
{
  "Active Cooling: " + $objItem.ActiveCooling
  "Availability: " + $objItem.Availability
  "Device ID: " + $objItem.DeviceID
  "Name: " + $objItem.Name
  "Status Information: " + $objItem.StatusInfo
}
```

Вывод скрипта примерно следующий (если кулер таки нашелся):
Code:

```
Active Cooling : True
Availability : 3
Device ID : root\cimv2 1
Name : Cooling Device
Status Information : 2
```

Неплохая идея — просмотреть доступные звуковые устройства (класс Win32_SoundDevice). На простеньких виртуалках данный запрос обычно ничего не возвращает. Также советуем тебе посмотреть доступные реальные принтеры через класс Win32_Printer. Если в системе зарегистрированы используемые принтеры, кроме дефолтных, возможно, ты имеешь дело с реальным компьютером.

Следующие несколько параметров связаны с видеоконтроллером и экраном:
* разрешение экрана;
* количество и тип видеоконтроллеров;
* объем видеопамяти.

Не теряя ни секунды, запрашивай разрешение экрана:
Code:

```
Get-WmiObject -Class Win32_DesktopMonitor |
Select-Object ScreenWidth,ScreenHeight
```

В реальной машине оно выводится без проблем:
Code:

```
ScreenWidth ScreenHeight
----------- ------------
1366 768
```

На ВМ данные поля, как правило, остаются пустыми или с низким расширением. Что там с видеоконтроллерами?
Code:

```
Get-WmiObject -Class Win32_VideoController
```

На реальном ноуте у нас их аж два — встроенный и дискретный:
Code:

```
AdapterCompatibility : NVIDIA
AdapterDACType : Integrated RAMDAC
AdapterRAM : 1073741824
Caption : NVIDIA GeForce GT 540M
VideoProcessor : GeForce GT 540M

AdapterCompatibility : Intel Corporation
AdapterDACType : Internal
AdapterRAM : 2210398208
Caption : Intel(R) HD Graphics 3000
VideoModeDescription : 1366 x 768 x 4294967296 colors
VideoProcessor : Intel(R) HD Graphics Family
```

Для сравнения — результат выполнения этого скрипта в ВМ с VDI инфраструктуры (перечислены значимые строки):
Code:

```
AdapterCompatibility : Citrix Systems Inc.
AdapterDACType : Virtual RAMDAC
AdapterRAM : 0
Caption : Citrix Display Only Adapter
VideoModeDescription : Цвета: 1440 x 900 x 4294967296
VideoProcessor : Citrix Virtual Display Adapter Chip

AdapterCompatibility : Citrix Systems Inc.
**AdapterDACType : **
**AdapterRAM : **
Caption : Citrix Indirect Display Adapter
**VideoModeDescription : **
VideoProcessor :

AdapterCompatibility : VMware, Inc.
AdapterDACType : n/a
AdapterRAM : 0
Caption : VMware SVGA 3D
VideoModeDescription : Цвета: 1440 x 900 x 4294967296
VideoProcessor : VMware Virtual SVGA 3D Graphics Adapter
```

Подробности — после парочки строчек кода. Если нужно сразу получить размер видеопамяти в человеческом виде, можно воспользоваться следующей командой PowerShell:
Code:

```
Get-WmiObject Win32_VideoController |
select name, AdapterRAM,@{Expression={$_.adapterram/1MB};label="MB"}
```

Память есть, теперь в мегабайтах!
Code:

```
name AdapterRAM MB
---- ---------- --
NVIDIA GeForce GT 540M 1073741824 1024
Intel(R) HD Graphics 3000 2210398208 2108
```

На виртуалке видеопамять имеет какое-то свое, сакральное значение — либо оно нулевое, либо его вовсе не достать, либо его значение небольшое.

В виртуальных машинах либо ты не сможешь узнать разрешение экрана, либо может использоваться очень странное разрешение вроде 800 на 600. То же касается и количества видеоконтроллеров. Если параметр AdapterDACType (Digital-to-Analog Converter) не содержит в названии строк вида Internal или Integrated либо вообще пустой, это очень подозрительно. Не менее подозрителен маленький объем видеопамяти, например меньше 256 Мбайт (в нашей задаче размер имеет значение). Невозможность получить этот параметр характерна для виртуальных сред.

Интересно дела обстоят с виртуальными видеоконтроллерами на примере Citrix или других устройств с VDI — у них размер видеопамяти равен нулю, но такие девайсы можно фильтровать по названию вендора. Плюс к этому виртуальный видеоадаптер сам по себе в единственном числе не существует, в системе должен присутствовать еще один тип Internal или Integrated.

Много интересных данных можно извлечь при наличии прав администратора. Например, есть шанс достучаться до показаний температурных сенсоров реальной машины, что невозможно в виртуальной (если они не проброшены или не симулируются). Если получено валидное значение температуры (ориентировочно от 1 до 150 °C), это может говорить, что комп реальный. Приведем пример проверки температуры CPU:
Code:

