**DOI: 10.15514/ISPRAS-2025-37(3)-23**



# Обзор механизмов усиления защищенности операционных систем и пользовательских приложений

[1] *Д.В. Ефремов, ORCID: 0000-0002-9916-056X <efremov@ispras.ru>*
[1,2,3] *А.К. Петренко, ORCID: 0000-0001-7411-3831 <petrenko@ispras.ru>*
[1,3,4] *Б.А. Позин, ORCID: 0000-0002-0012-2230 <bpozin@ec-leasing.ru>*
[1,5] *В.А. Семенов, ORCID: 0000-0002-8766-8454 <sem@ispras.ru>*

[1] *Институт системного программирования РАН,*
*Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Московский государственный университет имени М.В. Ломоносова,*
*Россия, 119991, г. Москва, Ленинские горы, д. 1.*

[3] *НИУ Высшая школа экономики,*
*Россия, 101978, г. Москва, ул. Мясницкая, д. 20.*

[4] *ЗАО ЕС-лизинг, Россия, 117405, Москва, Варшавское шоссе, д. 125.*

[5] *Московский физико-технический институт,*
*Россия, 141701, Московская область, г. Долгопрудный, Институтский пер., 9.*

**Аннотация.** В данной работе представлен систематический обзор механизмов усиления защищенности (hardening) операционных систем и пользовательских приложений. Рассматриваются различные типы защитных механизмов, включая механизмы защиты памяти, защиты аппаратного стека, защиты динамической памяти, рандомизация адресного пространства, защита потока управления и защита целостности системы. Детально анализируются принципы работы данных механизмов, их эффективность и влияние на производительность систем. Особое внимание уделяется реализации защитных механизмов в современных операционных системах, в частности, в ядре Linux. Работа предназначена для специалистов в области информационной безопасности, разработчиков операционных систем и исследователей, занимающихся вопросами защиты информации.

**Ключевые слова:** защита информации; операционные системы; безопасность приложений; защита памяти; контроль целостности; изоляция; усиление защищенности

**Для цитирования:** Ефремов Д.В., Петренко А.К., Позин Б.А., Семенов В.А. Обзор механизмов усиления защищенности операционных систем и пользовательских приложений. Труды ИСП РАН, том 37, вып. 3, 2025 г., стр. 325-354. DOI: 10.15514/ISPRAS–2023–37(3)-23.

**Благодарности:** Работа поддержана компанией “Лаборатория Касперского” в рамках проекта “Анализ мирового уровня техники по архитектурным средствам обеспечения доверия”.

# Overview of Hardening Mechanisms in Operating Systems and User Applications

[1] *D.V. Efremov, ORCID: 0000-0002-9916-056X <efremov@ispras.ru>*
[1,2,3] *A.K. Petrenko, ORCID: 0000-0001-7411-3831 <petrenko@ispras.ru>*
[1,3,4] *B.A. Pozin, ORCID: 0000-0002-0012-2230 <bpozin@ec-leasing.ru>*
[1,5] *V.A. Semenov, ORCID: 0000-0002-8766-8454 <sem@ispras.ru>*

[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

[3] *National Research University, Higher School of Economics, 20, Myasnitskaya ulitsa, Moscow, 101978, Russia.*

[4] *"EC-leasing" Co., 125 Varshavskoye shosse, Moscow, 117405, Russia.*

[5] *Moscow Institute of Physics and Technology, 9, Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.*

**Abstract.** This paper presents a systematic review of hardening mechanisms for operating systems and user applications. Various types of protection mechanisms are discussed, including memory protection mechanisms, hardware stack protection, dynamic memory protection, address space randomization, control flow protection, and system integrity protection. The principles of these mechanisms, their effectiveness, and their impact on system performance are analyzed in detail. Special attention is given to the implementation of protective mechanisms in modern operating systems, particularly in the Linux kernel. This work is intended for information security specialists, operating system developers, and researchers working on information security issues.

**Keywords:** information security; operating systems; application security; memory protection; integrity control; isolation; security hardening; hardening.

**For citation:** Efremov D.V., Petrenko A.K., Pozin B.A., Semenov V.A. Overview of Hardening Mechanisms in Operating Systems and User Applications. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 3, 2025., pp. 325-354. DOI: 10.15514/ISPRAS–2023–37(3)-23.

**Acknowledgements.** The work was supported by Kaspersky Lab as a part of the project "Analysis of world-class technology in architectural means of ensuring trust".

## *1. Введение*

Механизмы усиления защищенности (hardening) представляют собой комплекс технологий и методов, направленных на минимизацию рисков эксплуатации уязвимостей в программных системах. В условиях постоянно растущего числа и сложности кибератак, внедрение таких механизмов становится критически важным. Объектом внимания мероприятий, нацеленных на усиление защищенности, является как программный код, так и описания конфигураций собственно программных компонентов системы и сетевой составляющей. Иногда в тематику защищенности включают и вопросы контроля за полномочиями персонала. В данной статье мы ограничиваемся вопросами усиления защиты программного кода.

Усиление защищенности применяется к коду, который потенциально может подвергаться попыткам компрометации – коду, работающему с ограниченными ресурсами или имеющему доступ к конфиденциальной информации. Часто это низкоуровневый код операционных систем (ОС) или системных программ, от безопасности которых зависит защищенность всей вычислительной системы.

Современные кибератаки характеризуются значительным усложнением техник эксплуатации, что требует применения комплексных подходов к защите. Эффективное противодействие таким атакам возможно только при взаимодействии исследователей безопасности, разработчиков ОС, компиляторов и производителей аппаратного обеспечения. То есть усиление защищенности как система подразумевает рассмотрение разных аспектов конструирования программной системы и видов взаимодействия различных компонентов (например, межпроцессорные взаимодействия, связи в многопоточных программах, сетевые взаимодействия). Совокупность прямых и опосредованных связей в современной компьютерной системе крайне сложна, поэтому высока роль анализа архитектуры системы на различных уровнях детализации и привлечение методов архитектурного моделирования и анализа на всех фазах жизненного цикла вычислительной системы.

Комплексный подход к усилению защищенности методически существенно отличается от практик, которые ранее так или иначе использовались специалистами по информационной безопасности и администраторами компьютерных систем. Если старые методы в большей степени придерживались тактики реагирования на вновь обнаруженные уязвимости, то новая методология рассматривает усиление защищенности как планомерный процесс, в котором всегда предусматривается возможность появления проблем информационной безопасности и в проактивной форме реализуются меры по их нейтрализации. Важным инструментом планирования такой проактивной деятельности также является архитектурных анализ. Отметим, что следуя принципам конструктивной безопасности, заботиться о защищенности системы нужно начиная с фазы ее замысла, однако при этом тогда, когда система уже создана и эксплуатируется, тоже необходимо, возвращаться к проблемам, которые рассматривались в ходе ее проектирования и разработки, и иногда проводить ретроспективный анализ вплоть до фазы замысла. Усиление защищенности, один из ориентиров, которые помогает выстроить систематический процесс анализа и постоянного улучшения в соответствии с принципами конструктивной информационной безопасности.

Мы можем увидеть результаты планомерного архитектурного подхода к усилению защиты комплексного программного обеспечения на примере проектов OpenBSD [1], Linux Kernel Self-Protection Project (KSPP) [2], Qubes OS [3], GrSecurity [4], Pax Project [5], GrapheneOS [6] и других.

В данной работе рассматриваются основные категории механизмов усиления защищенности, их принципы работы и реализация в современном программном обеспечении, а также перспективы их развития. Мы намеренно ограничиваемся обзором механизмов усиления, которые уже внедрены в большие открытые проекты и применимы в продуктовых сборках, а не только в качестве средства отладки. Существует большое количество исследовательских работ по механизмам усиления. Однако, их реальное внедрение в проект может быть сильно осложнено проблемами совместимости с существующим кодом, сложностью самого внедрения и ее последующей поддержкой, а также иными моментами, упущенными из рассмотрения в исходных исследовательских работах. Мы рассматриваем факты внедрения защиты в открытые проекты и одобрение со стороны сообщества разработчиков как фильтр, доказывающий на практике высокую ценность технологии и её зрелость.

## *2. Систематизация механизмов усиления защищенности*

В современной парадигме информационной безопасности механизмы усиления защищенности (hardening) представляют собой фундаментальный комплекс мер, направленных на минимизацию поверхности атаки и усиления защитных свойств как ОС, так и пользовательских приложений. Данные механизмы формируют многоуровневую систему защиты, охватывающую все аспекты функционирования программно-аппаратных комплексов: от архитектурного проектирования до этапа эксплуатации. Систематизация данных механизмов позволяет не только структурировать существующие подходы к

обеспечению безопасности, но и выявить их взаимосвязи, что важно при формировании целостной стратегии защиты информационных систем.

## 2.1 Концептуальные основы усиления защищенности

Усиление защищенности как методология базируется на принципе многоуровневой защиты (defense-in-depth), предполагающем выстраивание последовательных барьеров, преодоление каждого из которых требует от потенциального злоумышленника дополнительных ресурсов и компетенций.

Усиление защиты можно рассматривать в контексте разницы между стоимостью её внедрения, использования и увеличением стоимости компроментации программы. Хорошим можно считать такое усиление, которое кратно увеличивает стоимость компроментации программы по отношению к стоимости его внедрения и использования, где под стоимостью следует понимать не только прямое денежное измерение задачи, но также временные затраты и уровень специалистов, необходимых для решения задачи. Так, если простое использование специальных опций сборки программы, автоматом закрывает некоторый широко распространённый класс уязвимостей, то его внедрение требует минимальных усилий и компетенций, а практическим результат выражается в повышенных требованиях к квалификации взломщика, его временных затрат и возможности стабильного взлома программы.

В основе данного подхода лежит понимание того, что абсолютная безопасность недостижима, а эффективная защита должна основываться на комплексном применении разнородных механизмов, компенсирующих недостатки друг друга. Современные механизмы усиления защищенности можно классифицировать на основе разных признаков.

Например, на стадии жизненного цикла программного обеспечения можно выделить механизмы проектирования безопасности (архитектурный анализ, моделирование угроз), разработки безопасного кода (безопасные библиотеки, статический анализ кода), безопасной компиляции и сборки (защиты компилятора), защиты при эксплуатации (динамические средства защиты).

По уровню реализации механизмы усиления можно выделить аппаратные механизмы (NX-бит, защита целостности потока управления на уровне процессора), механизмы на уровне гипервизора/ядра ОС (KASLR, защита стека ядра), механизмы уровня пользовательского пространства (ASLR), прикладные механизмы (защита на уровне приложений). Многие из защит не специфичны для какого-то уровня и могут быть применены на каждом из уровней в не зависимости друг от друга.

