УДК 004.492.3

Средство обнаружения скрытого исполнимого кода в памяти ОС на базе Linux

В. А. Поддубный[1]*

[1] Национальный исследовательский ядерный университет «МИФИ», г. Москва, Российская Федерация
* e-mail: vladislav.poddubnyy@gmail.com

Аннотация. Цель работы – опережающее совершенствование средств защиты информации для обнаружения преднамеренно скрытого исполнимого кода в оперативной памяти ОС на базе Linux. В работе используются приёмы компьютерной криминалистики и техники дизассемблирования для анализа копий памяти. В результате работы выявлены недостатки существующих способов обнаружения скрытого исполнимого кода в памяти. При реализации продемонстрированного комплексного сценария сокрытия существующие способы обнаружения оказываются неэффективными, а для анализа остаются доступны только машинные инструкции, составляющие исполнимый код. Предложенный способ обнаружения, основанный на восстановлении графа потока управления, позволяет обнаружить исполнимый код, скрытый в результате реализации показанного сценария. Способ при анализе полагается только на машинные инструкции исполнимого кода, что делает противодействие ему затруднительным и потенциально позволяет обнаруживать исполнимый код, скрытый ещё не известными перспективными способами. Действенность способа показана путём разработки на его основе средства, способного обнаруживать скрытый исполнимый код в копии виртуальной памяти процесса.

Ключевые слова: Linux, копия памяти, скрытый исполнимый код, граф потока управления

Linux rootkit detection using memory forensics and control flow graphs

V. A. Poddubnyy[1]*

[1] National Research Nuclear University MEPhI (Moscow Engineering Physics Institute), Moscow,
Russian Federation
* e-mail: vladislav.poddubnyy@gmail.com

Abstract. This research aims to improve memory forensics systems for detection of deliberately hidden executable code in Linux memory by applying digital forensics approaches and disassembly techniques to memory dump analysis. After conducting analysis of existing detection approaches and their disadvantages, a complex approach to creation of highly stealth executable code is proposed. Such executable code cannot be detected with existing approaches as it presents the most complicated scenario for detection, making machine instructions that make up the executable code the only available information left for analysis. A new detection approach based on reconstruction of code's control flow graph (CFG) is proposed, able to detect executable code even in a highly complicated scenario. The proposed approach does not rely on any information besides executable code's instructions, making it resilient to both common and upcoming anti-forensic techniques.

Keywords: Linux, memory dump, rootkit detection, control flow graph

Введение

Одна из целей разработчиков вредоносного ПО (ВПО) – обеспечение возможно более длительного скрытого присутствия на поражённом компьютере. Зачастую она достигается путём использования руткит-механизмов – модификации системных структур и списков (процессов, модулей ядра и других). Такие действия предотвращают обнаружение ВПО с помощью штатных средств ОС, не нарушая работоспособности самого ВПО, ОС и других программ.

Штатные средства ОС на базе Linux не позволяют предотвратить сокрытие исполнимого кода. Так, по умолчанию отсутствуют средства контроля целостности, аналогичные Kernel Patch Protection (PatchGuard), встроенному в ОС Windows. В то же время выявлены образцы ВПО, успешно применяющие техники сокрытия исполнимого кода (Snakso [0], Skidmap [2]).

Существуют сторонние средства, осложняющие сокрытие исполнимого кода. Примером такого средства может служить LKRG [3], являющееся аналогом PatchGuard. Как и PatchGuard, LKRG способно обнаружить исполнимый код, скрытый путём модификации системных структур. Однако, по заявлению автора, средство не является универсальным и может быть обойдено. То же касается и других подобных средств.

Также существуют способы обнаружения скрытого исполнимого кода в копии памяти. Один из таких способов, описанный в работах [4–6], – просмотр двусвязных списков системных структур. В памяти ОС находится несколько двусвязных списков, содержащих структуры с информацией о запущенных процессах, загруженных динамических библиотеках, модулях ядра и соответствующих им исполнимым файлам. Эти структуры возможно модифицировать так, чтобы соответствующая скрываемому объекту структура была более не связана с соседними. В результате исполнимый код объекта будет скрыт, так как структура не будет обнаружена при просмотре списка.

Другая группа способов основана на сигнатурном обнаружении системных структур. Поиск структур в копии памяти возможно осуществлять как по статическим, известным заранее сигнатурам [7], так и по динамически выявляемым сигнатурам, характерных для структур из конкретного списка [8]. Сокрытие структуры от обнаружения достигается путём изменения её сигнатур модификацией входящих в неё полей.

