Positive Hack Days 2017

Исследуем миллиарды состояний в программе как профи. Как подготовить свою собственную быструю и масштабируемую утилиту для анализа безопасности на базе ДБИ.

Максим Шудрак

PhD, Security Researcher

IBM Research Israel

Haifa Labs

Часть 0. Мотивация или зачем нужен этот доклад.

В 2012 году Дмитрий Евдокимов проводил доклад на PHD посвященной инструментации программного кода, приведя обширный обзор существующих технологий в области инструментации, начиная от исполняемого кода, заканчивая байт-кодом, кратко затрагивая в том числе и динамическую бинарную инструментацию (ДБИ). В данном докладе я бы хотел сконцетрироваться эксклюзивно на ДБИ, углубиться в эту тему, показать те обширные возможности, которые даёт данная технология на «живых» примерах из практики, а также рассмотреть типичные проблемы её применения в «индустриальных» задачах.

Если говорить в целом, то на мой взгляд эта тема освещена сообществом недостаточно. Есть ряд докладов на международных конференциях (например: [1][2][3][4]), описывающих различные решения на базе ДБИ, однако не дающих представления о всех возможностях этой технологии и тех сложностях с которыми предстоит столкнуться при работе с ней. При этом существующая в Интернете документация ограничивается лишь простыми примерами (оценка покрытия кода, подсчет количества инструкций, обработка выделения или освобождения памяти), тогда как реальные задачи по анализу реальных приложений требуют значительно больше навыков и понимания того, как работает ДБИ в целом (особенно это касается вопросов производительности и надежности).

Часть 1. Введение и область применения

На сегодняшний день мы имеем несколько стабильных и многофункциональных ДБИ фреймворков для наиболее популярных ОС (Windows, Android, Mac OS, Linux), поддерживаемых крупными корпорациями и академическим сообществом. Под динамической бинарной инструментацией принято понимать метод анализа программного кода в процессе его исполнения с помощью вставки инструментирующего кода. В данном докладе мы сконцентрируемся на наиболее популярном фреймворке Intel PIN.

Общая архитектуру ДБИ можно представлена на рисунке 1. Согласно рисунку 1, первый этап выглядит как классический DLL-injection, приложение запускается в suspended состоянии, производится инжектирование базовой библиотеки фреймворка, в которую затем передается управление. На следующем этапе, код библиотеки выполняет инициализацию среды исполнения, подгружает необходимые для её функционирования сторонние модули, а затем загружает пользовательскую DLL для начала инструментации. После этого на этапах 4-6, начинает выполняться последовательная трансляция каждого базового блока (последовательность инструкций без условных или безусловных переходов) в промежуточное представление (intermediate representation, IR). В зависимости от нужд пользовательской библиотеки, фреймворк производит встраивание инструментирующего кода (в нашем примере обозначены серым), а затем его обратную трансляцию в исполняемый код для исполнения в так называемом «кодовом кэше». Затем производится вычисление адреса следующего исполняемого базового блока, трансляция, инструментация, обратная трансляция, исполнение и так далее, до тех пор, пока не будет выполнена вся программа.