Также можно классифицировать виды защиты по типу предотвращаемых угроз. Например, защита от переполнения буфера, защита от инъекций кода, утечек данных, повышения привилегий и др. Классификация самих угроз достаточно обширна и до сих пор пополняется, так как появляются новые вектора атак. Наиболее известная из таких классификаций Common Weakness Enumeration (CWE) [7] представляет собой реестр, созданный организацией MITRE. Реестр представляет собой многоуровневую древовидную структуру, состоящую из типов и определений различных дефектов и недостатков безопасности программного обеспечения, возникающих на стадиях проектирования, разработки, интеграции и эксплуатации информационных систем.

Хорошим примером классификации усилений защиты по типу предотвращаемых угроз может служить карта защитных механизмов ядра Linux [8].

### 2.2 Архитектурный анализ и моделирование угроз

Архитектурный анализ представляет собой этап обеспечения безопасности информационных систем, предшествующий непосредственной реализации защитных механизмов. Данный процесс включает в себя декомпозицию системы на компоненты, анализ их взаимодействия

и выявление потенциальных уязвимостей, возникающих на границах взаимодействия. Результаты архитектурного анализа служат основой для последующего моделирования угроз.

Моделирование угроз [9,10] является методологическим подходом к систематическому выявлению, оценке и приоритизации потенциальных угроз безопасности системы. Наиболее распространенными методологиями моделирования угроз являются Microsoft STRIDE, P.A.S.T.A (Process for Attack Simulation and Threat Analysis) и OCTAVE. Каждая из этих методологий предлагает структурированный подход к идентификации угроз, классифицируя их по различным категориям и связывая с конкретными компонентами анализируемой системы. Процесс моделирования угроз включает следующие этапы:

- Идентификация “активов”, подлежащих защите;
- Документирование архитектуры системы;
- Декомпозиция системы на компоненты;
- Идентификация угроз для каждого компонента;
- Ранжирование угроз по уровню риска;
- Определение стратегий минимизации рисков.

Результаты моделирования угроз напрямую определяют выбор и конфигурацию механизмов усиления защищенности, обеспечивая целенаправленное укрепление наиболее уязвимых аспектов системы.

## 2.3 Анализ рисков информационной безопасности

Анализ рисков представляет собой процесс оценки вероятности реализации угроз и потенциального ущерба от них, что позволяет оптимизировать распределение ресурсов при внедрении механизмов защиты. Современные методологии анализа рисков включают количественные и качественные подходы, такие как FAIR (Factor Analysis of Information Risk) [11], NIST Risk Management Framework (RMF) [12] и ISO 27005 [13]. Ключевыми компонентами анализа рисков являются:

- *Идентификация рисков* – выявление потенциальных сценариев реализации угроз;
- *Оценка вероятности* – определение вероятности реализации каждого сценария;
- *Оценка воздействия* – анализ потенциального ущерба при реализации угроз;
- *Приоритизация рисков* – ранжирование рисков на основе их вероятности и воздействия;
- *Выбор стратегии управления рисками* – принятие, передача, снижение или избежание рисков.

Результаты анализа рисков служат основой для формирования стратегии усиления защищенности, определяя приоритетные направления и оптимальный уровень инвестиций в механизмы зашиты.

Архитектурный анализ информационных систем, в комбинации с методологиями моделирования угроз и формализованными фреймворками анализа рисков, формирует комплексный, многоаспектный подход к идентификации уязвимостей, что непосредственно влияет на реализацию мер по усилению безопасности. В частности, архитектурный анализ создает фундаментальное понимание структуры системы, включая точки взаимодействия компонентов, каналы передачи данных и границы доверия, в то время как моделирование угроз (STRIDE) обеспечивает систематическую идентификацию потенциальных угроз. Данные, полученные в результате этих процессов, подвергаются количественной и качественной оценке посредством методологий анализа рисков, где FAIR предлагает экономическую модель оценки потенциальных потерь, NIST RMF фокусируется на соответствии нормативным требованиям, а ISO 27005 предоставляет структурированный подход к управлению информационными рисками.

Усиление безопасности преобразует выявленные риски и уязвимости в конкретные технические и организационные меры защиты, приоритезированные согласно их потенциальному воздействию на критические активы. Данный интегративный подход позволяет организациям разрабатывать стратегии усиления безопасности, который не только противодействуют идентифицированным угрозам через целенаправленное укрепление уязвимых компонентов архитектуры, но и оптимизируют распределение ресурсов безопасности с учетом экономической эффективности защитных мер относительно стоимости потенциальных инцидентов. Синтез архитектурного анализа, моделирования угроз и оценки рисков создает научно обоснованный фундамент для разработки дифференцированных, контекстуально соответствующих стратегий усиления безопасности, повышая общую устойчивость информационных систем к киберугрозам. Таким образом, hardening переходит из области лучших практик в дисциплинированную реализацию мер, направленных на минимизацию конкретных, формально выявленных и оценённых рисков.

Рассмотрев общий контекст, в который вписывается усиление защищенности приложений, а также с какой точки зрения они могут и должны оцениваться, перейдем к рассмотрению самих технологий. Мы выстраиваем наш обзор различных технологий усиления защищенности ОС и пользовательских приложений от типа защищаемого ресурса, степени зрелости и уровня внедрения технологии. При этом, мы намеренно ограничиваемся защитами на уровне кода самого приложения (как исходного, так и бинарного), оставляя такие темы как защита от зловредных периферийных устройств, системы контроля доступа, антивирусные технологии, системы предотвращения утечек информации, системы обнаружения вторжений (IDS), защиты конечных точек (EDR), и многие другие за пределами нашего обзора.

Рассматриваемые технологии реализуются на уровне загрузчиков, ядер ОС, в системных библиотеках, средах выполнения (runtime), программах пользовательского пространства и компиляторов. Меры усиления построены на принципах рандомизации, проверки границ, изоляции и ограничения времени жизни и доступа, контроля целостности, дублирования вычислений. Архитектура фон Неймана, подразумевает принцип однородности памяти: хранения данных и инструкций в одном разделе памяти. Поэтому мы начинаем наш обзор с механизмов защиты памяти. Защита кода основывается на механизмах защиты памяти и дополняет их, поэтому она продолжает этот обзор. В отдельную категорию мы выносим контроль целостности. Несмотря на то, что сам принцип лежит в основе множества защит, рассматриваемых в прошлых разделах, мы считаем необходимым поместить рассматриваемые технологии в отдельный раздел, так как они не могут быть применены к программам автоматически и планирование их внедрения происходит на архитектурном уровне. По тому же принципу мы выделяем в отдельный раздел механизмы самоограничения процессов.

## *3. Механизмы защиты памяти*

Механизмы защиты памяти обеспечивают контроль над использованием памяти программой и предотвращают эксплуатацию уязвимостей, связанных с нарушением целостности данных или несанкционированным доступом.

В программах можно выделить различные регионы памяти по типу их предназначения и тому, как они используются. Такие как: стек вызовов, динамическая память (куча, heap), кеш и регистры процессора, память для хранения данных и кода программы, регионы памяти для взаимодействия с аппаратурой (MMIO, PMIO, DMA). Атаки на память в общем виде сводятся к несанкционированному чтению из неё и записи в неё. Механизмы защиты в общем виде можно свести к выставлению разрешений на доступ к памяти, проверке границ доступа к памяти, к разделению и изоляции регионов памяти друг от друга, к рандомизации адресов

памяти и смещений в ней, к принудительной инициализации и затиранию регионов памяти, к проверке целостности данных и согласованности структур данных, хранящихся в памяти.

## 3.1 Механизмы защиты памяти

Рассмотрим наиболее известные и распространенные механизмы защиты памяти. Специализированные механизмы для конкретного типа памяти учитывают специфику использования данного типа памяти и векторов атак на него. Как правило, за счет этого они накладывают меньшие накладные расходы на производительность чем общие механизмы защиты. Ограничимся рассмотрением механизмов, которые применяются на практике в ОС со страничной организацией памяти, работающих на вычислительных машинах с архитектурой фон Неймана.

**Рандомизация размещения адресного пространства**

Рандомизация размещения адресного пространства Address Space Layout Randomization (ASLR) [14, 15] представляет собой механизм обеспечения безопасности, основной целью которого является усложнение эксплуатации уязвимостей программного обеспечения, основанных на знании конкретных адресов в памяти. Суть данного подхода заключается в случайном изменении расположения ключевых областей данных и кода в виртуальном адресном пространстве процесса при каждом его запуске.

В контексте ядра ОС этот механизм известен как Kernel Address Space Layout Randomization (KASLR) [16] в Linux. KASLR предназначен для рандомизации адресов, по которым загружается код самого ядра и размещаются его структуры данных. KASLR инициируется на этапе загрузки системы.я Этот процесс включает генерацию случайного значения смещения в пределах допустимого диапазона адресов, последующее перемещение сегментов ядра в соответствии с этим смещением и обновление системных структур управления памятью, таких как таблицы страниц и дескрипторы памяти, для отражения нового, рандомизированного расположения. При успешной реализации KASLR увеличивает сложность выполнения атак, требующих точного знания адресов в памяти ядра, тем самым затрудняя обход защитных механизмов или использование уязвимостей, зависящих от статических адресов.

Для эффективного применения принципов ASLR используется технология создания исполняемых файлов, независимых от позиции загрузки (Position Independent Executables, PIE). PIE позволяет ОС загружать код и данные программы в произвольные участки виртуальной памяти. Это достигается за счет того, что исполняемый файл генерируется с использованием относительных адресов вместо абсолютных. При загрузке такой программы ОС, применяя механизмы ASLR, определяет случайные базовые адреса для ее сегментов (кода, данных, стека, кучи). Все внутренние ссылки в программе затем пересчитываются во время выполнения относительно этих динамически определенных базовых адресов.

Слабым местом защиты является то, что смещения в структурах между полями, между функциями, между сегментами памяти является константным. Имея собранную программу, злоумышленник может вычислить эти смещения для использования в эксплоите. Если в ядре или программе не применяется дополнительных техник противодействия утечкам информации, которые мы рассмотрим далее, то злоумышленнику достаточно одной такой ошибки, чтобы узнать адрес и имея предварительно вычисленные смещения обойти защиту ASLR.

В OpenBSD для решения проблемы постоянства смещений, упомянутой выше, реализована защита Kernel Address Randomized Link (KARL) [17]. В отличие от KASLR, который рандомизирует только базовый адрес загрузки ядра, KARL при каждой загрузке системы заново компонует ядро. В процессе этой перекомпоновки объектные файлы, составляющие ядро, связываются в случайном порядке. Это приводит к тому, что относительные смещения между различными функциями и структурами данных внутри самого ядра становятся непредсказуемыми от загрузки к загрузке. Защита KARL значительно усложняет

эксплуатацию уязвимостей, основанных на знании фиксированных смещений внутри ядра. Это делает эксплойты, полагающиеся на такие смещения, непереносимыми между перезагрузками одной и той же системы.