Сигнатурное обнаружение применимо не только к системным структурам. Так, возможно обнаружение исполнимых ELF-файлов с помощью поиска их характерных структурных элементов: заголовка, таблицы импорта, прологов и эпилогов функций, составляющих исполнимый код. Сокрытие от обнаружения достигается так же, как и для прочих сигнатурных способов – путём изменения/удаления сигнатур. Исполнимый код может быть обфусцирован, а заголовок и таблица импорта – перезаписаны после запуска файла [9].

Ещё один способ обнаружения исполнимого кода в копии памяти – статистический [10]. В его основе лежит тот факт, что энтропия исполнимого кода лежит в известном диапазоне [11]. Таким образом, для обнаружения исполнимого

кода необходимо провести вычисление энтропии участков копии памяти способом скользящего окна. Однако в работе [12] предложен способ сокрытия, заключающийся во вставке в исполнимый код специальных последовательностей байт с низкой энтропией. Такая мера позволяет избежать обнаружения исполнимого кода статистическим способом.

Таким образом, для каждого из рассмотренных способов обнаружения выявлен соответствующий способ сокрытия, при применении которого способ обнаружения становится неэффективным.

Комплексный сценарий сокрытия

На основе выявленных способов сокрытия исполнимого кода в памяти составлен комплексный сценарий, наиболее сложный для обнаружения:

– прологи и эпилоги всех функций обфусцированы;

– все внешние функции находятся в динамических библиотеках, загружаемых во время исполнения с помощью функции загрузки библиотеки dlopen и функции получения адреса требуемого символа dlsym;

– в исполнимом коде присутствуют специальные последовательности байт, препятствующие его обнаружению с помощью статистического способа;

– после загрузки в память и запуска исполнимого файла его заголовок перезаписывается;

– после загрузки в память и запуска исполнимого файла системные структуры, содержащие информацию о нём (например, структура task_struct, содержащая информацию о процессе) специальным образом модифицируются и удаляются из соответствующих системных списков.

При реализации такого сценария, для обнаружения исполнимого кода становится невозможным использование каких-либо связанных с ним объектов кроме непосредственно составляющих его машинных инструкций. Таким образом, показана необходимость построения способа обнаружения исполнимого кода, опирающегося в своей работе только на входящие в состав кода инструкции.

Предлагаемый способ обнаружения

Суть предлагаемого способа обнаружения заключается в восстановлении графа потока управления скрытого исполнимого кода путём дизассемблирования копии памяти способом скользящего окна с непрерывной проверкой корректности получаемых результатов.

Граф потока управления – множество всех возможных путей исполнения программы, представленное в виде графа [13]. Граф потока управления программы возможно восстановить путём дизассемблирования входящих в состав её исполнимого кода инструкций. В контексте работы применим только метод рекурсивного спуска, так как метод линейного прохода подвержен ошибкам при попытке дизассемблирования последовательностей байт, не являющихся исполнимым кодом, но входящим в его состав. Такая ситуация реализуется при сокрытии исполнимого кода путём добавления специальных последовательностей байт [12].

При реализации рассматриваемого сценария сокрытия заранее не известно смещение от начала копии памяти, по которому располагается исполнимый код. Поэтому способ обнаружения должен быть способен различать исполнимый код и прочие данные, не являющиеся исполнимым кодом. Кроме того, исполнимый код может быть неверно дизассемблирован, если дизассемблирование произведено по не совпадающему с границей инструкций смещению. Следовательно, необходимо проверять, является ли результат дизассемблирования каждой очередной инструкции корректным.

Проверка корректности очередной полученной инструкции осуществляется в два этапа. На первом этапе проверяется, не принадлежит ли полученная инструкция одному из следующих классов:

– инструкции привилегированного режима;
– инструкции, не используемые компиляторами;
– инструкции, редко используемые в современном коде;
– инструкции, использующие дальние указатели;
– инструкции с определёнными префиксами.

Списки инструкций, составляющих перечисленные классы, определены исследователями компании FireEye [14]. В случае принадлежности инструкции одному из этих классов она признаётся некорректной.

Второй этап проверки выполняется для инструкций передачи управления и заключается в проверке целевого смещения, по которому передаётся управление. Если управление передаётся уже исследованному исполнимому коду, однако передаётся некорректно (например, целевое смещение не совпадает с началом какой-либо исследованной инструкции), то полученная инструкция признаётся некорректной.