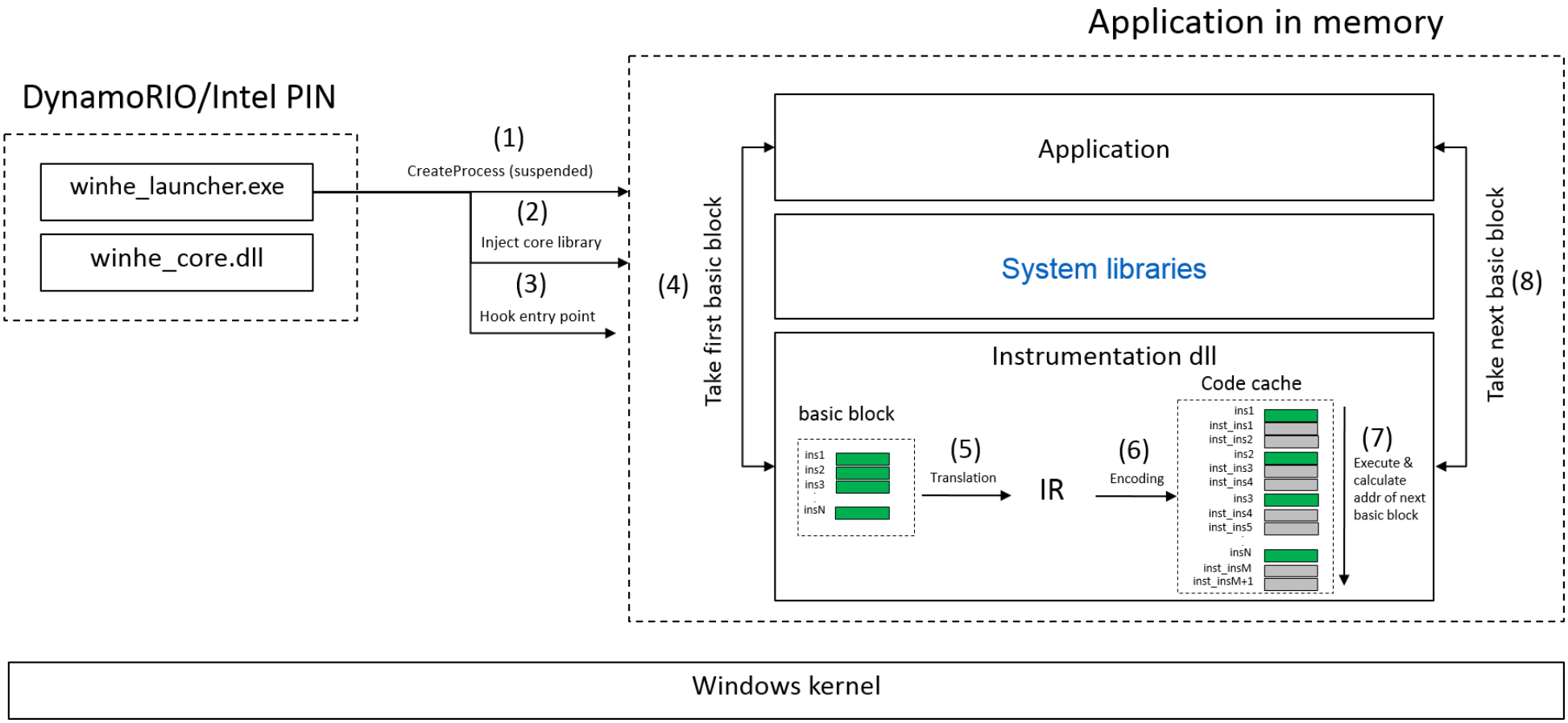

Рисунок 1. Архитекура ДБИ и пошаговое описание процедуры инструментации.

Несмотря на то, что общая идея инструментации выглядит относительно тривиально, корректная и эффективная (с точки зрения надежности, «прозрачности» и производительности) реализация такого подхода представляет собой сложнейшую инженерную задачу и подробно рассматривается в данных работах [5-7].

На сегодняшний день ДБИ успешно применяется в следующих областях:

- Поиск ошибок и уязвимостей в исполняемом коде, особенно в условиях отсутствия исходного кода [8]. Решения на базе ДБИ успешно интегрируется в SDL [9], используются для поддержки фаззинга [10], а также для автоматизированного обнаружения факта эксплуатации уязвимостей в программе [11-12].
- Анализ вредоносного кода [13]. ДБИ позволяет получить список библиотечных и системных вызовов [14] и может использоваться для автоматической распаковки [3-4]. Несмотря на то, что инструментируемое приложение может достаточно легко обнаружить факт инструментации, как было показано в данных докладах [15-16].
- Реверс-инжиниринг. ДБИ успешно применяется для выполнения анализа помеченных данных [17], анализа и построения графа потока управления программы [18], а также отладки [19].
- Во многих других областях не связанных напрямую с безопасностью, но связанных с анализом программного обеспечения (анализ производительности, поиск утечек памяти, оптимизация и т.д.).