**Политика W^X (Write XOR Execute)**

Политика “запись, ЛИБО выполнение (W^X)” или запрета выполнения данных (DEP) воплощает принцип разделения данных и кода. Согласно этой политике, страница памяти может быть либо доступна для записи, либо для выполнения, но никогда одновременно и для того, и для другого. Этот подход усложняет эксплуатацию уязвимостей, поскольку даже если атакующему удастся записать вредоносный код в память, он не сможет его выполнить, и если он найдет способ выполнить код из какой-либо области, он не сможет ее модифицировать.

Реализация опирается на аппаратную поддержку, предоставляемую современными процессорами (например, бит NX – No-eXecute у компании AMD, или XD – Execute Disable у компании Intel), и ядро ОС, которое соответствующим образом настраивает атрибуты страниц памяти [18-20].

Защита W^X применяется не только к программам пользовательского пространства. Подобная защита может быть реализована и в ядре для защиты его кода [21-22]. Этот же принцип реализуется в механизме NX-physmap [23], который представляет собой усиление безопасности ядра ОС, направленную на предотвращение исполнения кода из области прямого отображения физической памяти (physmap).

**Запрет отображение памяти в нижней части адресного пространства процесса**

Механизм MMAP_MIN_ADDR [24] предотвращает отображения памяти в наиболее низких адресах виртуального адресного пространства процесса. Ключевая задача данного механизма – исключить возможность для процессов создавать отображения, начинающиеся с нулевого адреса, что исторически являлось вектором атак на ядро Linux.

Уязвимость, которую устраняет MMAP_MIN_ADDR, заключалась в том, что разыменования нулевого указателя (NULL pointer dereference) в коде ядра могла быть эксплуатирована злонамеренным процессом. Такой процесс мог предварительно отобразить контролируемую им страницу памяти по нулевому адресу, разместить на ней вредоносный код и затем спровоцировать ошибку в ядре. Это приводило к тому, что ядро, пытаясь обратиться по нулевому адресу, фактически передавало управление вредоносному коду, который исполнялся с привилегиями ядра, компрометируя всю систему.

Для противодействия описанной угрозе параметр ядра MMAP_MIN_ADDR устанавливает минимально допустимый адрес, с которого процессу разрешено запрашивать выделение памяти посредством системного вызова `mmap()`. Любые попытки отобразить память по адресу ниже этого установленного порога отклоняются ядром.

**Аппаратные защиты SMAP и SMEP разграничения адресных пространств пользователя и ядра**

Механизмы предотвращения доступа в режиме супервизора (Supervisor Mode Access Prevention, SMAP) [25] и предотвращения исполнения в режиме супервизора (Supervisor Mode Execution Prevention, SMEP) [26] представляют собой аппаратные функции безопасности, реализованные в современных процессорах, предназначенные для усиления изоляции между пространством ядра (режим супервизора) и пространством пользователя. Эти механизмы направлены на предотвращение эксплуатации уязвимостей ядра, которые могли бы позволить атакующему получить контроль над системой.

SMEP предотвращает исполнение кода, находящегося на страницах памяти, принадлежащих пространству пользователя, когда процессор работает в режиме ядра (режим супервизора). Если ядро, работая с наивысшими привилегиями, пытается выполнить инструкцию, расположенную в памяти, помеченной как пользовательская (например, из-за уязвимости, позволяющей атакующему контролировать указатель инструкций ядра), процессор генерирует исключение. Это значительно усложняет классические атаки, при которых

злоумышленник размещает вредоносный код в пользовательском пространстве и затем обманом заставляет ядро перейти на его исполнение. Без SMEP, если атакующий мог перенаправить поток выполнения ядра на адрес в пользовательском пространстве, то он выполнить произвольный код с привилегиями ядра. SMEP блокирует такую возможность на аппаратном уровне, требуя, чтобы код, исполняемый ядром, находился исключительно в памяти, принадлежащей ядру [27].

SMAP расширяет защиту, предоставляемую SMEP, предотвращая произвольный доступ (чтение или запись) ядра к данным, находящимся на страницах памяти пространства пользователя, если такой доступ не разрешен явно. По умолчанию, любая попытка ядра прочитать или записать данные в пользовательском пространстве (за исключением стека инструкций, который покрывается SMEP) приведет к генерации исключения. Для легитимного обмена данными между ядром и пространством пользователя (например, при системных вызовах с использованием функций `copy_to_user` или `copy_from_user` в Linux) существуют специальные механизмы, позволяющие временно и контролируемо обойти ограничения SMAP [28].

SMEP и SMAP совместно создают барьер, затрудняющий эксплуатацию многих типов ошибок в ядрах ОС. SMEP защищает от прямого исполнения пользовательского кода ядром, а SMAP – от несанкционированного доступа ядра к пользовательским данным. Эти механизмы значительно повышают сложность взлома для атакующих. Включение SMEP и SMAP является стандартной практикой в современных ОС для усиления их безопасности и снижения поверхности атаки на ядро.

**Рандомизация полей структур при сборке**

Механизм рандомизации полей структур, известный как RANDSTRUCT [29,30], представляет собой технику усиления защиты ядра ОС, направленную на усложнение эксплуатации уязвимостей, связанных с повреждением памяти и утечкой информации. Основная цель RANDSTRUCT заключается в том, чтобы сделать непредсказуемым внутреннее расположение (смещения) полей в критически важных структурах данных ядра.

Принцип действия RANDSTRUCT основан на изменении порядка следования полей внутри структур ядра на этапе сборки. Это достигается с помощью специального плагина для компилятора GCC, который случайным образом переставляет поля в структурах, помеченных для рандомизации, при каждой сборке ядра. Таким образом, смещения полей относительно начала структуры становятся уникальными для каждой конкретной сборки ядра, что делает невозможным создание эксплойтов, полагающихся на фиксированное расположение полей.

Внедрение RANDSTRUCT затрудняет атаки, основанные на знании точного расположения полей в структурах ядра. Злоумышленники часто полагаются на фиксированные смещения для чтения или перезаписи определенных данных внутри структур с целью повышения привилегий или обхода защитных механизмов. Рандомизация делает такие атаки ненадежными, поскольку смещения полей становятся неизвестными заранее. Также RANDSTRUCT снижает вероятность успешной эксплуатации уязвимостей, приводящих к утечке информации, так как атакующему сложнее определить, какие именно данные он считывает из памяти. Этот механизм эффективно дополняет другие техники усиления защиты, такие как KASLR.

Необходимо отметить, что RANDSTRUCT является защитой времени сборки. Если злоумышленнику удастся получить доступ к конкретному собранному образу ядра, он сможет определить актуальное расположение полей. Эта защита не имеет смысла, если собранные образы ядра выкладываются публично. Однако, RANDSTRUCT значительно повышает барьер для разработки универсальных эксплойтов, работающих на различных сборках ядра, и усложняет анализ ядра для поиска уязвимостей.

**Санитайзеры (ASan, KASan, UBSan) и KFENCE**

Санитайзеры представляют собой набор инструментов динамического анализа, предназначенных для обнаружения различных типов ошибок в программах во время их выполнения. К наиболее известным относятся AddressSanitizer (ASan) и UndefinedBehaviorSanitizer (UBSan) [31].

AddressSanitizer (ASan) разработан для выявления ошибок работы с памятью в приложениях пользовательского пространства, таких как использование памяти после освобождения (use-after-free), переполнения буферов на куче или стеке, а также утечки памяти. ASan характеризуется существенными накладными расходами на производительность и увеличением потребления памяти, что обычно исключает его применение в стандартных производственных сборках. KernelAddressSanitizer (KASan) [32] функционирует аналогично ASan, но предназначен для ядра ОС, помогая обнаруживать схожие ошибки доступа к памяти в коде ядра. Как и ASan, KASan значительно снижает производительность и увеличивает потребления памяти, что делает его непригодным для повсеместного использования в производственных версиях ядер. Его основное назначение – отладка и тестирование ядра разработчиками. Однако, падение производительности может быть сведено до приемлемого уровня в случае использования аппаратных механизмов тегирования памяти, что делает использование ASan актуальным для усиления защищенности.

UndefinedBehaviorSanitizer (UBSan) нацелен на обнаружение различных видов неопределенного поведения в программах, например, переполнения знаковых целых чисел, некорректных операций сдвига, использования невыровненных указателей и других подобных ошибок неопределённое поведения. UBSan обладает значительно меньшими накладными расходами по сравнению с ASan и KASan. Некоторые из его проверок могут быть достаточно легковесными для использования в производственных сборках, особенно для критически важных участков кода.

Kernel Electric Fence (KFENCE) [33] представляет собой механизм обнаружения ошибок памяти в ядре Linux, специально спроектированный для использования в производственных системах. В отличие от KASan, KFENCE отличается низкими накладными расходами. Его принцип работы заключается в выделении небольшого процента объектов кучи на специально защищенных страницах памяти. Такой подход позволяет с высокой вероятностью обнаруживать ошибки типа выхода за границы буфера и использование после освобождения для этих выборочных аллокаций. Благодаря своей низкой ресурсоемкости, KFENCE может быть активирован в производственных ядрах для выявления трудноуловимых ошибок памяти без существенного негативного влияния на общую производительность системы.

Memory Tagging Extension (MTE) [34] является аппаратной функцией, реализованной в некоторых современных процессорах архитектуры ARM, и предоставляет механизм для обнаружения ошибок памяти с низкими накладными расходами. MTE позволяет ассоциировать теги как с указателями, так и с областями памяти. При каждом доступе к памяти процессор проверяет соответствие тега указателя тегу области памяти. Несовпадение тегов указывает на потенциальную ошибку, такую как выход за границы выделенной памяти или использование памяти после ее освобождения. Инструменты-санитайзеры, например, Hardware-assisted AddressSanitizer (HWASan) [35], могут использовать MTE для эффективного обнаружения ошибок памяти в производственных условиях, оказывая значительно меньшее влияние на производительность по сравнению с чисто программными реализацией ASan.

**Очистка регистров, используемых вызываемой функцией (Call-used register wiping)**

Механизм очистки регистров, используемых вызываемой функцией (call-used register wiping) [36,37], представляет собой меру по усилению безопасности, направленную на предотвращение утечек конфиденциальной информации через регистры процессора. Основная цель данной техники заключается в минимизации риска того, что данные,

оставшиеся в регистрах после выполнения одной функции, могут быть доступны другой, потенциально контролируемой злоумышленником, функции.