Особого внимания требует исполнимый код, скомпонованный динамически в отложенном времени. При применении такого способа компоновки связывание исполняемого кода с внешними символами происходит «по требованию» — в момент первого обращения к этому символу. По этой причине на момент получения копии памяти не все внешние символы исполняемого в этот момент кода могут быть связаны с ним. Такая ситуация представляет проблему в рамках рассматриваемого сценария сокрытия, так как при его реализации недоступна сопутствующая исполнимому коду служебная информация из исполнимого файла. В результате становится невозможно однозначно распознать временные «заглушки» динамического компоновщика для ещё не связанных символов. В то же время, при попытке продолжения дизассемблирования после прохода нераспознанной «заглушки» возможно возникновение ошибок дизассемблирования. «Заглушки» динамического компоновщика имеют вид базового блока, управление которому передаётся при помощи инструкции вызова подпрограммы и первая инструкция которого – инструкция безусловного перехода (рис. 1). По этой причине все базовые блоки такого вида признаются потенциальными «заглушками» динамического компоновщика и далее не исследуются во избежание возникновения ошибок.

```
LOAD:0000000000401110 sub_401110      proc near               ; CODE XREF: sub_401BF0+1C↓p
LOAD:0000000000401110                                         ; sub_401BF0+64↓p ...
LOAD:0000000000401110                 jmp     cs:qword_407088
LOAD:0000000000401110 sub_401110      endp
```

Рис. 1. «Заглушка» динамического компоновщика

Таким образом, в ходе работы способа исполнимый код не будет обнаружен при несовпадении границы скользящего окна с началом исполнимого кода (рис. 2) из-за возникших ошибок дизассемблирования или некорректности результатов. Однако на одной из итераций граница скользящего окна совпадёт с началом исполнимого кода (рис. 3), дизассемблирование завершится успешно и будет восстановлен граф потока управления (рис. 4). Факт успешного восстановления графа потока управления позволяет сделать вывод об обнаружении исполнимого кода.

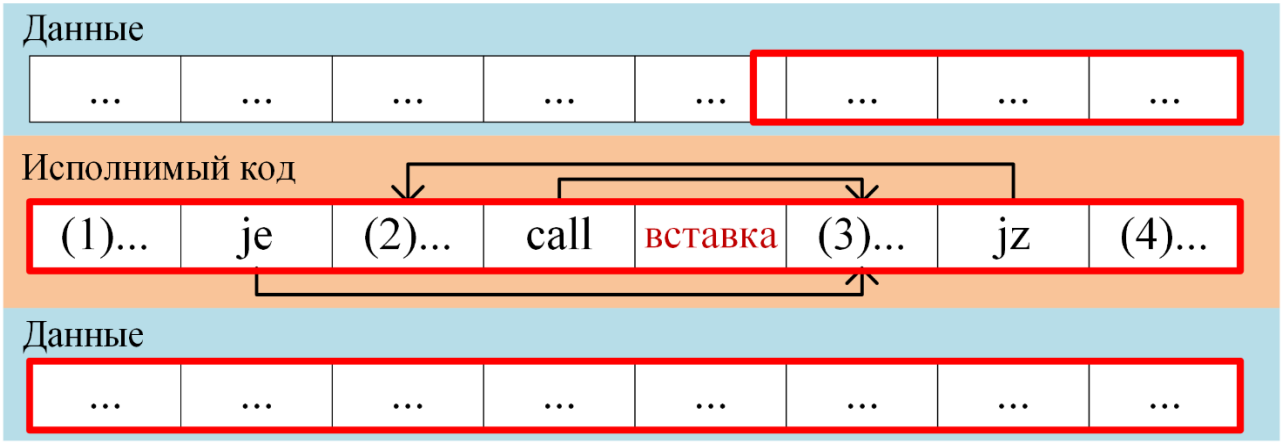

Рис. 2. Несовпадение границы скользящего окна (показано красным) с началом исполнимого кода

