СРЕДСТВО ОБНАРУЖЕНИЯ СКРЫТОГО ИСПОЛНИМОГО КОДА В ПАМЯТИ ОС WINDOWS

Поддубный В.А.[1], Коркин И.Ю.[2]

Цель работы: *опережающее совершенствование средств защиты информации для обнаружения преднамеренно скрытого исполнимого кода в оперативной памяти ОС Windows.*

Метод: *в работе используются методы теории графов и приёмы компьютерной криминалистики для анализа копий памяти.*

Полученный результат: *выявлены недостатки существующих способов обнаружения скрытого исполнимого кода в памяти. Предложено два способа сокрытия, реализующих наиболее сложный сценарий для обнаружения. При их применении совместно с остальными способами сокрытия стало возможным создать экспериментальный прототип исполнимого кода, необнаружимый существующими средствами. Для обнаружения такого кода авторами предложен способ, основанный на побайтовом поиске в памяти инструкций условного и безусловного перехода, построении с их помощью ориентированного графа и выделении его компонент слабой связности. На основе разработанного способа создано программное средство, способное обнаруживать 32-разрядный скрытый исполнимый код в копии виртуальной памяти процесса. Предложенный способ обнаружения не использует системные структуры и особенности исполнимых файлов, что делает противодействие ему крайне затруднительным.*

Ключевые слова: *копия памяти, исполнимый файл, энтропия, гистограмма, Zeus, последовательность байт, ассемблерная вставка, инструкции перехода, дизассемблирование, компонента слабой связности.*

DOI:10.21681/2311-3456-2019-5-75-82

1. Введение

Для обеспечения постоянного присутствия на заражённом компьютере нарушители могут использовать различные приёмы. Один из таких приёмов – руткит-механизмы, в результате действия которых информация о загруженном вредоносном ПО (ВПО) удаляется из списков процессов, драйверов и других системных списков. Указанная манипуляция с одной стороны исключает выявление загруженного ВПО с помощью штатных средств обнаружения, а с другой стороны не нарушает работоспособность ОС и других программ.

В работе проведён анализ и выявлены недостатки существующих способов обнаружения преднамеренно скрытого ВПО. Для обнаружения исполнимого кода такого ВПО авторами был выбран подход на основе анализа содержимого копии памяти.

2. Текущее состояние вопроса

Сокрытие исполнимого кода не может быть предотвращено такими штатными средствами ОС Windows, как Kernel Patch Protection, известным как PatchGuard, и запретом запуска неподписанных драйверов. В то же время оно применяется в различном ПО. Примером применения может служить ВПО Zeus.

Существует несколько способов обнаружения скрытого исполнимого кода в копии памяти. В работах [1–3] описан способ, основанный на просмотре двусвязных списков системных структур. При загрузке исполнимого файла в память создаётся системная структура, хранящая информацию о нём, и добавляется в соответствующий системный двусвязный список. Исполнимый код возможно скрыть от обнаружения данным способом путём модификации связей между структурами в списке. В результате модификации структура, соответствующая скрываемому исполнимому файлу, «выпадает» из списка и более не может быть обнаружена при просмотре списка.

В работе [4] описан способ, основанный на поиске удалённых из списков системных структур по известным заранее сигнатурам. В работе [5] предложено развитие сигнатурного способа, позволяющее динамически выявлять характерные особенности структур каждого из списков и целенаправленно искать удалённые из него структуры. Скрыть структуру от обнаружения таким способом возможно путём изменения её сигнатуры, модифицировав входящие в неё поля структуры.

Также возможно обнаружение исполнимых файлов путём поиска таких их структурных элементов, как PE-заголовок и таблица импорта. В работе [6] демонстрируется сокрытие исполнимого кода с помощью перезаписи PE-заголовка и таблицы импорта загруженного в память исполнимого файла.

В работе [7] предложен статистический способ обнаружения, основанный на вычислении энтропии способом «скользящего окна» с предварительным построением гистограммы частоты появления значений байта. Значения энтропии исполнимого кода находятся в известном диапазоне [8], благодаря чему он может быть обнаружен в копии памяти. Для сокрытия исполнимого кода от обнаружения путём вычисления энтропии в

1 Поддубный Владислав Александрович, студент кафедры «Криптология и кибербезопасность» НИЯУ МИФИ, г. Москва, Россия. E-mail: vladislav.poddubnyy@gmail.com

2 Коркин Игорь Юрьевич, кандидат технических наук, ведущий инженер-исследователь отдела информационной безопасности ООО «Центр Специальной Системотехники», г. Москва, Россия. E-mail: igor.korkin@gmail.com

него могут быть вставлены последовательности байт с низкой энтропией. Эта мера позволяет снизить энтропию исполнимого кода и избежать его обнаружения. Такой подход был применён в ВПО Zeus. Предварительное построение гистограммы частоты появления значений байта позволяет обнаружить такие последовательности и игнорировать их при вычислении энтропии, в результате чего обнаружение скрытого исполнимого кода становится возможным. Для создания наиболее сложного случая для обнаружения авторами было предложено два способа генерации последовательностей байт для вставки в исполнимый код.