Принцип действия механизма основан на принудительной очистке содержимого регистров, которые, согласно соглашению о вызовах конкретной архитектуры, считаются “используемыми вызываемой функцией”. Это означает, что вызывающая функция не может полагаться на сохранение их значений после вызова другой функции. Очистка обычно происходит либо непосредственно перед возвратом из функции, либо при входе в новую функцию, особенно в критически важных компонентах системы, таких как ядро ОС. Защита реализуется на уровне компилятора.

Внедрение очистки регистров, используемых вызываемой функцией, способствует повышению общей безопасности системы. Во-первых, это значительно снижает поверхность атаки для утечек данных, поскольку чувствительная информация, временно хранящаяся в регистрах, не остается доступной после завершения операции, для которой она была необходима. Во-вторых, данный механизм усложняет эксплуатацию некоторых типов уязвимостей, так как злоумышленник, получивший контроль над потоком выполнения, не сможет легко извлечь полезные данные из регистров, если они были предварительно очищены. Эта техника дополняет другие меры по усилению защиты, такие как рандомизация адресного пространства и защита стека, создавая более устойчивую к атакам среду. Хотя очистка регистров может вносить некоторые накладные расходы на производительность из-за дополнительных инструкций, ее применение оправдано в системах с высокими требованиями к безопасности.

**Проверка границ при вызове стандартных функций**

Механизм FORTIFY_SOURCE [38] представляет собой меру по усилению безопасности, реализованную в компиляторах и предназначенную для обнаружения и предотвращения некоторых распространенных типов уязвимостей, связанных с переполнением буфера, при работе со стандартными функциями библиотеки языка C.

Основной принцип действия FORTIFY_SOURCE заключается в замене небезопасных версий стандартных библиотечных функций (например, `memcpy()`, `strcpy()`, `sprintf()`) на их более защищенные аналоги. Эти аналоги включают дополнительные проверки во время компиляции и во время выполнения, чтобы убедиться, что операции с памятью не выходят за пределы выделенного буфера назначения. Компилятор использует информацию, полученную в ходе оптимизационных проходов, для определения размеров буферов.

Когда FORTIFY_SOURCE обнаруживает потенциальное переполнение буфера во время компиляции, он выдает ошибку. Если проверка происходит во время выполнения и обнаруживается попытка записи за пределы буфера, программа обычно аварийно завершается, что предотвращает эксплуатацию уязвимости, которая могла бы привести к выполнению произвольного кода или повреждению данных.

Преимущества использования FORTIFY_SOURCE заключаются в повышении устойчивости программного обеспечения к атакам, эксплуатирующим переполнения буфера в стандартных функциях. Это достигается без необходимости модификации исходного кода приложения, так как защита встраивается компилятором.

Для функций ядра, осуществляющих копирование из регионов памяти пользовательского пространства и в них могут быть реализована аналогичная защита. Например, для ядра Linux была реализована защита PAX_USERCOPY и включена в основную ветвь разработки под названием HARDENED_USERCOPY [39].

**Пометка флагом “только для чтения” записей в таблице перемещений**

Технология RELRO (Relocation Read-Only) [40] направлена на усиление безопасности исполняемых файлов и разделяемых библиотек путем установления атрибута “только для чтения” для определенных участков памяти после их первоначальной инициализации динамическим компоновщиком. Ключевой задачей является защита внутренних структур

данных, используемых для разрешения адресов символов во время выполнения, в частности Global Offset Table (GOT), от несанкционированной модификации злоумышленником.

Применение RELRO обеспечивает существенное повышение уровня безопасности, поскольку полностью исключает возможность модификации таблицы GOT после завершения процесса загрузки программы. Это эффективно противодействует атакам, основанным на перезаписи указателей в GOT с целью перенаправления вызовов легитимных функций на вредоносный код. Подобные атаки часто применяются для обхода других защитных механизмов, таких как W^X.

В качестве компромисса за повышенную безопасность выступает некоторое увеличение времени запуска программы при использовании RELRO. Это обусловлено необходимостью разрешения всех символов до начала выполнения основного кода, а не по мере их использования. Для большинства приложений данное увеличение времени запуска является незначительным, однако оно может быть более заметным для программ, имеющих большое количество динамических зависимостей.

## 3.2 Механизмы защиты аппаратного стека

Стек в программе – это область памяти, которая используется для хранения локальных переменных функций, адресов возврата и других данных, необходимых для корректной работы программы. Он работает по принципу LIFO (Last In, First Out – “последний пришёл, первый ушёл”), что означает, что последний элемент, добавленный в стек, будет первым извлечён из него. Стек является уязвимым местом, которое может быть использовано злоумышленниками в худшем случае для выполнения произвольного кода и изменения поведения программы. Однако, на нем также могут хранится чувствительные данные (пароли, ключи шифрования), несанкционированное считывание которых приводит к утечке информации.

Одна из наиболее распространённых уязвимостей связана с переполнением буфера в стеке. Если программа не проверяет границы массива или не контролирует количество данных, записываемых в буфер, злоумышленник может записать данные за пределы выделенной области памяти и изменить содержимое стека.

**Стековые канарейки**

Самым известным механизмом защиты стека является StackGuard [41,42]/Stack Smashing Protector [43]. Он основан на концепции “канареек” (stack canaries). Этот метод защиты получил своё название по аналогии с канарейками, которых шахтёры брали с собой в шахты для раннего обнаружения опасных газов.

Механизм работает путем размещения специального контрольного значения (канарейки) между локальными переменными функции и служебной информацией стека, такой как адрес возврата. При входе в функцию это значение записывается в стек, а при выходе проверяется. Если значение изменилось, это означает, что произошло переполнение буфера, которое могло повредить важные данные стека.

Существует три основных типа канареечных значений, каждый со своими особенностями защиты: терминаторные канарейки, использующие символы-терминаторы (NULL, CR, LF, EOF, …) для предотвращения переполнения при использовании строковых функций; случайные канарейки; и случайные XOR-канарейки, дополнительно комбинируемые с нестатическим значением (обычно регистр `%rbp`), что в современных ОС с рандомизацией адресного пространства обеспечивает дополнительную защиту. В x86-64 Linux с glibc, где используется 64-битное случайное значение с обнуленными последними байтами.

Метод встроен в современные компиляторы, такие как GCC, Clang и включается по умолчанию. Это позволяет осуществлять сборку программного обеспечения со встроенной защитой. Существуют разные степени глубины внедрения проверок, от проверки каждой функции и до полного отключения защиты, что позволяет найти баланс между

производительностью и безопасностью. Сбалансированным подходом выступает эвристика оценки степени задействованности стека функцией. Если функция активно использует стек, хранит на нем значений больше определенного порога, вызывает сторонние функции, работающие со значениями на стеке, то срабатывает эвристика и компилятор добавляет проверку контрольного значения со стека в функцию. Помимо этого, компиляторы GCC и Clang для повышения эффективности защиты размещают переменные на стеке специальным образом: ближе всего к канарейке располагаются большие массивы и структуры с ними, затем малые массивы и их структуры, за ними следуют переменные, на которые ссылаются, и в последнюю очередь – все остальные переменные. Это максимизирует вероятность срабатывания защиты при переполнении.

Защита имеет свои недостатки. Она лучше всего срабатывает, когда эксплуатируется ошибка переполнения стека с линейной последовательной перезаписью памяти. Если атакующий контролирует смещение, по которому производится перезапись памяти на стеке, то он может обойти эту защиту. Также, эксплуатируя другую ошибку, приводящую к утечке информации со стека, возможно считать или угадать значение канарейки.

Анализ современных настольных дистрибутивов показывает, что большинство пакетов уже собирается со встроенной защитой канарейки. Такие дистрибутивы как Fedora, Ubuntu, Arch Linux, OpenBSD, Alt Linux и другие включили защиту при сборке всех пакетов более 10 лет назад. Так, в работе [44] было показано что 85% пакетной базы Debian собрано с данной защитой. Однако, анализ прошивок встроенных устройств в той же работе показывает, что уровень внедрения защиты в них на 2022 сильно отстает и составляет в среднем 29.7% (и опускается до 3.28% в случае исключения прошивок двух производителей из общего анализа).

Не смотря на результаты приведенного анализа можно утверждать, что данная технология защиты является устоявшейся. В исследованиях [45,46] поднимается вопрос эффективности данной защиты с точки зрения предотвращения эксплуатации ошибки переполнения стека и накладных расходов на производительность, по сравнению с более новыми методами защиты SafeStack и “теневым стеком” (shadow stack).

**Страничная защита стека (Stack Guard Gap+Stack Clash Protection)**

В современных ОС для предотвращения эксплуатации уязвимостей, связанных с переполнением стека и конфликтом областей памяти (stack clash) [47-49], применяются механизмы Stack Guard Gap на уровне ядра ОС и Stack Clash Protection, встроенная в компиляторы. Эти технологии обеспечивают изоляцию стека пользовательского пространства от других регионов памяти за счёт размещения специальных неотображаемых (guard) страниц и “зондирование” (probing) памяти на стеке при каждом её выделении.

Механизм атаки Stack Clash позволяет злоумышленнику добиться перекрытия областей стека и, например, кучи. Используя различные техники (выделение больших регионов памяти на стеке, переменные окружения или рекурсивные вызовы функций), злоумышленник сближает области стека и кучи, используя знания об относительном размещении областей. После этого стек и куча “сталкиваются”, что позволяет атакующему манипулировать данными в куче через операции со стеком и наоборот.

Основная идея защиты заключается в создании “защитного промежутка” (guard gap) между стеком и другими областями памяти. Этот промежуток реализуется как неотображаемый регион памяти. При попытке выхода за границы стека (например, из-за переполнения) процесс обращается к одной из защитных страниц, что приводит к генерации исключения и аварийному завершению программы, предотвращая потенциальную эксплуатацию уязвимости. Исследования Qualys [49] и более ранние [47] показали, что использования одной страницы памяти (guard page) для изоляции стека недостаточно [50] и должен использоваться больший промежуток [51] (guard gap).

Увеличение размера защитной области между стеком и кучей является только частичным решением проблемы, поскольку крупные аллокации памяти на стеке все еще могут “перепрыгнуть” даже через увеличенную защитную область. Для полноценной защиты необходимо обеспечить “зондирование” (probing) памяти на стеке при каждом выделении. Для этого в компиляторах GCC и Clang была реализована опция -fstack-clash-protection [52,53].

Для защиты стеков самого ядра ОС применяется схожий механизм изоляции. Рассмотрим защиту на примере стеков самого ядра Linux. В основной ветви разработки было внедрено виртуальное отображение стека ядра VMAP_STACK [54]. Вместо использования непрерывного адресного пространства каждый стек ядра размещается в отдельном виртуальном регионе, окружённом защитными страницами. При выходе за границы стека происходит обращение к защитной странице, что вызывает ошибку памяти. Дополнительно, в ядре Linux с версии v4.20 на уровне исходного кода было запрещено использование массивов переменной длины (Variable-Length Arrays, VLA) [55]. С точки зрения безопасности, использование VLA значительно затрудняет статическую оценку использования стека программой, что может привести к выходу за его пределы, особенно в контексте ядра ОС.