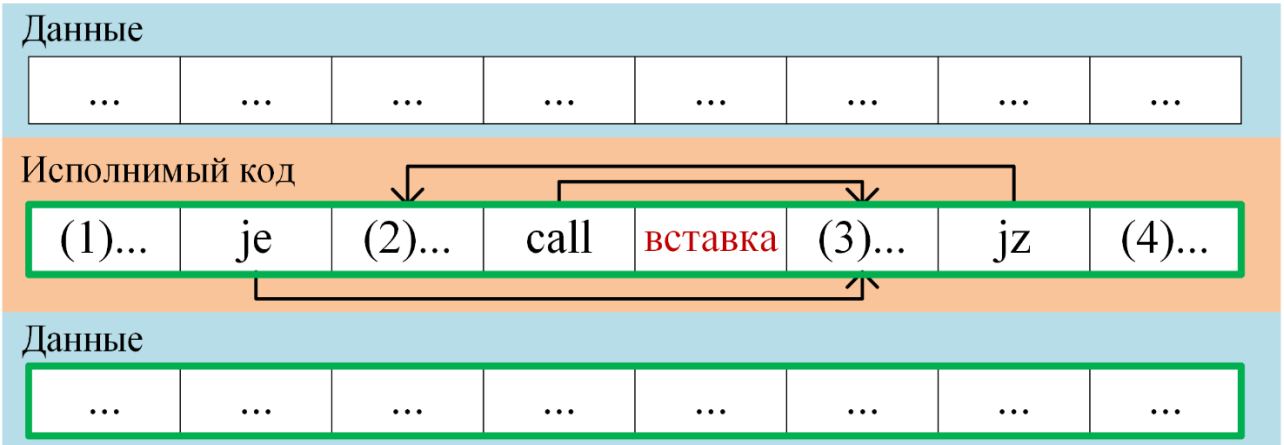

Рис. 3. Совпадение границы скользящего окна (показано зелёным) с началом исполнимого кода

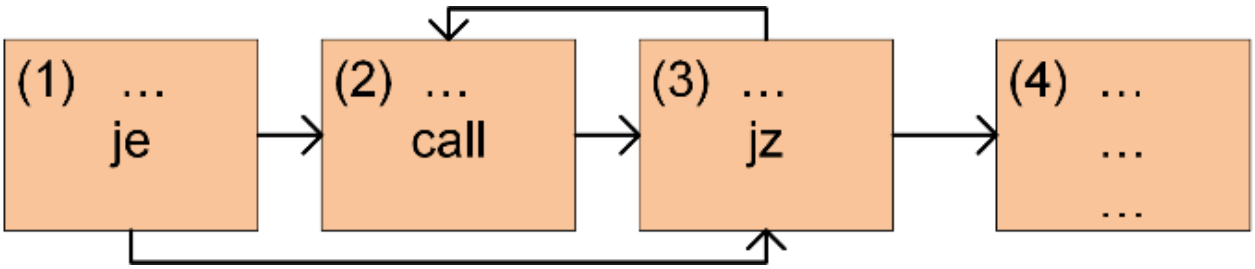

Рис. 4. Восстановленный граф потока управления

Экспериментальная проверка

На основе предложенного способа создано программное средство обнаружения скрытого исполнимого кода для архитектуры x86_64 в копии виртуальной памяти процесса. Дизассемблирование осуществляется средствами библиотеки SharpDisasm [15].

В процессе тестирования проведён анализ копии памяти процесса экспериментального образца, к которому был применён способ сокрытия, описанный в [12]. Копия памяти получена описанным в [16] способом. В результате работы средства исполнимый код экспериментального образца был успешно обнаружен, в частности обнаружена функция main (рис. 5).

