УДК 004.492.2

Способ предотвращения атак на динамические структуры файловой подсистемы ядра ОС на базе Linux

О. А. Казаков[1]*

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

Аннотация. Цель работы – предотвращение атак на динамически выделяемые структуры файловой подсистемы в памяти ядра ОС семейства Linux. Работа затрагивает проблему разделения доступа к ресурсам памяти ядра ОС на базе Linux. В работе показано, что конфиденциальность пользовательских данных может быть нарушена путем реализации атак на динамические структуры файловой подсистемы ядра, среди существующих средств защиты нет таких, которые способны защищать конфиденциальные данные пользователя от таких атак. Работа с любым файлом данных в Linux начинается с системного вызова открытия файла, в котором происходит проверка прав пользователя на доступ к файлу. Однако последующие вызовы чтения или записи доверяют результатам системного вызова открытия, которые могут быть изменены злоумышленником. В работе предлагается ввести дополнительную защиту системных вызовов, позволяющих взаимодействовать с файловой подсистемой. Предложенный способ защиты является архитектурно независимым, а также не требует внесения изменений в исходный код ядра Linux.

Ключевые слова: Linux, динамические структуры ядра, защита конфиденциальности, разделение доступа

A Way to prevent attacks on the dynamic structures of the file subsystem of the Linux-based OS kernel

O. A. Kazakov[1]*

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

Abstract. The purpose of the work is to prevent attacks on dynamically allocated structures of the file subsystem in the memory of the kernel of the Linux OS family. The work touches upon the problem 0of sharing access to memory resources of the Linux-based OS kernel. The paper shows that the confidentiality of user data can be violated by implementing attacks on the dynamic structures of the kernel file subsystem, among the existing there are no protection tools capable of protecting confidential user data from such attacks. Working with any data file in Linux begins with the open file system call, which checks the user's access rights to the file. However, subsequent read or write calls trust the results of the open system call, which can be modified by an attacker. The paper proposes to introduce additional protection for system calls that allow interacting with the file subsystem. The proposed protection method is architecturally independent and does not require changes to the source code of the Linux kernel.

Keywords: Linux, dynamic data structures, confidentiality protection, access control

Введение

Ядро операционной системы – одна из приоритетных целей злоумышленников, поскольку получение доступа к ядру операционной системы предоставляет почти неограниченный доступ ко всей операционной системе, в том числе данным

пользователей. Злоумышленники придумывают все новые способы проведения атак на ядро операционной системы [1]. Существует такой тип атак, как атаки на динамические структуры ядра. Такие атаки в отношении динамических структур файловой подсистемы ядра могут повлечь нарушение конфиденциальности пользовательских данных, что может повлечь финансовый и репутационный ущерб.

Существующие средства защиты ядра [2], включающие в себя ограничения на доступ к различным областям памяти ядра, ограничения доступа к системным вызовам, обеспечение целостности некоторых структур ядра, перемешивание адресов структур ядра не могут предотвратить рассматриваемые атаки.

Защита динамических структур является задачей гораздо более сложной, чем защита статических структур: данные в таких структурах постоянно и легальным образом изменяются, что означает, что проверки целостности или запрет на изменение, успешно используемые для защиты статических данных, не применимы. Подход, при котором все динамические данные непрерывно записываются, и, например, по установленным правилам блокируется или разрешается конкретное действие, является ресурсоемким настолько, что для защиты системы будет использоваться большая часть вычислительных ресурсов системы.

В работах [3, 4, 5, 6, 7] непрерывный мониторинг предлагается использовать в сочетании с аппаратными подходами по изоляции чувствительных сущностей ядра ОС семейства Linux. Авторы отмечают, что использовать программно-реализованный мониторинг невозможно из-за слишком большого расходования ресурсов защищаемой системы.

Анализ средств, так или иначе использующих мониторинг для реализации целей защиты, позволяет сделать следующие выводы:

– практически все авторы отмечают, что их идеи (на уровне концепции) могут быть реализованы программно, однако сильно завышенные требования по использованию ресурсов защищаемой машины самим средством защиты делают такой подход неприменимым;

– внедрение дополнительных контролирующих сущностей в целях осуществления мониторинга позволяет существенно снизить количество затрачиваемых ресурсов, однако в то же время и существенно снижается область применения предлагаемого средства защиты;

– авторы решений, использующих мониторинг, отмечают, что при отсутствии внесения некоторых изменений в защищаемую систему средство защиты становится менее эффективным, а необходимость внесения изменений еще больше сужает область применимости предлагаемых решений.