Часть 2. Особенности применения ДБИ

Современные ДБИ – фреймворки имеют развитую API и позволяет выполнять инструментацию с различным уровнем гранулярности - на уровне каждой инструкции, базового блока, функции, трассы, модуля или определенного события связанного с программой (например выход программы или создание дочернего процесса). Используя этот богатый API, автором был реализована утилита для автоматизированного поиска ошибок типа переполнения буфера в исполняемом коде, описываемая ниже.

WinHeap Explorer эта утилита для Windows, позволяющая выполнять детектирования ошибок типа переполнения кучи и use-after-free в ходе исполнения программы. WinHeap Explorer построен на базе Intel PIN. Ценность этой утилиты заключается в том, что в отличии от уже существующих систем (DrMemory/Valgrind) существует возможность выполнять «частичную»инструментацию и тем самым снижать накладные расходы, вносимые инструментацией (в среднем от 24% до 71% в зависимости от приложения как показано на рисунке 2).

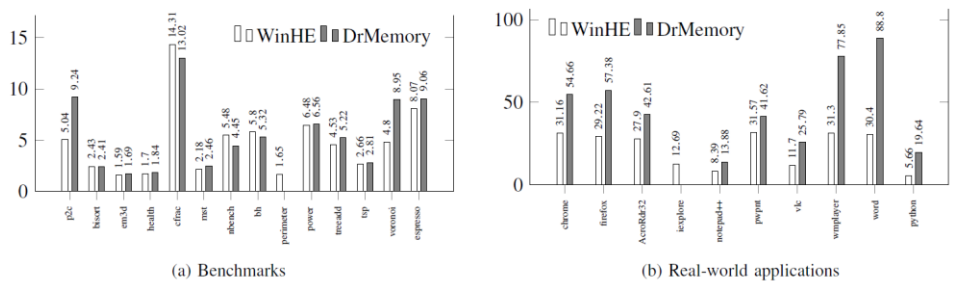

Рисунок 2. Сравнение производительности WinHeap Explorer и DrMemory

Общая идея данной утилиты заключается в реализации проверки каждой попытки доступа к куче на предмет выхода за границы выделенного блока памяти. Для этого до и после каждого выделяемого блока памяти устанавливаются специальные redzones. Если программа выполняет чтение или запись по адресу где находится redzone, WinHeap Explorer сигнализирует об ошибке. Звучит достаточно просто, однако на практике существует ряд серьезных сложностей.

В рамках утилиты выполняется инструментация на уровне каждой инструкции, библиотечного вызова выделения памяти и загрузки нового модуля. Также выполняется инструменация библиотечных вызовов, которые выполняют выделение/освобождение памяти в куче. Цель инструментации реализовать подход описанный выше (установить/снять redzones для каждого блока памяти в куче).

Затем мы выполняем инструментацию каждой выполняемой программой инструкции, которая осуществляет доступ к выделенной памяти. Если хотя бы один байт, к которому осуществляется доступ, принадлежит redzone, WinHeap Explorer записывает информацию об ошибке в лог в следующем формате.

```
[HEAP OVERFLOW] at 0x496f43, writing out of bound to 0x37110f; heap has been allocated at 0x495f13, size = 0x300, flags = 0
```

В ходе реализации данной идеи мне пришлось столкнуться с двумя серьезными проблемами:

1) Механизм выделения памяти в Windows представляет собой сложный недокументированный процесс, который распределен между несколькими системными библиотеками и реализован на разных уровнях WinAPI, иногда без официальной документации и с десятком недокументированных флагов, значительно меняющих механизм управления памятью.
2) Количество выделяемых/освобождаемых блоков в современных программах может достигать миллионов, а количество попыток доступа к ним миллиардов. В такой ситуации, необходимо как то хранить информацию об установленных redzones и эффективно предоставлять к ней доступ в случае чтения или записи.