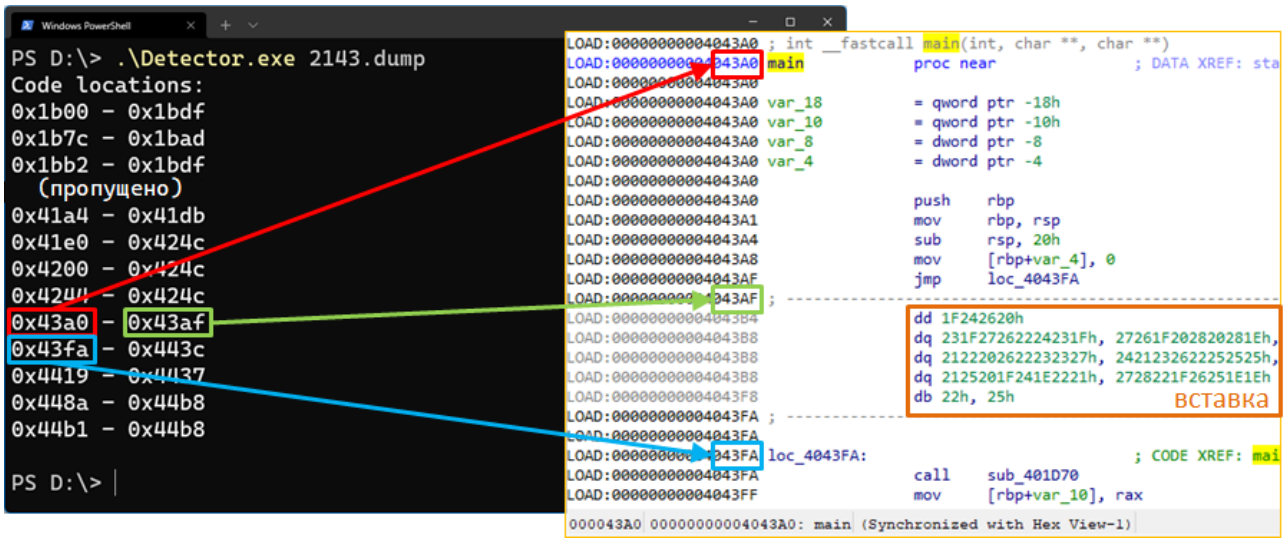

Рис. 5. Успешное обнаружение исполнимого кода экспериментального образца

Заключение

В работе проанализированы существующие способы обнаружения скрытого исполнимого кода в копии памяти. Для каждого из способов выявлен соответствующий способ сокрытия, при применении которого способ обнаружения становится неэффективным. На основе выявленных способов составлен комплексный сценарий сокрытия, представляющий наибольшую сложность для обнаружения. При реализации такого сценария обнаружить исполнимый код становится

невозможно ни одним из рассмотренных способов, а единственной доступной информацией для анализа остаются составляющие его машинные инструкции.

Для обнаружения такого исполнимого кода в копии памяти предложен способ, основанный на восстановлении его графа потока управления. Обнаружение осуществляется только на основе инструкций, составляющих исполнимый код, что позволяет обнаруживать как исполнимый код, скрытый в результате реализации предложенного сценария, так и в результате применения других перспективных способов сокрытия. Работоспособность способа подтверждена путём создания на его основе средства обнаружения скрытого исполнимого кода в виртуальной памяти процесса. Средство способно успешно обнаруживать исполнимый код для архитектуры x86_64.

Полученные результаты могут быть использованы при совершенствовании средств защиты информации и систем расследования инцидентов информационной безопасности для обнаружения перспективных угроз в ОС на базе Linux, связанных с применением сокрытия исполнимого кода.

Дальнейшее направление работ – доработка разработанного средства с целью поддержки платформ, отличных от x86 (в первую очередь, платформы ARM).

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. New 64-bit Linux Rootkit Doing iFrame Injections. – URL: https://securelist.com/new-64-bit-linux-rootkit-doing-iframe-injections-30/34623/ (дата обращения: 28.04.2022).

2. Skidmap Linux Malware Uses Rootkit Capabilities to Hide Cryptocurrency-Mining Payload. – URL: https://blog.trendmicro.com/trendlabs-security-intelligence/skidmap-linux-malware-uses-rootkit-capabilities-to-hide-cryptocurrency-mining-payload/ (дата обращения: 28.04.2022).

3. LKRG – Linux Kernel Runtime Guard. – URL: https://www.openwall.com/lkrg/ (дата обращения: 28.04.2022).

4. 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.

5. 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

6. 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

7. 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.

8. 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.

9. 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

10. 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

11. 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

12. Поддубный, В.А. Средство обнаружения скрытого исполнимого кода в памяти ОС Windows / В.А. Поддубный, И.Ю. Коркин // Вопросы кибербезопасности. 2019. № 5 (33). С. 75- 82. DOI: 10.21681/2311-3456-2019-5-75-82

13. Бланк, Я.А. Граф потока управления / Я.А. Бланк, М.К. Савкин // Наукоемкие технологии в приборо- и машиностроении и развитие инновационной деятельности в вузе: материалы региональной научно-технической конференции, 19-21 апреля 2016 г. Т. 3. Калуга: Издательство МГТУ им. Н. Э. Баумана, 2016. 204 с.

14. 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: 28.04.2022).

15. Репозиторий библиотеки SharpDisasm. – URL: https://github.com/spazzarama/SharpDisasm (дата обращения: 28.04.2022).

16. How to dump process memory in Linux. – URL: https://davidebove.com/blog/2021/03/27/how-to-dump-process-memory-in-linux/ (дата обращения: 28.04.2022).