В работах [5, 6, 8, 9, 10, 11] авторы предлагают защищать структуры ядра, используя различные способы на основе аппаратной изоляции. В качестве основных особенностей данного подхода можно выделить следующие:

– возможность создания прозрачности для защищаемой системы – защищаемая система «не знает» о существовании защищающей системы, это означает, что от нее не требуется реализации никакого дополнительного функционала, что делает такую систему очень портируемой (переносимой) и гибкой к изменениям, внесению дополнительного функционала защиты, хотя практика показывает, что

применение данного подхода без внесения изменений в защищаемую систему существенно снижает эффективность средства защиты в отношении количества ресурсов защищаемой системы, направленных на работу средства защиты;

– сниженное потребление ресурсов защищаемой системы в целях защиты – поскольку все задачи защиты возложены на внешнюю систему, то защищаемая система производит существенно меньше действий для защиты по сравнению с программной реализацией той же защитной идеи;

– необходимость наличия контролирующей вышестоящей сущности.

Системы защиты, построенные принципу аппаратной изоляции, являются перспективными и с точки зрения их функциональности, и с точки зрения влияния на производительности защищаемой системы, однако их применимость сильно ограничена.

На уровне концепции любой из методов, реализованных в рассмотренных выше работах быть реализован программно. Однако необходимость использования большого количества системных ресурсов при программной реализации затрудняет использование программных средств и является основным недостатком применения этого подхода.

Однако этот способ, в отличие от перечисленных ранее, позволяет получить следующие преимущества:

– возможность использования средства защиты без необходимости наличия контролирующей сущности;

– независимость от архитектуры аппаратного обеспечения — подход является переносимым, его реализация не зависит (в случае необходимости) от архитектуры машины, на которой планируется запускать средство защиты;

– имеется возможность реализовать средство защиты без внесения изменений в ядро защищаемой системы.

Существует несколько основных подходов к обнаружению атак на структуры ядра (в том числе динамические), на основе которых разработаны средства защиты от различных атак на структуры ядра. Однако в следующем разделе будет показан пример атаки, которую не сможет предотвратить ни одно из существующих средств защиты.

Сценарий атаки на динамические структуры файловой системы

Сценарий атаки приведен на рисунке 1.

Атака происходит следующим образом:

– вредоносный загружаемый модуль ядра отыскивает процесс, в рамках которого легальный пользователь взаимодействует с целевым файлом;

– вредоносный загружаемый модуль ядра отыскивает процесс, в рамках которого нарушитель взаимодействует и доступным для него файлом;

– вредоносный загружаемый модуль ядра отыскивает структуру, которая соответствует открытому целевому файлу, к которому злоумышленник хочет получить доступ;

– вредоносный загружаемый модуль ядра отыскивает структуру, которая соответствует открытому файлу, к которому злоумышленник имеет доступ;

– вредоносный загружаемый модуль ядра подменяет адрес структуры файла, к которому злоумышленник имеет доступ, на адрес структуры, к которой имеет доступ легальный пользователь.

После перечисленных шагов злоумышленник может совершать операции чтения и записи с файлом легального пользователя, хотя прав на доступ к этому файлу у него нет.