Для решения первой проблемы я выполнил детальный reverse-engineering системных библиотек Windows, отвечающих за выделение памяти в куче и пришел к выводу, что нам нет необходимости выполнять инструментацию высокоуровневых WinAPI вызовов, достаточно выполнять инструментацию вызовов на уровне ntdll.dll (таблица 1), так как в любом случае, высокоуровневые вызовы используют `RtlAllocateHeap`, `RtlReAllocateHeap`, `RtlFreeHeap` для выделения памяти.

В свою очередь, для решения второй проблемы, в первой итерации, была выполнена реализация «в лоб»; информация о redzones и выделенных блоках памяти хранилась в хеш-таблице. К сожалению в рамках DBI традиционные в Computer Science структуры данных, которые предоставляют высокую скорость доступа, все равно недостаточно эффективны, вносят огромные накладные расходы и не могут использоваться для инструментации реальных приложений (эксперименты показали, что например на запуск Microsoft Word уходит в среднем 17 минут при таком подходе, не говоря уже о 500 кратном увеличении потребляемой оперативной памяти).

Таблица 1. Управления памятью в Windows.

Функция	**Имя библиотеки**	**API – функция**	**Соответствующая низкоуровневая API - функция**
Выделение памяти	kernel32.dll	HeapAlloc GlobalAlloc LocalAlloc	RtlAllocateHeap
	msvcr*.dll	malloc calloc operator new []	
	ole32.dll	CoTaskMemAlloc	
Реалокация памяти	kernel32.dll	HeapReAlloc GlobalReAlloc LocalReAlloc	RtlReAllocateHeap
	msvcr*.dll	realloc	
	ole32.dll	CoTaskMemrealloc	
Освобождение памяти	kernel32.dll	HeapFree GlobalFree LocalFree	RtlFreeHeap
	msvcr*.dll	free operator delete []	
	ole32.dll	CoTaskMemFree	

Для решения этой проблемы было принято решение использовать «теневую память». Теневая память это специальная область в виртуальном адресном пространстве исследуемой программе, используемая для отслеживания и хранения метаинформации о других блоках памяти в ходе исполнения этой программы. Рассмотрим в качестве примера, как этот подход используется для хранения информации об одном блоке памяти на рисунке 3.

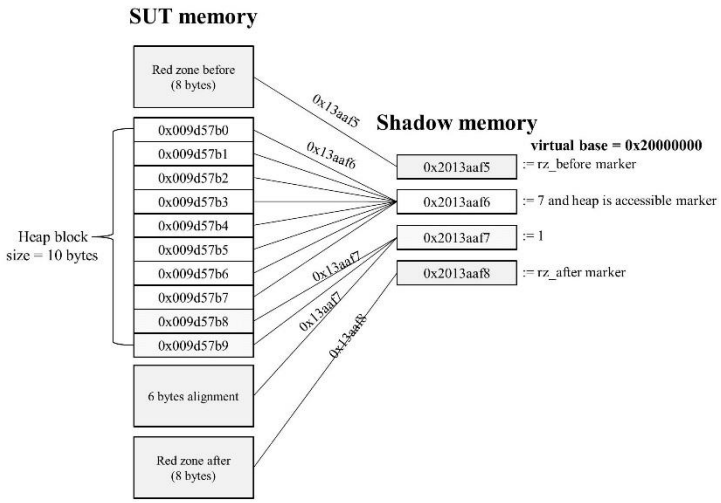

Рисунок 3. Пример использования теневой памяти для хранения информации об одном базовом блоке. SUT – software under test.

Так как память в куче выделяется со смещением в 8 байт, мы можем описать каждые 8 байт выделенные в кучи одним байтом в теневой памяти, как показано на рисунке 3. Таким образом, все что нам нужно чтобы получить информацию о каком либо адресе программы это разделить его на 8 и добавить базовый виртуальный адрес по которому была выделена теневая память. Такой подход позволяет значительно снизить накладные расходы, требуемые для инструментации одной инструкции.