3. Демонстрация возможности сокрытия

Используемый авторами подход, обеспечивающий сокрытие исполнимого кода от обнаружения статистическим способом, является развитием подхода, использованного ранее в ВПО Zeus [9]. Он основан на вставке в исполнимый код специальных последовательностей байт с низкой энтропией. Предлагается два способа формирования таких последовательностей байт.

Первый способ состоит в следующем:

- необходимо разделить все возможные значения байта на диапазоны по 7-10 значений;
- значение каждого вставляемого байта выбирается из первого диапазона случайным образом;
- после выбора значений каждых 4096 вставляемых байт заменять диапазон, из которого выбираются значения, на следующий.

Сравнение графиков энтропии и гистограмм частоты появления значений байт исполнимого файла, модифицированного применённым в Zeus способом и первым предлагаемым способом, приведено на рисунке 1. Здесь и далее энтропия вычислялась для окна размером 256 байт, шаг сдвига окна составляет 128 байт. Оба графика демонстрируют, что большинство значений энтропии исполнимого кода не превышает нижнюю границу диапазона значений, характерных для исполнимого кода (5 бит на байт [8]). Эксперимент показал, что вычисление энтропии способом «скользящего окна» не пригодно для обнаружения такого исполнимого кода.

На рисунке 1а представлена гистограмма исполнимого кода, модифицированного применённым в Zeus способом. На графике отчётливо выделяются значения 32, 64, 96, 128, 160, 192 и 224, которые встречаются наиболее часто. Это позволяет выявить факт вставки последовательностей байт, имеющих такие значения, и игнорировать байты с таким значением при вычислении энтропии. На гистограмме кода, модифицированного первым предлагаемым способом, отсутствуют резко выделяющиеся значения. Следовательно, с помощью анализа гистограммы невозможно определить, какие байты были вставлены, и игнорировать их при вычислении энтропии.