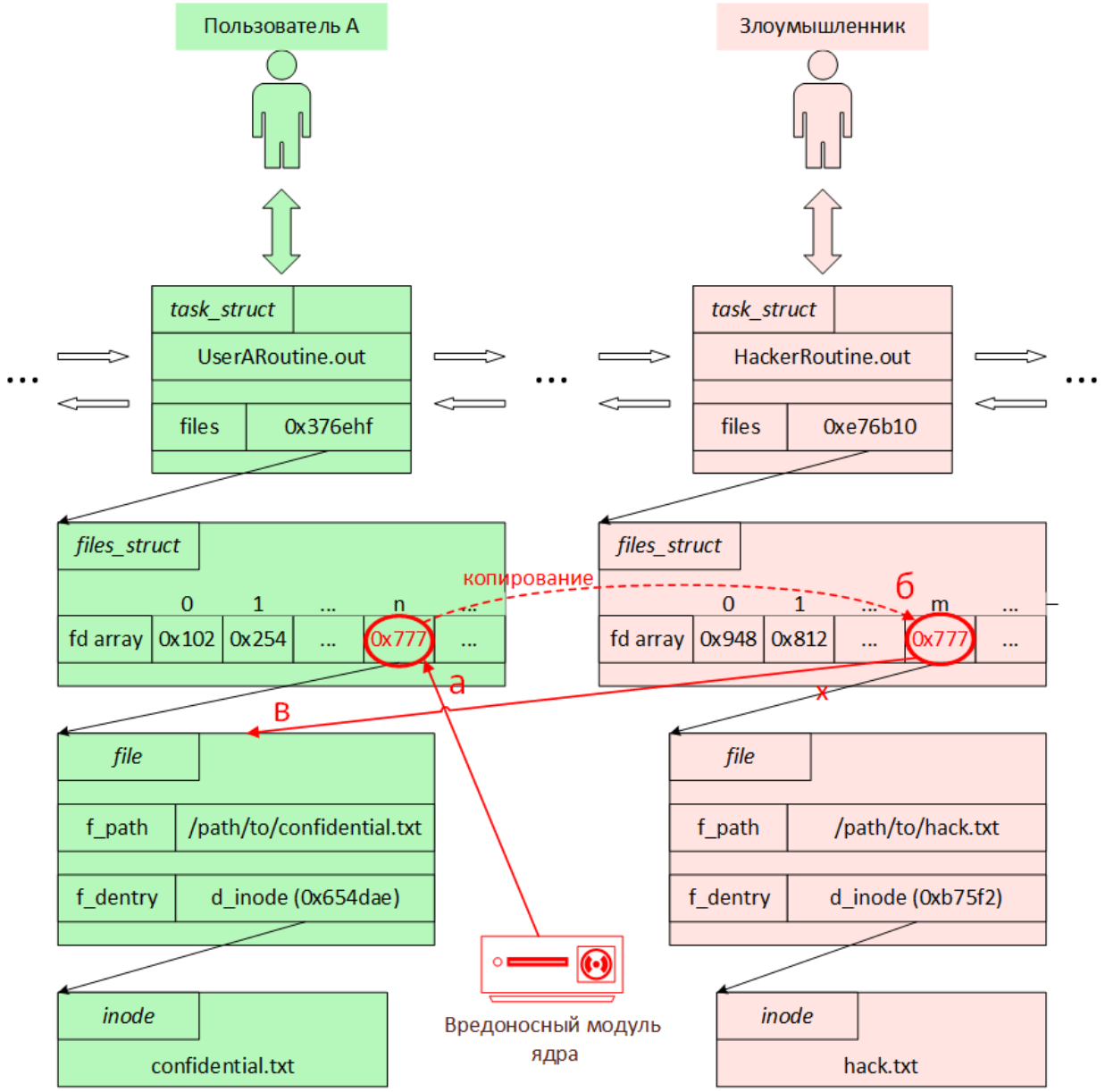

Рис. 1. Сценарий атаки на динамические структуры файловой системы

Предлагаемый способ предотвращения

В соответствии с процессом взаимодействия ядра с файловой системой основными этапами взаимодействия является использование системных вызовов *openat*, *read* и *close*. Поэтому для защиты данных необходимо вмешаться в логику работы ядра, создав дополнительную абстракцию, которая позволит запретить

доступ к структурам типа *file* тем процессам, которые этот файл не открывали. Предлагается предоставить пользователю указать файлы, которые он хотел бы защищать, а затем применять алгоритм, который разделен на три этапа в соответствии с логикой работы операционной системы с файлами. В процессе описания алгоритма будет использоваться понятие таблицы защищаемых файлов, содержащей следующие поля:

– идентификатор процесса, использующего защищаемый файл;

– полное имя (путь) защищаемого файла;

– номер файлового дескриптора, соответствующего защищаемому файлу в указанном процессе;

– физический (реальный) адрес структуры типа file, соответствующий защищаемому файлу (файловому дескриптору);

– адрес-приманка структуры типа file, соответствующий защищаемому файлу (файловому дескриптору).

Связка идентификатор процесса и идентификатор файла полностью определяет открытый файл, то есть не может быть двух разных строк в таблице, где эти два поля совпадают.

Сценарий защиты системного вызова *openat* приведен на рисунке 2.