Для хранения информации о выделяемом блоке памяти необходимо выполнить следующие шаги:

1) Получить размер выделенного блока памяти.
2) Последовательно разделить адрес каждого выделенного байта памяти на 8 и сохранить остаток от деления (плюс флаг `heap_is_accessible`) в соответсвующем теневом байте.
3) Установить redzones до и после выделенного блока памяти, пометив соответствующие байты в теневой памяти как `redzone_before` и `redzone_after`.
4) Обратная операция выполняется если программа освобождает блок памяти (все флаги в теневой памяти сбрасываются, блок памяти помечается как `heap_is_freed`).

Теперь если инструкция выполняет доступ к какому либо байту в куче, мы можем легко проверить не принадлежит ли он redzone или освобожденному блоку памяти (разделить на 8 и добавить базовый виртуальный адрес по которому выделена теневая память). Такой подход позволяет значительно снизить накладные расходы (в 40-65 раз) на инструментацию.

В ходе реализации WinHeap Explorer я решил пойти дальше и добавить возможность выполнять частичную инструментацию, только тех участков исполняемого кода, которые представляют наибольший интерес для исследователя, тем самым снизив накладные расходы ещё больше. Это может быть полезно если утилита интегрируется в фаззинг, где производительность тестирования играет далеко не последнюю роль. Например в тех случаях, когда исследователь не хочет выполнять инструментацию кода системных библиотек (рисунок 4), а сконцетрироваться на поиске ошибок в приложении.

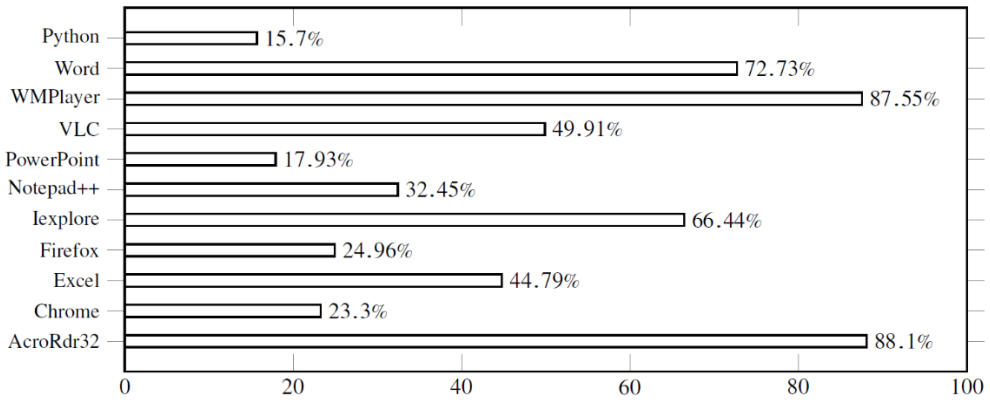

Рисунок 4. Соотношение числа выполненных инструкций в системных библиотеках к числу инструкций выполненных в исследуемом приложении. Данные были собраны с помощью специально реализованного ДБИ плагина для Intel Pin.