Страничная защита стека является важным механизмом безопасности, который предотвращает атаки типа Stack Clash, направленные на обход защиты Stack Guard Gap. Комбинация защиты на уровне ядра и на уровне компилятора обеспечивает хороший уровень защиты от данного типа атак.

**Проверка целостности конца стека**

Также, существуют механизмы дополнительные проверки целостности конца стека. В конец стека вставляется канарейка, и при каждом переключении контекста происходит проверка целостности стека. Это актуально, когда программа сама управляет своим стеком. Например, в ядре Linux используется механизм SCHED_STACK_END_CHECK [56].

В отличии от механизма стековых канареек Stack Protector, который размещает канарейку перед локальными переменными в каждом фрейме стека, механизм SCHED_STACK_END_CHECK делает это только в конце всего стека потока. У каждого потока ядра есть свой стек, и при каждом вызове планировщика ядра происходит проверка целостности конца стека путем сверки значения канарейки. Stack Protector же осуществляет проверку при выходе из функции. Stack Protector защищает от переполнения буфера в рамках функции. SCHED_STACK_END_CHECK - обнаруживает общее переполнение всего стека потока. В большей степени данная опция является отладочной, однако в виду легковесности проверки, её часто также рассматривают как механизм усиления.

**Предотвращение утечек информации через стек ядра (StackLeak)**

StackLeak [57,58] является механизмом защиты в ядре Linux, который нацелен на предотвращение утечек конфиденциальной информации через неинициализированные участки стека ядра. Проблема, которую он решает, заключается в том, что функции ядра при завершении своей работы могут оставлять на стеке чувствительные данные. Эти данные, если их не очистить, могут быть прочитаны другими функциями, которые впоследствии будут использовать ту же область стека, создавая тем самым уязвимость утечки информации [59]. Это особенно важно в случаях, когда злоумышленник может выполнять последовательные системные вызовы и анализировать содержимое стека.

StackLeak работает на двух уровнях: во время компиляции и во время выполнения. На этапе компиляции специальный плагин компилятора GCC добавляет в функции ядра инструментацию для отслеживания максимальной глубины использования стека. Во время выполнения эта информация используется для определения области стека, которую необходимо очистить. При выходе из системного вызова StackLeak определяет, насколько глубоко был использован стек, и очищает все области стека выше этой отметки. Это

предотвращает утечку информации через неинициализированные области стека при последующих системных вызовах.

Таким образом, StackLeak включает в себя несколько ключевых компонентов. Первый компонент – это плагин компилятора, который добавляет необходимую инструментацию в код ядра. Он анализирует каждую функцию и вставляет код для отслеживания использования стека. Второй компонент – это механизм очистки стека, который активируется при возврате из системных вызовов. Третий компонент – это система мониторинга использования стека, которая помогает обнаруживать потенциальные проблемы с глубиной стека на ранних стадиях.

За счет использования оптимизированных инструкции процессора для очистки памяти, и ограничения инструментации кода до функций, которые действительно могут привести к утечке информации, разработчикам удалось минимизировать накладные расходы.

**Принудительная инициализация переменных на стеке**

Такие руководства по безопасному коду как MISRA C [60] не спроста содержат требования к инициализации переменных. Использование неинициализированных переменных может привести уязвимостям утечки информации. Однако на практике, проконтролировать инициализацию локальных переменных в больших проектах с длительной историей разработки, которые изначально не придерживались стандартов безопасного написания кода, имеют много контрибьютеров с разными взглядами на оформление кода, представляется сложным. Статические анализаторы могут выявлять такого рода ошибки, но не дают гарантий их полного отсутствия в проекте, а также могут сообщать ложноположительные срабатывания. Динамические анализаторы, выявляют подобные ошибки, но также не дают гарантий их полного отсутствия в проекте, накладывают существенные накладные расходы при работе и имеют ложноотрицательные срабатывания. К тому же, в больших открытых проектах, не все изменения кода проходят статический и/или динамический анализ перед тем, как попасть в релиз. Это означает, что ошибка может быть в проекте достаточно длительное время, до того как исправление будет принято, попадет в новый релиз и релизы с исправлениями.

В качестве ограниченного решения проблемы неинициализированных переменных, которые могут приводить к утечкам информации в пространство пользователя при обработке системных вызовов в ядре Linux, команда PaX предложила плагин для GCC STRUCTLEAK [61]. Плагин добавляет принудительную инициализацию полей структур при сборке ядра, которые могут копироваться в пространство пользователя при обработке системных вызовов в ядре Linux. Позднее этот плагин был добавлен в основную ветку разработки ядра Linux [62]. Однако, плагин завязан на специфику ядра Linux и защищает только от утечек информации из ядра в пространство пользователя, накладывая минимальные накладные расходы.

Разработчики компиляторов GCC и Clang реализовали более общий вариант защиты и добавили механизм принудительной инициализации неинициализированных переменных на стеке trivial-auto-var-init [63, 64]. Он работает для всех переменных, не завязан на специфику какого-либо из проектов, и превентивно защищает от утечек информации через неинициализированные переменные на стеке. Позднее, разработчики ядра Linux, также включили этот механизм в ядро в виде INIT_STACK_ALL_ZERO [65].

**Рандомизация смещения стека ядра при каждом системном вызове**

Рандомизация смещения стека ядра Linux (PaX RANDKSTACK [66]/RANDOMIZE_KSTACK_OFFSET [67]) – это механизм защиты, повышающий безопасность системы за счет внесения случайности (энтропии) в расположение стека ядра при обработке системных вызовов. Это значительно затрудняет злоумышленникам предсказание адресов в стеке и, следовательно, эксплуатацию уязвимостей, связанных со стеком.

Изначально защита была реализована Pax Team в виде механизма RANDKSTACK, но в основную ветвь разработки ядра Linux она попала позднее в виде переработанного механизма RANDOMIZE_KSTACK_OFFSET. Рассмотрим последний детальнее.

Основная идея заключается в том, что, поскольку смещение стека рандомизируется при каждом системном вызове, злоумышленнику становится сложнее надежно попасть в определенное место в стеке, даже при утечках адресов, так как база стека будет меняться при следующем системном вызове.

Традиционно ядро выделяет новый стек для процесса или потока, выравнивая его по фиксированному смещению относительно базового адреса области стека. RANDOMIZE_KSTACK_OFFSET изменяет этот подход, добавляя к указателю стека случайное смещение при каждом системном вызове. Это смещение выбирается из определенного диапазона, что позволяет соблюсти архитектурные требования к выравниванию стека и одновременно делает его точное положение непредсказуемым.

При каждом системном вызове, после сохранения регистров процессора, но до начала активного использования стека, ядро генерирует это случайное смещение. Оно применяется к указателю стека, сдвигая его фактическое начало.

С точки зрения безопасности, RANDOMIZE_KSTACK_OFFSET затрудняет атаки методом возвратно-ориентированного программирования (ROP) [68], так как адреса гаджетов в стеке становятся менее предсказуемыми. Помогает защититься от атак на переполнение стека, поскольку неопределенность в расположении стека мешает злоумышленникам точно разместить свою полезную нагрузку. Препятствует использованию утечек адресов стека: адрес, полученный в одном системном вызове, становится неактуальным для следующего, так как база стека изменится.

Влияние RANDOMIZE_KSTACK_OFFSET на производительность, как правило, минимально, но измеримо и составляет порядка 1%. RANDOMIZE_KSTACK_OFFSET эффективно дополняет другие механизмы безопасности ядра. Он усиливает рандомизацию адресного пространства ядра (KASLR), внося дополнительный уровень непредсказуемости именно для адресов в стеке. Механизм также хорошо сочетается со стековыми канарейками (stack canaries) и другими техниками защиты стека (например, STACKLEAK), формируя многоуровневую систему защиты от атак на стек.

## 3.3 Механизмы защиты динамической памяти

Для усиления безопасности динамического выделения памяти применяется комплекс механизмов. Основная цель данных механизмов – противодействие широкому спектру уязвимостей, связанных с повреждением памяти, таких как использование после освобождения (use-after-free), переполнения буфера кучи (heap overflows), двойное освобождение (double-free) и утечки конфиденциальных данных. Эти защиты могут охватывать различные компоненты подсистемы управления памятью, включая аллокаторы кучи (например, SLAB/SLUB в ядре Linux) и страничные аллокаторы. Для достижения этой цели применяются разнообразные техники, включая рандомизацию расположения объектов и списков свободной памяти, принудительную инициализацию и очищение выделяемой/освобождаемой памяти, кодирование указателей и изоляцию критически важных структур. Совокупность этих мер значительно повышает сложность и снижает надежность эксплуатации уязвимостей, основанных на манипуляциях с динамической памятью.

Стоит отметить, что описанные подходы к защите динамической памяти не являются уникальными для какой-либо одной системы или только для пространства ядра. Аналогичные принципы и механизмы активно применяются и в пользовательском пространстве, в частности, в реализациях стандартных библиотечных аллокаторов памяти, таких как glibc `malloc`, `jemalloc` или `tcmalloc`. Эти аллокаторы также могут включать

в себя рандомизацию, защиту метаданных, “красные зоны” (red zones) и другие техники для противодействия атакам на кучу.

Мы рассмотрим реализацию этих механизмов защиты динамической памяти на примере ядра Linux.

**Рандомизация списков свободной памяти аллокатора**

Механизм SLAB_FREELIST_RANDOM [69] представляет собой средство усиления защиты, направленное на рандомизацию порядка следования освобожденных объектов в списках свободных блоков распределителя памяти SLAB. Принцип его действия заключается в том, что в момент освобождения блока памяти он помещается не в предопределенную позицию списка свободных элементов (например, в его начало или конец), а на случайным образом выбранное место. Данная рандомизация осуществляется на этапе инициализации каждого отдельного SLAB-кэша, который представляет собой пул для хранения объектов определенного размера. В результате формируется уникальный, непредсказуемый паттерн организации списка свободных блоков, остающийся неизменным на протяжении всего жизненного цикла соответствующего кэша.

Такой подход к управлению свободными блоками памяти значительно затрудняет эксплуатацию определенных классов уязвимостей. В частности, усложняется реализация атак типа использование после освобождения, поскольку предсказать, какой именно ранее освобожденный объект будет выделен повторно, становится значительно труднее. Также снижается эффективность атак методом заполнения кучи, целью которых является размещение контролируемых злоумышленником данных по предсказуемым адресам. В более общем смысле, любые атаки, основанные на повреждении памяти и использующие предсказуемость расположения объектов, затрудняются подобной защитой.