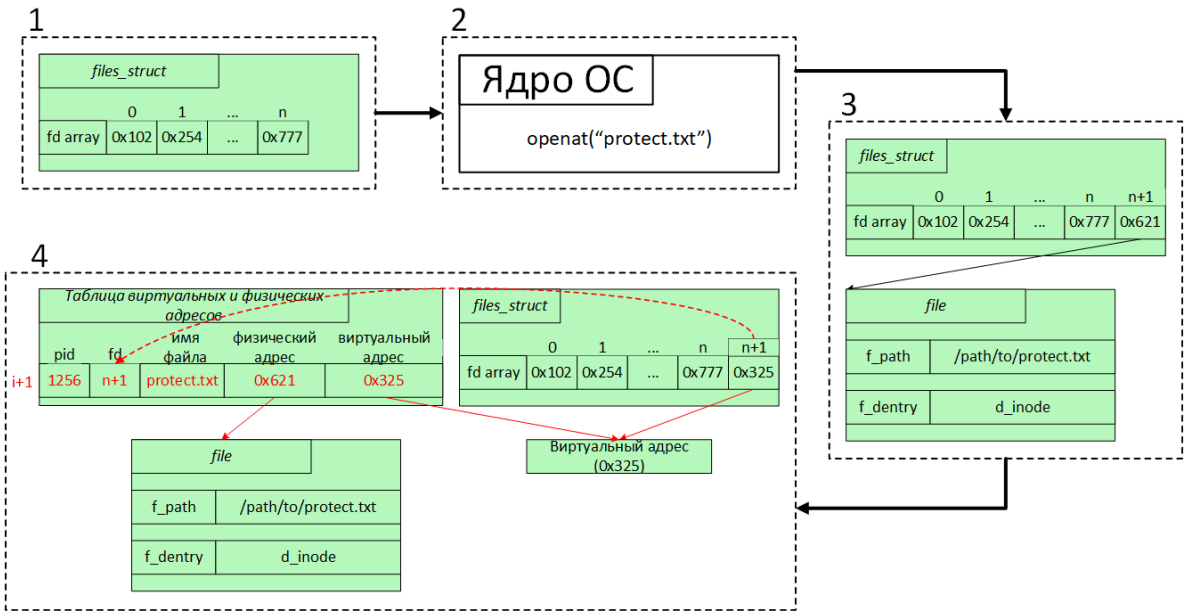

Рис. 2. Сценарий защиты системного вызова *openat*

Для защиты данных открываемого файла предлагается выполнить следующий алгоритм:

– получить идентификатор текущего процесса;

– извлечь из аргументов системного вызова имя открываемого файла, сравнить его с именем защищаемого файла;

– открыть файл штатными средствами ядра без изменения передаваемых аргументов, то есть непосредственно совершить системный вызов openat;

– если открываемый файл не является защищаемым или системный вызов openat завершился с ошибкой, то завершить работу алгоритма с кодом возврата системного вызова openat;

– получить файловый дескриптор для открытого файла (по имени файла, которое содержится в поле f_path структуры file) путем поиска имени защищаемого файла во всех структурах открытых данным процессом файлов;

– получить физический адрес структуры file, соответствующей защищаемому файлу в рамках текущего процесса;

– сгенерировать адрес-приманку структуры file, соответствующей защищаемому файлу в рамках текущего процесса;

– добавить в таблицу защищаемых файлов новую строку и внести все полученные данные в соответствующие поля этой строки;

– поменять физический адрес структуры file, соответствующей защищаемому файлу, на адрес-приманку;

– завершить работу алгоритма с кодом возврата системного вызова openat.

Стоит обратить внимание на то, что после описанного алгоритма злоумышленник не сможет получить доступ к данным защищаемого файла по сценарию атаки из предыдущего раздела, поскольку в его распоряжении окажется адрес-приманка, не указывающий на реальные данные защищаемого файла. Однако оказывается, что и процесс легального пользователя в таком случае не сможет получить данные файла, а также закрыть файл, поэтому необходимо внести изменения и в порядок совершения системных вызовов *read* и *close*.

Для защищенной работы системного вызова *read* необходимо выполнить следующий алгоритм:

– получить идентификатор текущего процесса;

– извлечь из аргументов системного вызова файловый дескриптор;

– проверить наличие связки идентификатор процесса и файловый дескриптор в таблице виртуальных адресов;

– в случае отсутствия вызвать системный вызов read без внесения каких-либо изменений и завершить работу с кодом возврата системного вызова;

– заменить адрес-приманку соответствующей структуры file на реальный адрес, взятый из соответствующей строки таблицы защищаемых файлов;

– совершить системный вызов read;

– заменить физический адрес соответствующей структуры file на адрес-приманку, взятый из соответствующей строки таблицы виртуальных адресов;

– завершить работу алгоритма с кодом возврата системного вызова.