Однако не следует забывать, что уязвимость может возникнуть в процессе взаимодействия пользовательского кода и системной библиотеке. Например при вызове `memcpy` переполнение будет осуществлено в коде системной библиотеке (рисунок 5).

```
1   int some_function() {
2   <function prolog is skipped>
3   FILE *f = fopen("some_file.txt", "r");
4   004112E9  push      offset string "r"
5   004112EE  push      offset string "some_file.txt"
6   004112F3  call      dword ptr fopen
7   004112F9  add       esp,8
8   004112FC  mov       dword ptr [f],eax
9   char *buf_a = (char *)malloc(0x300);
10  004112FF  push      300h
11  00411304  call      dword ptr malloc
12  0041130A  add       esp,4
13  0041130D  mov       dword ptr [buf_a],eax
14  char *buf_b = (char *)malloc(0x30);
15  00411310  push      30h
16  00411312  call      dword ptr malloc
17  00411318  add       esp,4
18  0041131B  mov       dword ptr [buf_b],eax
19  fgets(buf_a, 0x100, f);
20  0041131E  mov       eax,dword ptr [f]
21  00411321  push      eax
22  00411322  push      100h
23  00411327  mov       ecx,dword ptr [buf_a]
24  0041132A  push      ecx
25  0041132B  call      dword ptr fgets
26  00411331  add       esp,0Ch
27  memcpy(buf_b, buf_a, 0x100);
28  00411334  push      100h
29  00411339  mov       eax,dword ptr [buf_a]
30  0041133C  push      eax
31  0041133D  mov       ecx,dword ptr [buf_b]
32  00411340  push      ecx
33  00411341  call      memcpy
34  631DC970  <function prolog is skipped>
35  631DC9C7  rep movs dword ptr [edi],dword ptr [esi]
36  ;edi -> buf_b, esi -> buf_a
37  631DC9XX  <function epilog is skipped>
38  00411346  add       esp,0Ch
39  00411349  xor       eax,eax
40  }
```

Рисунок 5. Пример переполнения кучи возникающих в процессе взаимодействия с системной DLL. Переполнение возникает по адресу `0x631dc9c7` в библиотеке msvcr100.dll, когда инструкция `rep movs` записывает за границы буфера `buf_b`.

Основываясь на этой идее, был реализован скрипт на базе IDAPython (доступен в репозитории WinHeap Explorer), позволяющих включить только «интересные» с точки зрения анализа системные процедуры. За основу был взят список опасных и нежелательных функций из Secure Development Lifecycle. Banned Function Calls [20], однако пользователь может включить в этот список и другие функции.

Схожий подход используется и в случае пользовательских библиотек, отдельный скрипт позволяет выделить процедуры в которых производится вызов потенциально опасных и нежелательных функций из списка Microsoft. Общий алгоритм работы данного скрипта представлен в листинге 1.

```python
def instrument_subcalls(subcall, current_depth, list_of_routines):
    global max_depth
    list_to_return = list()
    if routine in list_of_routines:
        return null
    if current_depth >= max_depth:
        return null
    list_of_subcalls = get_list_of_subcalls(subcall)
    for subcall in list_of_subcalls:
        list_to_return.append(instrument_subcalls(subcall, current_depth + 1, list_of_routines)

    return list_to_return


max_depth := <user specified value>
list_of_routines = get_list_of_interesting_routines()
final_list = list()
for routine in list_of_routines
    list_of_subcalls = get_list_of_subcalls(routine)
for subcall in list_of_subcalls:
    final_list.append(instrument_subcalls(subcall, 0, list_of_routines))
final_list.append(get_list_of_callers(depth))
```

Листинг 1. Инструментация пользовательских библиотек. Реализация на Python.

Таким образом общая архитектура решения и пример интеграции утилиты в процесс тестирования может быть обобщено представлен на рисунке 6.