**Кодирование указателей списка свободной памяти аллокатора**

Механизм SLAB_FREELIST_HARDENED [70] представляет собой средство усиления защиты в системе управления памятью ядра Linux. Его функционирование основано на преобразовании указателей, содержащихся в списках свободных объектов памяти. Суть преобразования заключается в том, что каждый указатель в таком списке подвергается побитовой операции XOR с двумя величинами: со случайным значением, которое является уникальным для каждого отдельного SLAB-кэша, и с адресом той ячейки памяти, где непосредственно хранится сам преобразуемый указатель.

Такой подход к кодированию указателей обеспечивает несколько важных аспектов защиты. Прежде всего, он эффективно противодействует прямой манипуляции указателями, поскольку злоумышленник не может просто заменить существующий указатель своим собственным значением: оно не будет корректно закодировано. Кроме того, система защищена от предсказания. Даже если атакующему удастся узнать один закодированный указатель, это не позволит ему узнать значения других указателей из-за случайного значения и зависимости от адреса хранения. Наконец, реализуется принцип локальности: закодированный указатель сохраняет свою валидность исключительно в том конкретном месте памяти, для которого он был сформирован, что ограничивает его возможное неправомерное использование в других контекстах.

**Разделение корзин аллокации для объектов ядра и пользовательского пространства**

Механизм SLAB_BUCKETS [71] представляет собой средство усиления безопасности управления памятью, позволяющее изолировать потенциально опасные аллокации памяти в отдельные, специализированные наборы корзин. Ключевым отличием от традиционного подхода, при котором все объекты одного размера могли размещаться в общих корзинах аллокатора, является целенаправленное разделение объектов, например по типу объектов, между пользовательским пространством и пространством ядра.

Такое разделение корзин аллокации имеет ряд преимуществ с точки зрения безопасности. Оно обеспечивает физическое разграничение между объектами, которые может

контролировать пользователь (например, через системные вызовы) и объектами ядра. Это затрудняет атаки, стремящиеся повлиять на структуры ядра через манипуляции с пользовательскими данными. Кроме того, усложняется проведение атак типа заполнения кучи и снижается общая предсказуемость размещения объектов в памяти, что является важным фактором противодействия попыткам эксплуатации уязвимостей. Данный механизм также формирует дополнительный уровень изоляции для критических структур ядра, оберегая их от потенциального повреждения.

**Принудительная инициализация выделяемой и очищение освобожденной памяти**

Механизм INIT_ON_ALLOC [72] обеспечивает принудительное затирание нулевыми значениями всей выделяемой памяти. Это позволяет полностью устранить уязвимости, связанные с использованием неинициализированной памяти кучи, предотвратить утечки конфиденциальных данных и обеспечить детерминированное начальное состояние выделенной памяти.

Механизм INIT_ON_FREE [72], в свою очередь, принудительно затирает нулевыми значениями всю освобождаемую память. Данный подход способствует сокращению времени жизни данных в памяти, противодействует криминалистическому анализу памяти, обеспечивает защиту от атак с холодной загрузкой (cold boot attacks) и предоставляет дополнительный уровень защиты от утечек данных. Сокращение времени жизни данных затрудняет такие атаки типа заполнения кучи.

Влияние этих механизмов на производительность различно. Использование INIT_ON_ALLOC приводит к снижению производительности примерно на 1% в реальных системах, тогда как INIT_ON_FREE может вызвать более заметное снижение, составляющее приблизительно 3-5% в реальных условиях эксплуатации.

**Рандомизация кэшей аллокации**

Механизм RANDOM_KMALLOC_CACHES [73] является средством усиления защиты ядра Linux, вносящим элемент случайности в систему управления динамической памятью. В отличие от традиционного подхода, где выделения памяти определенного размера и типа обслуживаются одним и тем же кэшем, RANDOM_KMALLOC_CACHES предусматривает создание нескольких идентичных наборов кэшей. При каждом запросе на выделение памяти система производит случайный выбор одного из этих наборов для его обслуживания. Такое решение существенно снижает предсказуемость размещения объектов в памяти для потенциального злоумышленника.

Функционирование данного механизма основано на двух ключевых аспектах. Во-первых, на этапе инициализации системы создается несколько идентичных наборов кэшей. Во-вторых, при каждом выделении памяти осуществляется случайный выбор одного из этих наборов.

Внедрение RANDOM_KMALLOC_CACHES значительно усиливает защиту ядра от атак, основанных на предсказании адресов памяти. Невозможность для атакующего точно определить, в каком из кэшей будет размещен целевой объект, существенно затрудняет проведение атак типа заполнения кучи, эксплуатацию уязвимостей использования после освобождения, а также иные манипуляции, связанные с предсказуемым расположением объектов в памяти. Таким образом, механизм усложняет эксплуатацию уязвимостей, вынуждая злоумышленника учитывать множество вероятных сценариев размещения объектов, что требует разработки более сложных и менее надежных эксплойтов.

Данный механизм эффективно взаимодействует с другими средствами защиты. Например, если SLAB_FREELIST_RANDOM обеспечивает рандомизацию порядка объектов внутри отдельного кэша, то RANDOM_KMALLOC_CACHES вносит дополнительный уровень случайности путем выбора самого кэша, формируя таким образом многоуровневую систему защиты. Аналогичным образом, при совместном использовании с SLAB_BUCKETS, рандомизация выбора кэшей еще более затрудняет атаки, основанные на предсказуемом размещении объектов в памяти.

В заключение, RANDOM_KMALLOC_CACHES представляет собой действенный инструмент повышения безопасности ядра ОС за счет привнесения непредсказуемости в процесс управления памятью. Сочетание незначительных накладных расходов с ощутимым усилением защиты делает его ценным элементом комплексной стратегии обеспечения безопасности ядра Linux.

## *4. Защита потока управления*

### 4.1 Контроль целостности потока управления (CFI)

Контроль целостности потока управления (Control-Flow Integrity, CFI) [74,75] – это технология защиты, разработанная для предотвращения атак, направленных на несанкционированное изменение потока выполнения программы. Основной принцип CFI заключается в обеспечении того, чтобы выполнение программы строго соответствовало заранее определенному графу потока управления (Control-Flow Graph, CFG). Этот граф представляет собой набор всех допустимых путей выполнения, который обычно определяется на этапе компиляции путем статического анализа исходного кода или бинарного файла, либо во время выполнения посредством динамического анализа (профилирования).

CFI предназначен для противодействия эксплуатации уязвимостей, связанных с повреждением памяти (например, переполнение буфера, использование после освобождения). Подобные уязвимости могут позволить атакующему получить контроль над указателем инструкций (instruction pointer, IP) и перенаправить выполнение программы на произвольный или контролируемый им код. Часто для этого используются такие техники, как возвратно-ориентированное программирование (Return-Oriented Programming, ROP), переходо-ориентированное программирование (Jump-Oriented Programming, JOP), а также их вариации (Call-Oriented Programming, Loop-Oriented Programming). Продвинутые техники эксплуатации зачастую способны обойти механизмы защиты предотвращения исполнения данных, рандомизация размещения адресного пространства и защиты от переполнения стека. CFI существенно усложняет или делает практически невозможным использование этих уязвимостей, принуждая поток управления оставаться в пределах доверенного CFG.

Механизм CFI основан на динамической проверке инструкций косвенной передачи управления во время выполнения программы. К таким инструкциям относятся косвенные вызовы функций, косвенные переходы и возвраты из функций, чьи адреса назначения определяются в процессе работы программы. Механизмы CFI призваны защищать как “прямые ребра” графа потока управления (forward edges), соответствующие косвенным вызовам и переходам, так и “обратные ребра” (backward edges), соответствующие инструкциям возврата из функций. Защита обратных ребер часто реализуется с помощью теневого стека вызовов (Shadow Call Stack, SCS), который будет рассмотрен далее.

Процесс реализации CFI обычно включает следующие ключевые этапы:

1. Построение графа потока управления (CFG): На этом этапе генерируется CFG, который определяет все легитимные точки назначения для каждой инструкции косвенной передачи управления в программе.
2. Инструментация кода: Каждой допустимой точке назначения косвенного перехода (например, началу функции или определенному блоку кода) присваивается уникальный идентификатор (метка, ID или тип). Перед каждой инструкцией косвенной передачи управления вставляется специальный код (проверка).
3. Проверка во время выполнения: Непосредственно перед осуществлением косвенной передачи управления выполняется ранее вставленный проверочный код. Эта проверка заключается в том, чтобы убедиться, что целевой адрес перехода

соответствует ожидаемому идентификатору (или типу) для данной точки перехода согласно CFG.

4. Обработка нарушений: Если проверка не проходит (т.е. обнаружено отклонение от развешенного CFG), это сигнализирует о потенциальной атаке. В этом случае выполнение программы обычно прерывается.

Реализации CFI могут значительно различаться по нескольким ключевым характеристикам, которые влияют на их строгость, производительность и практическую применимость [75]. Различают два основных подхода к точности построения CFG: fine-grained CFI и coarse-grained CFI. Fine-grained CFI стремится к построению максимально точного и строгого CFG, разрешая переходы потока управления только по точно определенным допустимым ребрам, при этом каждому ребру CFG могут назначаться уникальные метки, которые проверяются при передаче управления. В отличие от этого, coarse-grained CFI ослабляет строгость проверок, используя более общие и простые правила, которые могут не требовать полного и точного CFG. Например, в этом случае возвраты могут быть разрешены к любой инструкции, непосредственно следующей за инструкцией вызова, а косвенные переходы – к любой точке входа в функцию. Такой подход выступает компромиссным между уровнем обеспечиваемой безопасности и накладными расходами.

CFG может быть построен с использованием статического анализа, динамического анализа или анализа на основе типов. Статический анализ вычисляет CFG на основе исходного кода или бинарных файлов; его ограничения (например, невозможность точно разрешить все косвенные переходы) могут привести к аппроксимации реальных путей выполнения, включая несуществующие (но теоретически возможные с точки зрения анализатора) ребра в CFG, что потенциально снижает безопасность. Динамический анализ строит или уточняет CFG во время исполнения программы на основе собранных трасс; при этом существует риск ложных срабатываний CFI, если не все легитимные участки кода были пройдены при анализе. Анализ на основе типов (Type-based CFI) генерирует CFG или его часть на основе информации о типах, например, сигнатур функций, для проверки корректности вызовов.

Современные процессоры [76] начинают включать аппаратные функции для CFI, такие как Intel Control-flow Enforcement Technology (CET) [77] и ARM Pointer Authentication Code (PAC) [78]. Они направлены на обеспечение практичного и эффективного применения CFI с низкими накладными расходами.

Хотя CFI значительно усложняет атаки, нацеленные на перехват потока управления, различные исследования продемонстрировали ограничения и уязвимости в существующих реализациях.