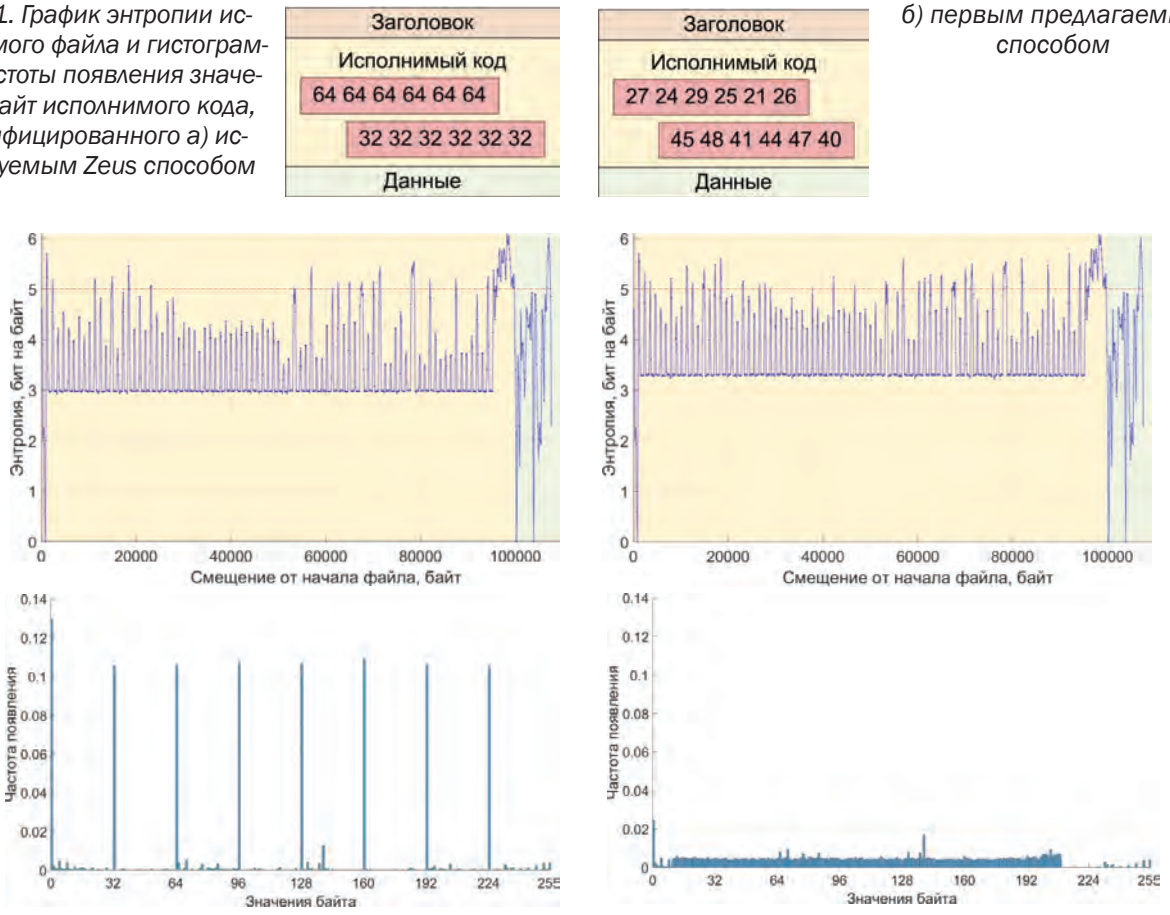


Рис. 1. График энтропии исполнимого файла и гистограмма частоты появления значений байт исполнимого кода, модифицированного а) используемым Zeus способом

б) первым предлагаемым способом

а

б

УДК 004.491.42: 004.492.3 DOI: 10.21681/2311-3456-2019-5-75-82

Второй способ состоит в следующем:

- из всех возможных значений байта необходимо выбрать 130 и разделить их на 13 диапазонов по 10 значений;
- значение каждого вставляемого байта выбирается из первого диапазона случайным образом;
- после выбора значений каждых 26624 вставляемых байт заменять диапазон, из которого выбираются значения, на следующий.

Пример графиков энтропии и гистограмм частоты появления значений байт исполнимого файла, модифицированного применённым в Zeus способом и вторым предлагаемым способом, приведен на рисунке 2. Оба графика демонстрируют, что большинство значений энтропии исполнимого кода не превышает нижнюю границу диапазона значений, характерных для исполнимого кода. Следовательно, обнаружить такой исполнимый код путём вычисления энтропии способом «скользящего окна» невозможно.

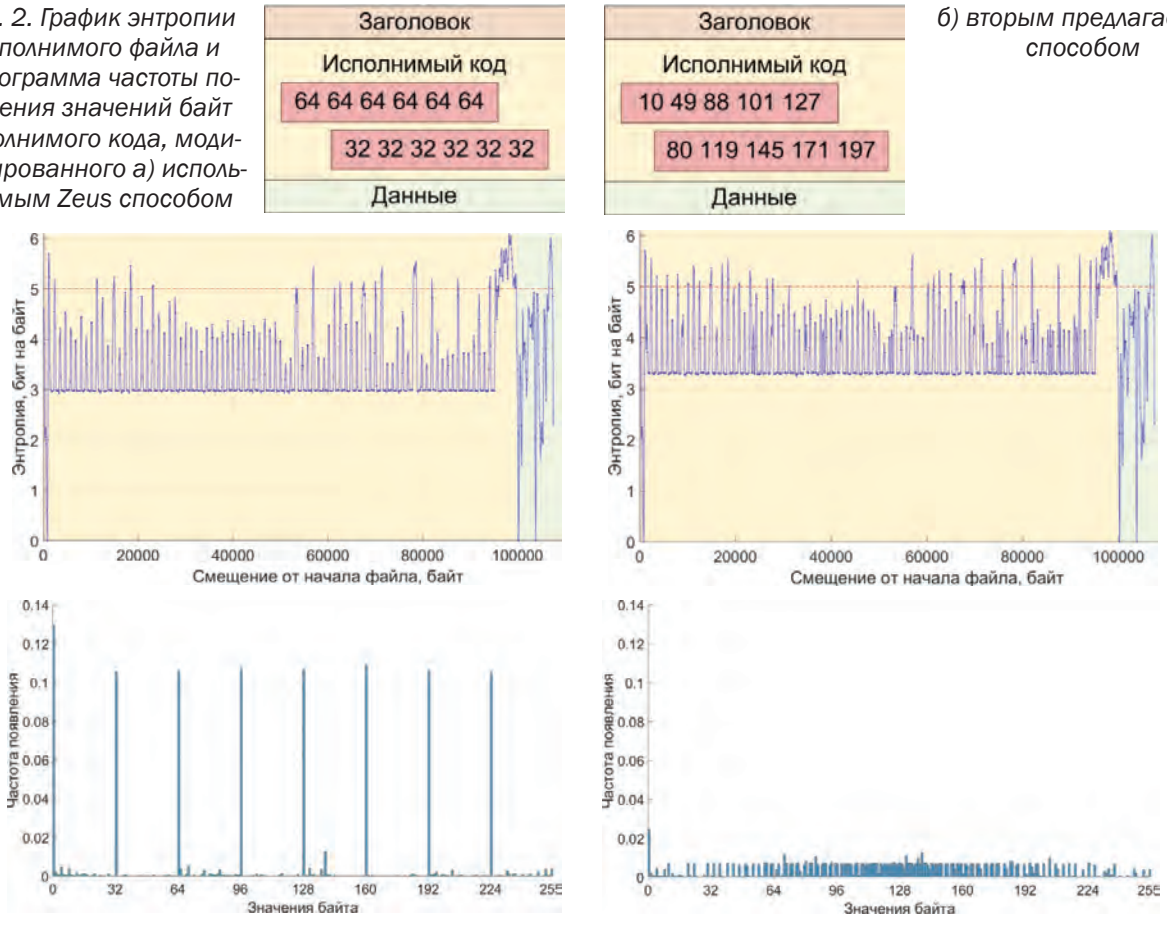
На гистограмме исполнимого кода, модифицированного вторым предлагаемым способом, выделяется 130 значений, что вызвано вставкой специальных последовательностей байт. Если при вычислении энтропии игнорировать все байты с такими значениями, то наряду с байтами вставленных последовательностей будут проигнорированы и байты исполнимого кода с теми же значениями, что сделает результат неинформативным.