```
$CPUt = Get-WmiObject MSAcpi_ThermalZoneTemperature -Namespace "root/wmi"
$currentTempKelvin = $CPUt.CurrentTemperature/ 10
$currentTempCelsius = $currentTempKelvin - 273.15

$currentTempCelsius
```

Если все пройдет успешно, ты узнаешь температуру проца в градусах Цельсия. Если не повезет — получишь значение -273.15, то есть значение абсолютного нуля и показатель, что достучаться до термодатчика стандартным путем нельзя.

Можно самостоятельно проверить свою виртуальную машину с помощью таких программ, как HWiNFO или AIDA64, и убедиться, что для виртуальной среды значения температуры будут недоступны (так же как ряд характеристик процессора и жесткого диска!). Однако в случае с Parallels значения датчиков температуры и вольтажа все же пробрасываются на виртуальные машины.

Проброс температуры и вольтажа проца в Parallels

И напоследок — самое сладкое (предупреждаем, что данный способ пока еще не опробован в боевых условиях). Проверим производительность графики с помощью WebGL и Canvas 3D. Легче всего это сделать, используя готовые онлайн-тесты, например GlowScript или Wirple. Мы остановились на втором варианте, поскольку он позволяет проводить блиц-стресс-тест производительности. Результаты смотри в таблице ниже.

Тест производительности GPU

Сравнивая полученные значения FPS между реальными и виртуальными машинами, мы пришли к выводу, что если итоговый результат по четырем тестам меньше 200 FPS, то перед нами виртуалка.

С FPS дело обстоит так: в виртуальных средах этот показатель на порядки ниже, чем на реальных машинах. В наших тестах производительность ВМ была на уровне 1–50 FPS, в то время как на реальном более-менее современном железе значение было порядка 1000–1500 FPS.

Заключение

Как можно заметить, не все методы позволяют со стопроцентной уверенностью определить, что программа запущена в виртуальной среде, однако использование нескольких разноуровневых методов существенно повышает вероятность получить правильный результат.

Нашей задачей было показать основные идеи и самые приметные характеристики ВМ. На Black Hat в 2015 году в качестве примера использовались проверки на количество логических процессоров и общий объем памяти, а также поиск VMware в названиях производителей и именах сетевого адаптера, в серийном номере BIOS. Также применялся поиск процесса vmtoolsd.exe.

Что касается методов, связанных с проверкой атрибутов устройств, часть из них может и не отличить реальную машину от виртуальной (если некоторые модели устройств не предоставляют данные через описанную функциональность или используются тонкие настройки и ограничения систем). Однако если применить сразу несколько методов, нацеленных на разные атрибуты, то с помощью весовых коэффициентов можно определить вероятность того, что машина виртуальна. На практике использование комбинации всех методов дает верный результат в 95% случаев.

Bug Bounty по песочницам еще не существует. Мы пытались сообщить нескольким вендорам, включая иностранных, о пробелах в их продуктах. Реакция производителей была разной — от насмешки (типа у нас настолько крутая система, что такого не может быть) до полного игнора, как только мы направляли доказательства наших слов. Именно поэтому мы решили опубликовать эту статью.

В итоге цикл «мы вам виртуалку — а мы вам метод ее обнаружения» может продолжаться до потери пульса. Тем не менее мер противодействия некоторым из представленных методов в коммерческих продуктах до сих пор нет.

Если ты заинтересовался темой: многие инструменты для детектирования песочниц можно найти в свободном доступе. Среди проектов на GitHub следует отметить InviZzzible от Check Point или агрегацию проектов в awesome-sandbox-evasion.

Если у тебя есть идеи, какие еще методы можно попробовать для определения виртуалок, или ты получил интересные результаты после выполнения команд из статьи — напиши нам, мы с удовольствием обсудим твои мысли!

Авторы: @ Boris Razor и @ Alex Mess
хакер.ру