Для корректной работы системного вызова *close*, в том числе корректного закрытия защищаемого файла после совершения безопасных операций с ним, необходимо выполнить следующий алгоритм (рисунок 4).

– получить идентификатор текущего процесса;

– извлечь из аргументов системного вызова файловый дескриптор;

– проверить наличие связки идентификатор процесса и файловый дескриптор в таблице защищаемых файлов;

– в случае отсутствия вызвать системный вызов close без внесения каких-либо изменений и завершить работу с кодом возврата системного вызова;

– заменить адрес-приманку соответствующей структуры file на реальный адрес, взятый из соответствующей строки таблицы защищаемых файлов;

– совершить системный вызов close;

– освободить адрес-приманку для соответствующей строки таблицы защищаемых файлов;

– удалить соответствующую строку из таблицы защищаемых файлов;

– завершить работу алгоритма с кодом возврата системного вызова close.

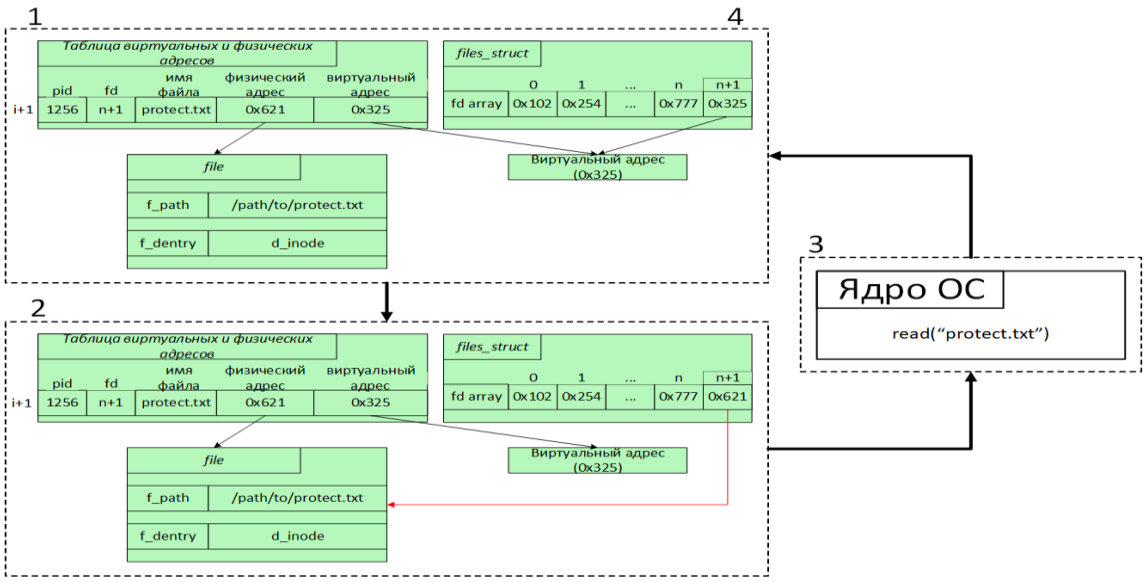

Рис. 3. Сценарий защиты системного вызова *read*

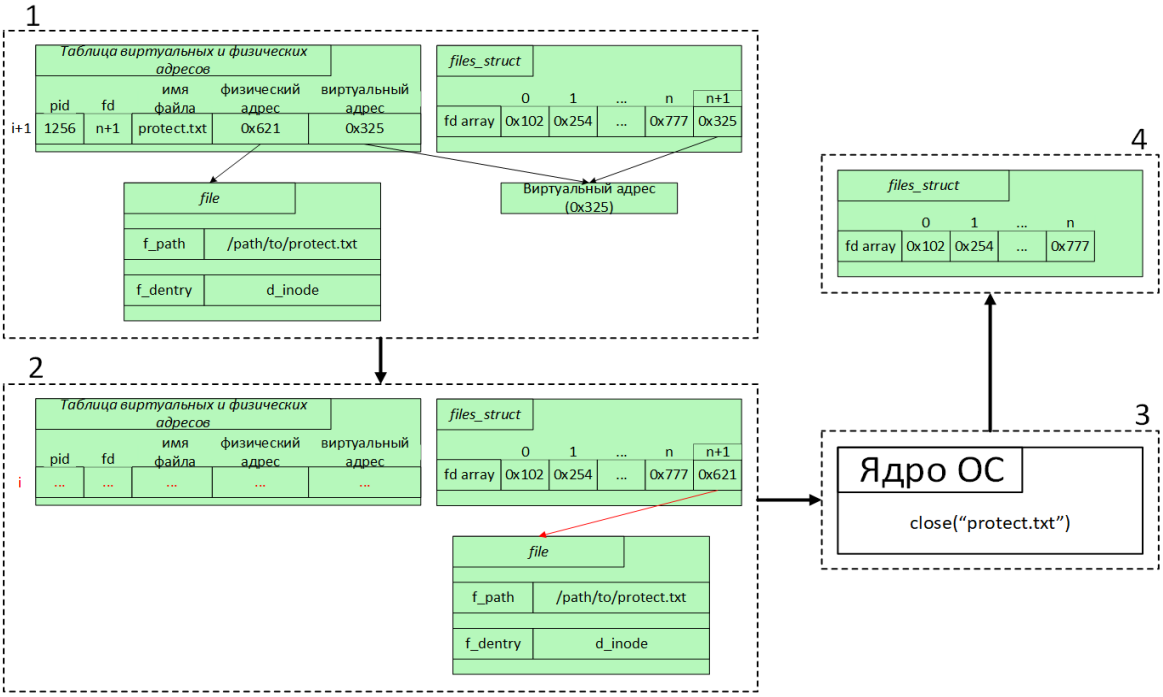

Рис. 4. Сценарий защиты системного вызова *close*

Заключение

В данной работе был приведен анализ существующих средств (как встроенных, так и сторонних) защиты структур ядра ОС на базе Linux. Предложен сценарий атаки, которому не может противостоять ни одно из существующих средств, а результатом реализации сценария является нарушение конфиденциальности данных пользователя.

Проведен анализ существующих сторонних средств защиты от атак на (динамические) структуры ядра ОС на базе Linux. Проанализированы основные плюсы и минусы. Выбран подход, который позволит создать наиболее универсальное средство защиты в контексте отсутствия ограничений применимости к различным защищаемым системам. Предложен способ защиты, который возможно реализовать на основе выбранного подхода.

Дальнейшее направление работ – анализ способов возможного обхода разработанного средства защиты, анализ возможных вариантов предотвращения обхода.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. J. Xiao, H. Huang, H. Wang Kernel Data Attack Is a Realistic Security Threat [Text] / // International Conference on Security and Privacy in Communication Systems. 2015.