Исследованные экспериментальные образцы ПО созданы из исходных текстов на языке С при помощи компилятора Microsoft Visual C++. Вставка последовательностей байт в исполнимый код производится во время сборки. Для этого в исходный текст программы необходимо предварительно добавить специального вида ассемблерные вставки, которые состоят из:

- инструкции безусловного перехода jmp, предназначенной для передачи управления на следующий после ассемблерной вставки оператор;
- псевдоинструкций _emit, предназначенных для вставки в исполнимый файл при сборке заданной последовательности байт;
- метки, расположенной в конце вставки, по которой передаётся управление упомянутым выше оператором безусловного перехода.

Пример ассемблерной вставки приведён на рисунке 3. Изображённая вставка предназначена для добавления в исполнимый файл при сборке последовательности из 5 байт со значением 144.

Ассемблерные вставки добавляются через каждые 7 операторов языка С, длина последовательности байт, вставляемой в исполнимый код при помощи каждой вставки, составляет 1024 байт.



а *б*

Рис. 2. График энтропии исполнимого файла и гистограмма частоты появления значений байт исполнимого кода, модифицированного а) используемым Zeus способом б) вторым предлагаемым способом

```
Table *createCache()
{
    int N;
    Table *cache;

__asm
{
jmp data10
_emit 144;
_emit 144;
_emit 144;
_emit 144;
_emit 144;
data10:
}

    printf("Enter cache size\n");
    if (!getInt(&N))
        return NULL; //eof
    cache = (Table *)malloc(sizeof(Table));
    cache->size = N;
    cache->elements = (Record *)calloc(N, sizeof(Record));
    return cache;
}
```

Рис. 3. Пример специальной ассемблерной вставки

4. Предлагаемый способ обнаружения

На рисунке 4 представлено схематичное изображение фрагмента копии памяти, содержащего два исполнимых файла и произвольные данные. В областях, содержащих исполнимые файлы, многоточиями обозначен исполнимый код, чёрными стрелками – передача управления инструкциями условного и безусловного перехода, словом «блок» – последовательности байт, вставленные в исполнимый код с целью его сокрытия. В областях фрагмента, не содержащих исполнимые файлы, многоточиями обозначены произвольные данные.

4.1 Поиск и проверка инструкций перехода

Копия памяти последовательно просматривается с целью обнаружения инструкций условного и безусловного перехода. В случае обнаружения инструкции перехода необходимо удостовериться, что инструкция находится в составе исполнимого кода, а не является неверно интерпретированными байтами данных. Проверка состоит из двух этапов.

На первом этапе производится дизассемблирование байт, находящихся перед найденной инструкцией перехода с целью получения 25 предшествующих ей инструкций. Так как длина машинной инструкции в архитектуре x86 не фиксирована, то неизвестно, каким количеством байт представлены 25 инструкций. Поэтому необходимо произвести несколько попыток дизассемблирования, каждый раз дизассемблируя различное количество байт перед инструкцией перехода. Такой подход применялся в работе [10]. Количество дизассемблируемых байт для 32-разрядного исполнимого кода вычисляется из расчёта от 2,7 до 4 байт на инструкцию, что было определено экспериментально. Если будет найдено количество байт, безошибочно дающих в результате дизассемблирования заданное число инструкций, необходимо перейти ко второму этапу. В противном случае инструкция перехода признаётся неверно интерпретированными байтами данных, и просмотр копии памяти с целью поиска инструкций перехода продолжается.