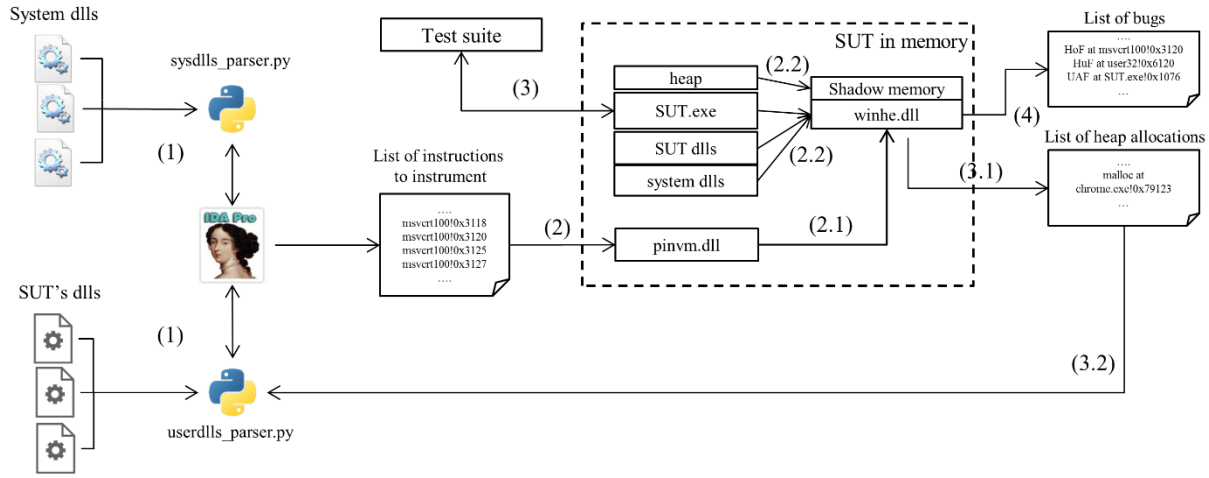

Рисунок 6. Пример интеграции WinHeap Explorer в процесс тестирования. IDAPython скрипты используются для уменьшения количества инструментируемого кода. SUT – software under test. Test suite – любой фреймворк для тестирования, в том числе фаззер.

Часть 3. Пример работы

Для начала инструментации с помощью WinHeap Explorer, достаточно выполнить следующую команду:

```
pin.exe -t winhe.dll -d [path] [app_name]
Аргументы:
-d [path] - путь к файлу с адресами инструкций для частичной инструментации, полученный в результате работы IDAPython скриптов.
-redzones_size [bytes #] - размер redzones.
-o <output path> - файл для вывода результов анализа.
```

Выше было отмечено, что специальный IDAPython скрипт используется для предварительного поиска функций в системных библиотек, которые также

необходимо включить в инструментацию. Для запуска поиска этих функций достаточно выполнить следующую команду:

```
sysdlls_parser.py [path to system dll]
user_dlls_parser.py [path to user dll]
```

Скрипт автоматически сгенерирует список инструкций из процедур, которые необходимо включить в инструментацию (необходимо иметь IDA с IDAPython).
Рассмотрим несколько примеров (*в презентации данный список будет расширен, показано DEMO, а также будет приведен детальный разбор того, как WinHeap Explorer детектирует уязвимость*), того как WinHeap Explorer выполняет детектирование переполнения буфера и use-after-free:

- Httpdx 1.5.4. Remote Heap Overflow. В результате некорректной обработки длины строки, уязвимый http сервер записывает `0x410` (используя `memcpy`) байт в буфер размером `0x400` байт. WinHeap выполняет обнаружение переполнения буфера в msvcr100.dll по смещению в библиотеке `0x39b60`.
- Microsoft Office Excel 2010 (CVE 2015-2523) BIFFRecord Use-After-Free. Приложение использует блок памяти, который был ранее освобожден в функции `AVrfpRtlFreeHeap` в библиотеке vfbasics.dll. WinHeap Explorer выполняет обнаружение use-after-free по адресу `0x30037cc5` в excel.exe.
- gdi32.dll (CVE 2016-0170) Heap-based Buffer Overflow in ExtExcape(). WinAPI функция `ExtEscape` из библиотеки gdi32.dll некорректно обрабатывает входящий аргумент, отвечающий за размер копируемой памяти. В результате функция выполняет вызов `memcpy(pDst, pSrc, size)`, где переменная `size` контролируется пользователем. WinHeapExplorer уведомляет об ошибке в msvcr100.dll по смещению в библиотеке `0x34cf3`.
- KingView 6.53 SCADA System (CVE 2011-0406) HMI Remote Heap-Overflow. Уязвимость происходит в модуле HistorySrv. Приложение выделяет `0x100` байт памяти, а затем в результате некорректной обработке входящего пакета данных копирует `0x140` байт с помощью инструкцию `rep movsb`. WinHeap Explorer уведомляет о наличии переполнения буфера по смещению `0x1aef` в модуле HistorySrv.exe.
- RealPlayer 14.0.1.633 (CVE-2011-1525) Local Heap Overflow. В результате некорректной обработки входящих данных, существует возможность контролировать размер выделяемого блока памяти. Приложение пытается записать `0x500` байт в контролируемый блок (мы можем выделить меньший объем памяти). В результате WinHeap Explorer детектирует переполнение кучи в функции memcpy по адресу `0x7855ae7a` в msvcr100.dll.