2. The Linux Kernel Security Documentation: Kernel Self-Protection [Электронный ресурс]. URL: https://www.kernel.org/doc/html/latest/security/self-protection.html (дата обращения: 28.04.2022).

3. M.Hicks, N.L.Petroni Jr. Automated Detection of Persistent Kernel Control-Flow Attacks // Proceedings of the 2007 ACM Conference on Computer and Communications Security. 2007.

4. J.Rhee, R.Riley, D.Xu, X.Jiang Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring // Proceedings of 2009 International Conference on Availability, Reliability and Security. 2009.

5. M.Graziano, L.Flore, A.Lanzi, D.Balzarotti Subverting Operating System Properties Through Evolutionary DKOM Attacks // Proceedings of 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. 2016.

6. LKIM: The Linux Kernel Integrity Measurer [Электронный ресурс]. URL: https://www.jhuapl.edu/Content/techdigest/pdf/V32-N02/32-02-Pendergrass-McGill.pdf (дата обращения: 20.04.2022).

7. A. Baliga, V. Ganapathy, L. Iftode Detecting Kernel-Level Rootkits Using Data Structure Invariants // IEEE Transactions on Dependable and Secure Computing. 2011. Vol. 8. P. 670-684. DOI: 10.1109/TDSC.2010.38

8. Korkin, I. Memory Ranger Prevents Hijacking FILE_OBJECT Structures in Windows Kernel // Proceedings of the 14th annual Conference on Digital Forensics, Security and Law. 2019.

9. Kernel Data Integrity Protection via Memory Access Control [Электронный ресурс]. URL: https://smartech.gatech.edu/bitstream/handle/1853/30785/GT-CS-09-04.pdf (дата обращения: 20.04.2022).

10. Enforcing Kernel Security Invariants with Data Flow Integrity [Электронный ресурс]. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/enforcing-kernal-security-invariants-data-flow-integrity.pdf (дата обращения: 20.04.2022).

11. Korkin, I. Hypervisor-Based Active Data Protection for Integrity and Confidentiality of Dynamically Allocated Memory in Windows Kernel // Proceedings of the 13th annual Conference on Digital Forensics, Security and Law (CDFSL). 2018.