На втором этапе производится проверка наличия определённых инструкций среди дизассемблированных. Списки инструкций определены исследователями компании FireEye [11] для определения корректности дизассемблирования и включают:

- привилегированные инструкции;
- инструкции, не используемые компиляторами;
- редко используемые инструкции;

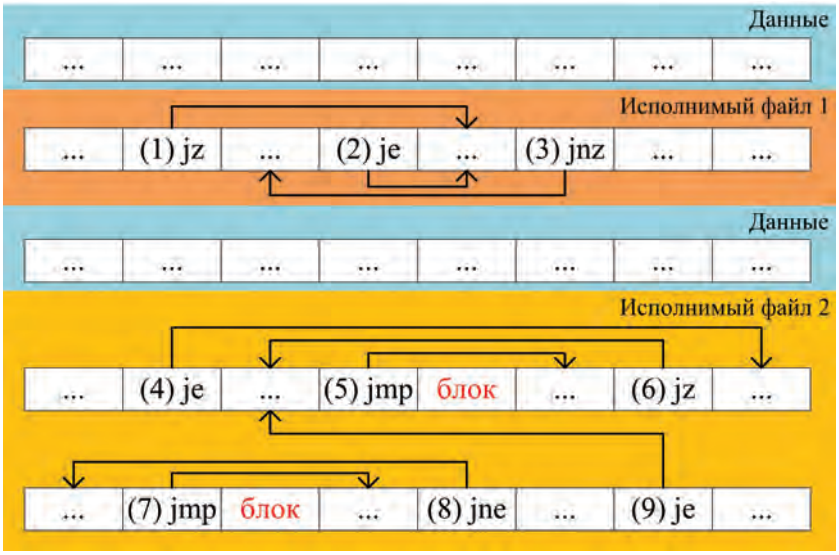


Рис. 4. Фрагмент копии памяти

УДК 004.491.42: 004.492.3 DOI: 10.21681/2311-3456-2019-5-75-82

- инструкции, предназначенные для работы с дальними указателями;
- инструкции, содержащие определённые префиксы.

В случае обнаружения инструкций из какой-либо категории найденная инструкция перехода пропускается, и просмотр копии памяти продолжается.

Привилегированные инструкции используются только в исполнимом коде ядра операционной системы и драйверов привилегированного режима, которые располагаются в системной области памяти. Такие инструкции не могут встречаться в исполнимом коде, загруженного в пользовательскую область память.

4.2 Построение графа инструкций перехода

На основе найденных инструкций перехода производится построение ориентированного графа, называемого графом инструкций перехода. Вершинами графа являются найденные инструкции. Вершины соединяются рёбрами по двум соображениям.

Первое соображение состоит в следующем:

- определить, по какому смещению представленная вершиной инструкция перехода передаёт управление;
- определить, какая из найденных инструкций перехода будет выполнена первой после передачи управления по этому смещению;
- соединить ребром вершины, представляющие эти инструкции перехода.

Если представленная вершиной инструкция перехода является инструкцией условного перехода, то к ней применимо второе соображение:

- определить, какая из найденных инструкций перехода будет выполнена первой при невыполнении условия инструкции;
- если расстояние между этими инструкциями меньше 1280 байт, то соединить ребром вершины, представляющие эти инструкции.

Граф инструкций перехода для фрагмента копии памяти, изображённого выше, представлен на рисунке 5.

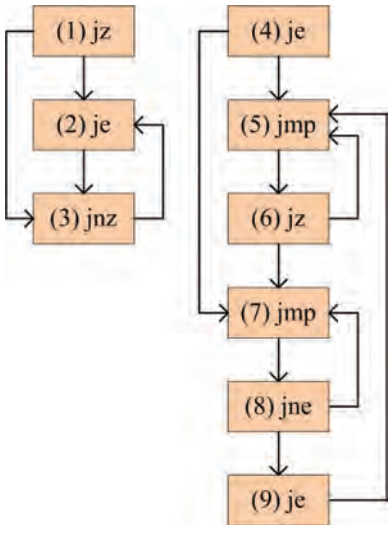


Рис. 5. Граф инструкций перехода для фрагмента копии памяти

4.3 Выделение компонент слабой связности

Из построенного графа инструкций перехода выделяются его компоненты слабой связности (КСС). Компоненты слабой связности графа инструкций перехода, изображённого выше, приведены на рисунке 6. Каждая компонента представляет собой исполнимый код, расположенный в соответствующей области копии памяти и входящий в состав исполнимого файла. Области копии памяти, в которых был обнаружен исполнимый код, подлежат дальнейшей экспертной оценке с целью определения исполнимого файла, содержащего этот исполнимый код, а также исследования этого файла.