Программные проверки CFI могут вызывать значительные накладные расходы на производительность, что выступает значительным барьером для их широкого внедрения. Аппаратная поддержка CFI призвана снизить эти издержки. Несмотря на эти проблемы, исследования продолжают улучшать техники CFI [79,80], фокусируясь на повышении точности анализа CFG, снижении накладных расходов, улучшении формальных гарантий безопасности и более эффективном использовании аппаратной поддержки.

## 4.2 Теневой стек (Shadow Call Stack)

Механизм теневого стека (Shadow Call Stack, SCS) представляет собой средство защиты, предназначенное для хранения копий адресов возврата функций в обособленной, защищенной области памяти. Дублирование позволяет эффективно обнаруживать и предотвращать атаки, нацеленные на несанкционированное изменение адресов возврата, сохраненных в основном стеке вызовов. Данный механизм является одной из реализаций концепции контроля целостности потока управления, он используется для защиты обратных ребер (Backward-Edge Protection) в CFI.

Принцип функционирования теневого стека основан на том, что в момент вызова функции ее адрес возврата записывается одновременно как в основной, так и в теневой стек. Впоследствии, при выполнении инструкций возврата из функции, адрес, извлекаемый из основного стека, подвергается сравнению с соответствующим адресом из теневого стека. В случае выявления каких-либо расхождений между этими адресами, что указывает на потенциальную модификацию адреса возврата в основном стеке, выполнение программы немедленно прерывается. Это действие направлено на предотвращение эксплуатации уязвимости и захвата контроля над потоком выполнения программы.

Существуют аппаратные реализации теневого стека. Например, буфер стека возвратов (Return Stack Buffer/Intel RSB) или теневой стек (AMD Shadow Stack), предназначенные для нейтрализации ROP-атак, которые осуществляются путем модификации адресов возврата в основном стеке вызовов. Они создают изолированную и защищенную на аппаратном уровне область памяти, предназначенную исключительно для хранения копий адресов возврата.

Теневой стек обеспечивает более высокий уровень защиты по сравнению с традиционными методами, такими как стековые канарейки. Это достигается за счет того, что адреса возврата не просто обрамляются контрольными значениями, а полностью дублируются в специально выделенной и защищенной области памяти. При корректной реализации данный подход делает практически невозможным успешное осуществление атак класса Return-Oriented Programming (ROP), которые полагаются на перезапись адресов возврата.

## *5. Защита целостности системы*

Защита целостности системы направлена на обеспечение того, чтобы программное обеспечение и данные не были несанкционированно изменены. Это важно для поддержания доверия к системе и предотвращения выполнения вредоносного кода или использования искаженных данных. Механизмы контроля целостности проверяют, что компоненты системы, такие как загрузчик, ядро операционной системы, системные файлы и конфигурации, соответствуют ожидаемому (доверенному) состоянию.

### 5.1 Безопасная загрузка (Secure Boot)

Безопасная загрузка (Secure Boot) представляет собой механизм обеспечения целостности программного обеспечения на ранних этапах инициализации вычислительной системы. Данная технология, являющаяся неотъемлемой частью стандарта UEFI (Unified Extensible Firmware Interface) [81] предназначена для предотвращения выполнения неавторизованного или модифицированного кода, такого как загрузочные вирусы (буткиты) и руткиты, до полной загрузки ОС.

Принцип действия Secure Boot основан на концепции криптографически заверенной “цепочки доверия” (chain of trust). Начиная с первоначального загрузочного кода прошивки, который предполагается доверенным, каждый последующий загружаемый компонент – от прошивки до загрузчика ОС, ядра ОС и его модулей – подвергается проверке цифровой подписи. Эта верификация осуществляется с использованием набора доверенных сертификатов (ключей), хранящихся в защищенной энергонезависимой памяти платформы, обычно в переменных UEFI, таких как Platform Key (PK), Key Exchange Keys (KEKs). В случае, если цифровая подпись компонента недействительна процесс загрузки прерывается или система переходит в режим ограниченной функциональности, предотвращая выполнение потенциально скомпрометированного кода.

Основным преимуществом Secure Boot является значительное усиление защиты от вредоносного программного обеспечения, активного на этапе загрузки. Путем обеспечения того, что только подписанный и доверенный код может быть выполнен до загрузки операционной системы, Secure Boot эффективно противодействует многим классам атак, направленных на раннюю компрометацию системы и обхода защитных механизмов ОС [82].

Secure Boot формирует основу для построения доверенной вычислительной среды и повышает общую целостность системы.

## 5.2 Контроль целостности

Архитектура измерения целостности (Integrity Measurement Architecture, IMA) [83] представляет собой компонент подсистемы безопасности ядра Linux, предназначенный для динамической проверки целостности файлов и конфигураций. Основная задача IMA заключается в обнаружении несанкционированных изменений в файловой системе путем вычисления контрольных сумм файлов перед их использованием и сравнения их с эталонными значениями.

Принцип работы IMA включает несколько этапов. Во-первых, это измерение: при каждом доступе к файлу (например, при открытии на чтение или выполнение) вычисляется его хэш. Во-вторых, это оценка: вычисленный хэш может сравниваться с ранее сохраненным эталонным значением, которое считается доверенным. В случае несоответствия система может предпринять определенные действия, такие как генерация оповещения или блокировка доступа к файлу. В-третьих, это аудит: все события измерения и результаты проверок могут протоколироваться для последующего анализа. Важной особенностью IMA является ее интеграция с доверенным платформенным модулем (TPM). Хэши измеряемых файлов могут сохраняться в регистрах конфигурации платформы (PCR) TPM, что обеспечивает защищенное хранение истории измерений и позволяет проводить удаленную аттестацию целостности системы.

Для усиления защиты, особенно в отношении метаданных файлов, IMA часто используется совместно с модулем расширенной проверки (Extended Verification Module, EVM) [84]. EVM нацелен на защиту расширенных атрибутов файлов (xattrs), включая те, в которых IMA может хранить эталонные хэши, а также других важных атрибуты безопасности, таких как права доступа. EVM вычисляет криптографическую подпись для набора защищаемых метаданных файла, используя ключ, который может быть защищен TPM. При каждом доступе к файлу EVM проверяет целостность этих метаданных, предотвращая атаки, направленные на их модификацию с целью обхода механизмов IMA.

Преимущества совместного использования IMA и EVM заключаются в обеспечении комплексной защиты как содержимого файлов, так и их важных метаданных от несанкционированных модификаций. Интеграция с TPM значительно повышает надежность системы, так как ключи и эталонные измерения защищены аппаратными средствами. Это позволяет строить доверенные вычислительные среды и проводить достоверную аттестацию состояния системы.

Однако применение данных механизмов сопряжено с определенными ограничениями. К ним относятся потенциально высокие накладные расходы на производительность, особенно при интенсивной работе с файловой системой. Кроме того, требуется тщательная настройка политик измерения и оценки для предотвращения ложных срабатываний и обеспечения корректной работы системы. Управление эталонными значениями хэшей и ключами EVM также представляет собой нетривиальную задачу.

## 5.3 Блокировка ядра (Kernel Lockdown)

Механизм блокировки ядра (Kernel Lockdown) [85] представляет собой модуль безопасности, реализованный в ядре Linux и предназначенный для усиления границы между пользовательским пространством и ядром. Основная цель данного механизма заключается в предотвращении как случайного, так и намеренного изменения кода ядра или его критических данных со стороны процессов, обладающих правами суперпользователя, что могло бы привести к компрометации ядра ОС. То есть, Lockdown выступает логическим продолжением механизмов Secure Boot, контроля целостности и в некоторой степени обеспечивает контроль целостности самого ядра.

Принципы работы механизма блокировки ядра основаны на ограничении доступа к определенным функциям и интерфейсам ядра, которые могут быть использованы для обхода существующих механизмов безопасности или для выполнения произвольного кода в контексте ядра, даже если процесс уже обладает привилегиями. Многие атаки нацелены на первоначальное получение прав суперпользователя с последующим использованием этих прав для загрузки вредоносных модулей ядра, модификации его памяти или иного вмешательства в его работу. Функциональность Lockdown включает запрет на загрузку неподписанных модулей ядра, особенно если система загружена в режиме UEFI Secure Boot. Также накладываются ограничения на доступ к таким интерфейсам, как `/dev/mem`, `/dev/kmem` и `/dev/port`, с целью предотвращения прямого чтения или записи в память ядра и управления портами ввода-вывода. Кроме того, ограничивается использование некоторых отладочных интерфейсов, которые могут быть неправомерно использованы для модификации поведения ядра, и запрещается изменение некоторых критически важных параметров ядра через файловые системы `/proc` или `/sys` после активации режима блокировки.

В ядре Linux механизм Lockdown может функционировать в нескольких режимах, в частности режимы целостности и конфиденциальности. Режим целостности направлен на предотвращение любых модификаций ядра, способных ее нарушить. Например, в этом режиме запрещается загрузка модулей ядра, если они не подписаны доверенным цифровым ключом, или использование интерфейсов, позволяющих изменять код работающего ядра. Режим конфиденциальности, в свою очередь, включает все ограничения режима целостности и дополнительно вводит меры, направленные на предотвращение извлечения конфиденциальной информации из памяти ядра. Это включает ограничение доступа к информации, которая могла бы раскрыть адреса в памяти ядра, структуру данных ядра или другие чувствительные сведения.

Применение механизма блокировки ядра предоставляет ряд существенных преимуществ для безопасности системы. Во-первых, он значительно сокращает поверхность атаки на ядро со стороны процессов, уже обладающих правами суперпользователя, ограничивая возможности для злонамеренных действий даже после получения привилегий. Во-вторых, усиливается защита от руткитов и других типов вредоносных программ, которые пытаются модифицировать ядро для сокрытия своего присутствия или для получения полного контроля над системой. В-третьих, повышается общий уровень безопасности системы, что особенно актуально для сред, где предъявляются высокие требования к доверию, например, при использовании технологии Secure Boot. Связка UEFI Secure Boot, IMA/EVM и Lockdown обеспечивает формирование непрерывной цепочки доверия от загрузчика до полностью функционирующего ядра.

Несмотря на очевидные преимущества, использование механизма блокировки ядра сопряжено с некоторыми ограничениями и требует учета определенных соображений. Существует вероятность нарушения нормальной работы приложений или специализированных системных инструментов, которым для выполнения своих функций необходим глубокий доступ к внутренним компонентам ядра. К таким инструментам могут относиться, например, отладчики, специфические драйверы устройств или программное обеспечение для виртуализации. Блокировка ядра не является универсальным решением всех проблем безопасности и должна применяться как один из элементов комплексной, многоуровневой стратегии защиты.