Ссылки на описание уязвимостей приведены здесь [21].

Выводы

Таким образом, ДБИ это мощная технология, позволяющая выполнять детальный анализ исполняемого кода. Существующие решения на её базе успешно применяются для анализа вредоносного кода, поиска уязвимостей, реверс-инжиниринга, а также во многих других областях Computer Science. В работе мы представили утилиту WinHeap Explorer для автоматизированного детектирования ошибок переполнения в куче, позволяющую выполнять «частичную» инструментацию и построенную на базе Intel PIN. На ее примере были показаны те сложности с которыми приходится сталкиваться при работе с ДБИ, а также приведен ряд примеров детектирования переполнения кучи и use-after-free с её помощью.

Ссылки

[1] D. Dorsey .Why Don't You Just Tell Me Where The ROP Isn't Suppose To Go. DefCon 22.
[2] SeokWoo Choi. API Deobfuscator. Resolving Obfuscated API Functions in Modern Packers. Black Hat USA 2015.
[3] S. Mariani, L. Fontana, F. Gritti, S. D'Alessio. PinDemonium. a DBI-based generic unpacker for Windows executables. BlackHat USA 2016.
[4] J. Wook Oh. Vulnerability Analysis and Practical Data Flow Analysis & Visualization. CanSecWest 2012.
[5] N. Nethercote, J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." *ACM Sigplan notices*. Vol. 42. No. 6. ACM, 2007.
[6] Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." *Acm sigplan notices*. Vol. 40. No. 6. ACM, 2005.
[7] D. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. Diss. Massachusetts Institute of Technology, 2004.
[8] K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference* (pp. 309-318).
[9] P. Godefroid, M. Levin, D. Molnar. "Automated Whitebox Fuzz Testing." *NDSS*. Vol. 8. 2008.
[10] M. Moroz. Scalable and Effective Fuzzing of Google Chrome. Positive Hack Days 2016.
[11] L. Davi, S. Ahmad-Reza, and M. Winandy. "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks." *Proceedings of the 2009 ACM workshop on Scalable trusted computing*. ACM, 2009.
[12] Z. Qin, A. Feng, et al. "Lift: A low-overhead practical information flow tracking system for detecting security attacks." *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.
[14] Y. Heng, et al. "Panorama: capturing system-wide information flow for malware detection and analysis." *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007.
[13] https://github.com/DynamoRIO/drmemory/tree/master/drltrace
[15] K. Sun, X. Li, Y. Ou. Break Out of the Truman Show. Active Detection and Escape Of Dynamic Binary Instrumentation. Blackhat Asia 2016
[16] F. Falcon and N. Riva. Detecting Binary Instrumentation Frameworks. ReCon – 2012.
[17] P. Saxena, R. Sekar, and V. Puranik. "Efficient fine-grained binary instrumentation with applications to taint-tracking." *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2008.
[18] https://software.intel.com/en-us/articles/pintool-dcfg
[19] Q. Zhao, et al. "How to do a million watchpoints: Efficient debugging using dynamic instrumentation." *International Conference on Compiler Construction*. Springer Berlin Heidelberg, 2008.
[20] https://msdn.microsoft.com/en-us/library/bb288454.aspx
[21]https://github.com/WinHeapExplorer/WinHeap-Explorer/blob/master/Links%20to%20exploits