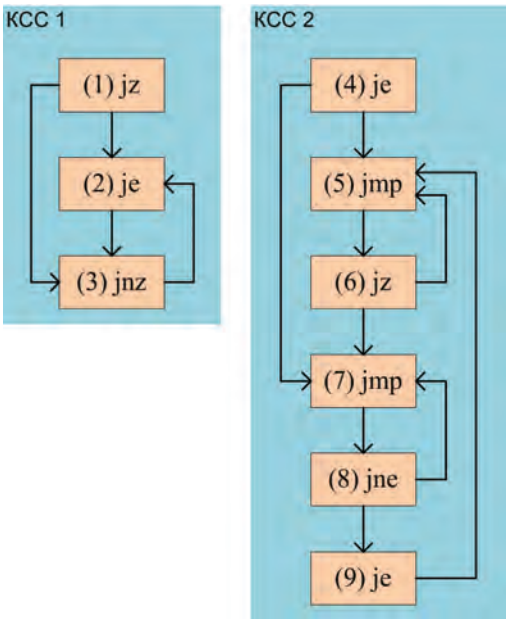


Рис. 6. Компоненты слабой связности графа инструкций перехода

5. Экспериментальная проверка

На основе предложенного способа создано программное средство обнаружения скрытого исполнимого кода. Средство реализовано на языке C# и предназначено для исследования копии виртуальной памяти процесса, которую необходимо получить предварительно. С помощью средства возможно исследование как всей копии памяти, так и некоторой её области, что определяется путём передачи соответствующих параметров при его запуске. Дизассемблирование осуществляется средствами библиотеки SharpDisasm [12], построение графов и работа с ними – средствами библиотеки QuickGraph [13].

В процессе тестирования проведён анализ копии памяти системы под управлением 64-разрядного выпуска ОС Windows 10 Pro версии 1809 (сборка 17763.194). Был запущен экспериментальный образец, скрытый предложенным способом, после чего получена копия физической памяти системы с помощью ПО Passmark OSForensics [14].

Дальнейший анализ копии памяти, в том числе запуск средства обнаружения, производился на системе

под управлением 64-разрядного выпуска ОС Windows 7 Professional SP1 (сборка 7601). С помощью The Volatility Foundation [4] и плагина memdump выделена копия виртуальной памяти процесса экспериментального образца. Далее файл копии виртуальной памяти и экспериментальный образец были открыты в шестнадцатеричном редакторе с целью определения смещения, по которому в копии памяти расположен экспериментальный образец. Под смещением понимается смещение от начала файла, содержащего копию виртуальной памяти, записанное в шестнадцатеричной системе счисления. Экспериментальный образец располагается по смещению 2E000, что продемонстрировано на рисунке 7. Исполнимый код образца располагается в области 2F000 – 342F5.

При запуске средству переданы следующие параметры:

- имя файла, содержащего копию виртуальной памяти процесса экспериментального образца;
- начальное смещение исследуемой области копии памяти – 2D000;
- конечное смещение исследуемой области копии памяти – 35000.

Результат работы средства приведён на рисунке 8. Экспериментальная проверка показала, что преднамеренно скрытый программный образец исполнимого кода был успешно обнаружен в копии памяти.

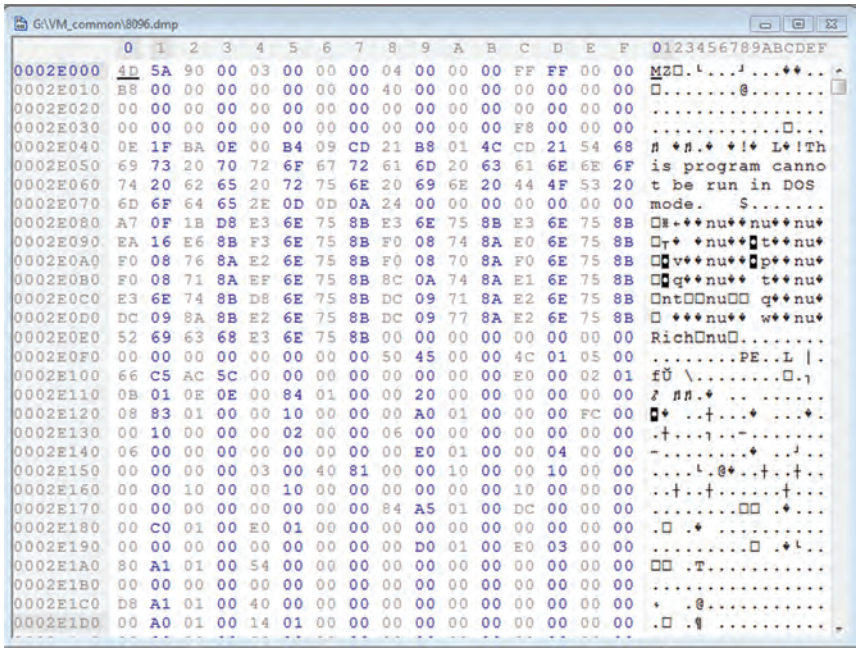


Рис. 7. Расположение экспериментального образца в копии памяти

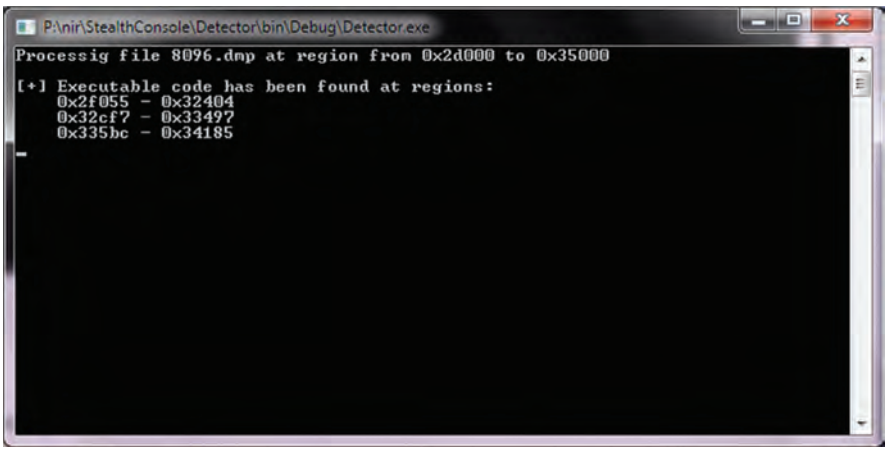


Рис. 8. Результат анализа копии памяти, содержащей экспериментальный образец

6. Выводы

В работе проанализированы существующие способы обнаружения скрытого исполнимого кода в копии памяти и выявлены их недостатки, на основе чего предложено два перспективных способа сокрытия. При их применении совместно с прочими выявленными способами возможно создать необнаружимый существующими средствами исполнимый код. Достигнутый результат является существенным развитием руткит-механизмов.

Для обнаружения такого исполнимого кода разработан способ, основанный на поиске в копии памяти инструкций условного и безусловного перехода, и на его основе создано программное средство, способное обнаруживать 32-разрядный скрытый исполнимый код в копии виртуальной памяти процесса.

Полученные результаты могут быть применены при разработке новых и улучшении существующих средств защиты информации и расследования инцидентов информационной безопасности, способных выявлять присутствие скрытого исполнимого кода в памяти ОС Windows.

Дальнейшее направление работ – добавление возможности обнаружения 64-разрядного скрытого исполнимого кода и поддержка новых аппаратных платформ.

Литература

1. Tsaur W.-J., Wu J.-X. New Windows Rootkit Technologies for Enhancing Digital Rights Management in Cloud Computing Environments // Proceedings of The 2014 International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government. 2014.
2. Eresheim S., R. Luh, S. Schrittwieser The Evolution of Process Hiding Techniques in Malware – Current Threats and Possible Countermeasures // Journal of Information Processing. 2017. Vol. 25. P. 866–874. DOI: 10.2197/ipsjjip.25.866
3. Vomel S., Lenz H. Visualizing Indicators of Rootkit Infections in Memory Forensics // Proceedings of The Seventh International Conference on IT Security Incident Management and IT Forensics. 2013. P. 122–139. DOI: 10.1109/IMF.2013.12
4. Ligh M.H., Case A., Levy J., Walters A. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory Indianapolis. Wiley, 2014.
5. Korkin I., Nesterov I. Applying Memory Forensics to Rootkit Detection // Proceedings of The 9th ADFSL Conference on Digital Forensics, Security and Law. 2014. P. 115–141.
6. Kawakoya Y., Shioji E., Otsuki Y. et al. Stealth Loader: Trace-Free Program Loading for API Obfuscation // Proceedings of 20th International Symposium on Research in Attacks, Intrusions, and Defenses. 2017. P. 217–237. DOI: 10.1007/978-3-319-66332-6_10
7. Ugarte-Pedrero X., Santos I., Sanz B. et al. Countering Entropy Measure Attacks on Packed Software Detection // Proceedings of the 9th IEEE Consumer Communications and Networking Conference. 2012. DOI: 10.1109/CCNC.2012.6181079
8. Lyda R., Hamrock J. Using Entropy Analysis to Find Encrypted and Packed Malware. // IEEE Security and Privacy Magazine. 2007. Vol. 5. N. 2. P. 40–45. DOI: 10.1109/msp.2007.48
9. Korkin I., Nesterow I. Acceleration of Statistical Detection of Zero-day Malware in the Memory Dump Using CUDA-enabled GPU Hardware // Proceedings of The 11th ADFSL Conference on Digital Forensics, Security and Law. 2016.
10. Bauman E., Lin Z., Hamlen K.W. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics // Proceedings of Network and Distributed System Security Symposium. 2018. P. 40–47. DOI: 10.14722/ndss.2018.23300
11. Recognizing and Avoiding Disassembled Junk. Блог компании FireEye. URL: https://www.fireeye.com/blog/threat-research/2017/12/recognizing-and-avoiding-disassembled-junk.html. (дата обращения: 18.06.2019).
12. Репозиторий библиотеки SharpDisasm. URL: https://github.com/spazzarama/SharpDisasm. (дата обращения: 18.06.2019).
13. Репозиторий библиотеки QuickGraph. URL: https://github.com/YaccConstructor/QuickGraph. (дата обращения: 18.06.2019).
14. Описание ПО Passmark OSForensics. URL: https://www.osforensics.com/osforensics.html (дата обращения: 18.06.2019).

ADVANCED ROOTKIT DETECTION USING MEMORY FORENSICS

Poddubnyy V.[3], Korkin I.[4]

Purpose: *improvement of memory forensic systems for detection of deliberately hidden executable code in Windows memory.*

Research methods: *graph theory methods and digital forensics approaches to memory dump analysis.*

Results: *two methods of creation of a highly stealth executable code are presented. Such executable code cannot be detected with existing statistical approach. If combined with other known anti-forensics techniques, the*

3 Vladisalv Poddubnyy, student of department №42 «Cryptology and cybersecurity» of NRNU MEPhI, Moscow, Russia. E-mail: vladislav.poddubnyy@gmail.com

4 Igor Korkin, Ph.D., Lead Security Research Engineer, Special System Engineering Centre (ssec.ru), Moscow, Russia. E-mail: igor.korkin@gmail.com

most complicated scenario for detection will be created. Second, a new detection method is presented, which is resilient to this hidden executable code. The method is based on a byte-wise search of conditional and unconditional jump instructions in a memory dump. Instructions found are used to create a directed graph. Each weakly connected component of this graph represents an area of a memory dump containing executable code. Proposed method does not rely on system structures and executable file features, which makes it resilient to popular anti-forensic techniques.

Keywords: *digital forensics, memory dump, executable code, entropy, byte histogram, Zeus, byte sequence, inline assembler, jump instruction, disassembly, weakly connected component.*

References

1. Tsaur W.-J., Wu J.-X. New Windows Rootkit Technologies for Enhancing Digital Rights Management in Cloud Computing Environments // Proceedings of The 2014 International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government. 2014.
2. Eresheim S., R. Luh, S. Schrittwieser The Evolution of Process Hiding Techniques in Malware – Current Threats and Possible Countermeasures // Journal of Information Processing. 2017. Vol. 25. P. 866–874. DOI: 10.2197/ipsjjip.25.866
3. Vomel S., Lenz H. Visualizing Indicators of Rootkit Infections in Memory Forensics // Proceedings of The Seventh International Conference on IT Security Incident Management and IT Forensics. 2013. P. 122–139. DOI: 10.1109/IMF.2013.12
4. Ligh M.H., Case A., Levy J., Walters A. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory Indianapolis. Wiley, 2014.
5. Korkin I., Nesterov I. Applying Memory Forensics to Rootkit Detection // Proceedings of The 9th ADFSL Conference on Digital Forensics, Security and Law. 2014. P. 115–141.
6. Kawakoya Y., Shioji E., Otsuki Y. et al. Stealth Loader: Trace-Free Program Loading for API Obfuscation // Proceedings of 20th International Symposium on Research in Attacks, Intrusions, and Defenses. 2017. P. 217–237. DOI: 10.1007/978-3-319-66332-6_10
7. Ugarte-Pedrero X., Santos I., Sanz B. et al. Countering Entropy Measure Attacks on Packed Software Detection // Proceedings of the 9th IEEE Consumer Communications and Networking Conference. 2012. DOI: 10.1109/CCNC.2012.6181079
8. Lyda R., Hamrock J. Using Entropy Analysis to Find Encrypted and Packed Malware. // IEEE Security and Privacy Magazine. 2007. Vol. 5. N. 2. P. 40–45. DOI: 10.1109/msp.2007.48
9. Korkin I., Nesterow I. Acceleration of Statistical Detection of Zero-day Malware in the Memory Dump Using CUDA-enabled GPU Hardware // Proceedings of The 11th ADFSL Conference on Digital Forensics, Security and Law. 2016.
10. Bauman E., Lin Z., Hamlen K.W. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics // Proceedings of Network and Distributed System Security Symposium. 2018. P. 40–47. DOI: 10.14722/ndss.2018.23300
11. Recognizing and Avoiding Disassembled Junk. URL: https://www.fireeye.com/blog/threat-research/2017/12/recognizing-and-avoiding-disassembled-junk.html. (accessed: 18.06.2019).
12. Github repository of SharpDisasm library. URL: https://github.com/spazzarama/SharpDisasm. (accessed: 18.06.2019)
13. Github repository of QuickGraph library. URL: https://github.com/YaccConstructor/QuickGraph. (accessed: 18.06.2019).
14. Passmark OSForensics software description. URL: https://www.osforensics.com/osforensics.html (accessed: 18.06.2019).