## *6. Самоограничение привилегий*

Принцип минимальных привилегий (Principle of Least Privilege, PoLP) – это концепция в области информационной безопасности, согласно которой каждому модулю системы (например, процессу, пользователю или программе) должны быть предоставлены только те

привилегии, которые необходимы для выполнения его непосредственных задач, и не более того. Этот подход направлен на ограничение потенциального ущерба, который может быть нанесен в случае компрометации или некорректной работы компонента системы.

Одним из ключевых методов реализации принципа минимальных привилегий является практика сброса привилегий (privilege dropping). Суть этого метода заключается в том, что процесс, изначально запущенный с повышенными привилегиями (например, правами суперпользователя, необходимыми для инициализации, такой как открытие привилегированных портов или доступ к защищенным файлам), после выполнения этих начальных операций добровольно и необратимо отказывается от избыточных прав. Таким образом, основную часть своей работы процесс выполняет с минимально необходимым набором полномочий. Это значительно снижает риски: если такой процесс будет скомпрометирован злоумышленником, то атакующий получит доступ только к ограниченному набору привилегий, что существенно сужает его возможности для дальнейшего развития атаки и нанесения вреда системе.

Рассмотрим два механизма в ядре Linux, когда программы сами могут ограничивать себя через специальные системные вызовы в исполнении системных вызовов и доступе к ресурсам.

## 6.1 Ограничение системных вызовов Seccomp

Seccomp [86] – это механизм ядра Linux, позволяющий процессу необратимо перейти в состояние ограниченного выполнения, в котором он может использовать только строго определённый набор системных вызовов. Основная цель Seccomp – минимизировать поверхность атаки ядра, доступную из пользовательского пространства. Если процесс скомпрометирован после наложения ограничений, Seccomp ограничивает набор действий, которые злоумышленник может выполнить через этот процесс.

Изначально Seccomp был представлен в очень строгом режиме [87], который разрешал процессу использовать только системные вызовы `read()`, `write()`, `_exit()` и `sigreturn()`. Любая попытка вызова другого системного вызова приводила к завершению процесса.

Позднее был добавлен более гибкий режим, который использует Berkeley Packet Filter (BPF) для определения пользовательских политик фильтрации системных вызовов. С помощью BPF-программы процесс может указать, какие системные вызовы разрешены, какие запрещены, а также проверять аргументы системных вызовов. Это позволяет создавать детальные политики безопасности, адаптированные под конкретные нужды приложения. Например, веб-серверу может быть запрещен доступ к сетевым вызовам, кроме тех, что необходимы для его работы, или приложению для обработки изображений может быть запрещен доступ к файлам вне определенной директории.

Ограничение доступных системных вызовов значительно уменьшает количество векторов атак на ядро через скомпрометированный процесс. Помогает изолировать процессы, особенно те, которые обрабатывают недоверенные данные, предотвращая их злонамеренное использование для атаки на другие части системы. Seccomp широко используется в различных системах для повышения безопасности, например, в контейнерных технологиях (Docker, Kubernetes), веб-браузерах (Google Chrome, Firefox) и других приложениях, где важна изоляция и ограничение привилегий.

## 6.2 Ограничение доступа к файлам Landlock

LandLock [88, 89] – это относительно новый механизм безопасности ядра Linux, который позволяет процессам создавать собственные политики контроля доступа к файловой системе без необходимости обладания привилегиями суперпользователя. LandLock также реализует принцип наименьших привилегий. Однако делает это на уровне доступа к файлам. LandLock позволяет процессу определить набор правил, которые ограничивают его доступ к

определенным иерархиям файлов и каталогов. Эти правила применяются к самому процессу и ко всем его дочерним процессам, созданным после активации политики LandLock. Политика определяет, какие типы доступа (например, чтение, запись, выполнение, создание файлов) разрешены для указанных путей в файловой системе.

Он обеспечивает гранулярность, позволяя тонко настраивать права доступа к конкретным файлам и каталогам. Также предлагает проактивную защиту, ограничивая потенциальный ущерб от уязвимостей в приложении, так как даже скомпрометированное приложение не сможет получить доступ к файлам за пределами разрешенной ему области. Кроме того, LandLock может использоваться совместно с другими механизмами безопасности Linux, такими как Seccomp, пространства имен (namespaces) и другими системами контроля доступа, для создания многоуровневой защиты.

LandLock представляет собой перспективный инструмент для усиления безопасности приложений, особенно тех, которые работают с файлами и могут быть подвержены атакам, направленным на несанкционированный доступ к данным. Он позволяет разработчикам встраивать механизмы самоограничения непосредственно в код приложений, повышая их устойчивость к эксплуатации уязвимостей.

## *7. Заключение*

Механизмы усиления защищенности представляют собой критически важный компонент современной информационной безопасности, поскольку они значительно повышают сложность успешной эксплуатации уязвимостей, а в некоторых случаях делают ее практически невозможной. Из проведенного обзора следует, что для обеспечения надежной защиты необходим комплексный подход, охватывающий все уровни системы – от аппаратного до прикладного, так как ни один отдельный механизм не может гарантировать полную безопасность. Большинство современных методов усиления сфокусированы на противодействии определенным классам уязвимостей, что подчеркивает важность применения комбинации различных защитных стратегий для создания многоуровневой и всесторонней защиты, обеспечивающей в том числе целостность системы и изоляцию ее компонентов. При внедрении таких механизмов необходимо тщательно взвешивать компромисс между повышением уровня безопасности и потенциальным снижением производительности системы, поскольку некоторые техники, например, полная инициализация освобождаемой памяти (INIT_ON_FREE) или контроль потока управления (CFI), могут оказывать заметное влияние на производительность.

Интеграции средств защиты с аппаратным обеспечением позволяет снизить накладные расходы и повысить эффективность защит. Кроме того, проактивное планирование мероприятий по усилению защиты требует проведения архитектурного анализа на всех этапах жизненного цикла программного обеспечения, что помогает сделать экономически обоснованный выбор между объектами защиты и применяемыми средствами.

Таким образом, эффективное применение механизмов усиления защищенности требует от специалистов по информационной безопасности глубокого понимания принципов их работы и особенностей реализации в конкретных операционных системах и приложениях. Только всесторонний, многоуровневый подход, учитывающий специфику защищаемой системы и актуальный ландшафт угроз, способен обеспечить достаточный уровень защиты в постоянно меняющихся условиях.

## Список литературы / References

[1]. OpenBSD. Доступно по ссылке: https://www.openbsd.org/, 19.05.2025.

[2]. Linux Kernel Self-Protection Projection (KSPP). Доступно по ссылке: https://kspp.github.io/, 19.05.2025.

[3]. Rutkowska J., Wojtczuk R. Qubes OS architecture. Invisible Things Lab Tech Rep, 2010.

[4]. Grsecurity. Доступно по ссылке: https://grsecurity.net/, 19.05.2025.
[5]. PaX project. Доступно по ссылке: https://pax.grsecurity.net/docs/pax.txt, 19.05.2025.
[6]. GrapheneOS. Доступно по ссылке: https://grapheneos.org/features, 19.05.2025.
[7]. Common Weakness Enumeration: A community-developed list of SW & HW weaknesses that can become. Доступно по ссылке: https://cwe.mitre.org/index.html, 19.05.2025.
[8]. Попов А. Карта средств защиты ядра Linux. Системный Администратор, 2022, Выпуск №3 (232), Доступно по ссылке: https://samag.ru/archive/article/4535, 19.05.2025.
[9]. Xiong W., Lagerström R. Threat modeling – A systematic literature review. Comput. Secur., vol. 84, no. C, pp. 53–69, Jul. 2019, DOI: 10.1016/j.cose.2019.03.010.
[10]. Shevchenko N., Chick T. A., O'Riordan P., Scanlon T. P., Woody C. Threat Modeling: A Summary of Available Methods. 2018.
[11]. Freund J., Jones J. Measuring and Managing Information Risk: A FAIR Approach. Amsterdam: Butterworth-Heinemann, 2015, doi: https://doi.org/10.1016/B978-0-12-420231-3.00001-4.
[12]. Joint Task Force Transformation Initiative Risk management framework for information systems and organizations: a system life cycle approach for security and privacy. National Institute of Standards and Technology, Gaithersburg, MD, NIST SP 800-37r2, Dec. 2018. doi: 10.6028/NIST.SP.800-37r2.
[13]. International Organization for Standardization ISO/IEC 27005:2022 Information security, cybersecurity and privacy protection - Guidance on managing information security risks. 2022.
[14]. De Raadt T. Exploit Mitigation Techniques in OpenBSD. OpenCON, 2005, Доступно по ссылке: https://www.openbsd.org/papers/ven05-deraadt/index.html, 19.05.2025.
[15]. Pax Team Address Space Layout Randomization. 2001, Доступно по ссылке: https://pax.grsecurity.net/docs/aslr.txt, 19.05.2025.
[16]. Molnar I. x86: Enable KASLR by default. 2017, Доступно по ссылке: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=6807c84652b0b7e2e198e50a9ad47ef41b236e59, 19.05.2025.
[17]. De Raadt T. KARL - kernel address randomized link. 2017, Доступно по ссылке: https://marc.info/?l=openbsd-tech&m=149732026405941&w=2, 19.05.2025.
[18]. Molnar I. NX (No eXecute) support for x86. 2004, Доступно по ссылке: https://git.kernel.org/pub/scm/linux/kernel/git/history/history.git/commit/?id=36bc33bac78f6bc08282c622138f4e432b62e7be, 19.05.2025.
[19]. Pax Team PaX pageexec. 2000, Доступно по ссылке: https://pax.grsecurity.net/docs/pageexec.txt, 19.05.2025.
[20]. Pax Team PaX mprotect(). 2000, Доступно по ссылке: https://pax.grsecurity.net/docs/mprotect.txt, 19.05.2025.
[21]. De Raadt T. amd64 kernel W^X. 2015, Доступно по ссылке: https://marc.info/?l=openbsd-tech&m=142120787308107&w=2, 19.05.2025.
[22]. Zijlstra P. module: Harden STRICT_MODULE_RWX. 2020, Доступно по ссылке: https://lore.kernel.org/all/20200403171303.GK20760@hirez.programming.kicks-ass.net/, 19.05.2025.
[23]. Kemerlis V., Polychronakis M., Keromytis A. ret2dir: Rethinking kernel isolation. In 23rd USENIX Security Symposium (USENIX Security 14), pages 957–972, 2014.
[24]. Paris E. Allow Kconfig to set default mmap_min_addr protection. 2007, Доступно по ссылке: https://www.mail-archive.com/linux-security-module@vger.kernel.org/msg02399.html, 19.05.2025.
[25]. Intel Corporation INTEL64 AND IA-32 ARCHITECTURES SOFTWARE DEVELOPER'S MANUAL. Instruction Set Extensions Programming Reference. 2013.
[26]. George V., Piazza T., Jiang H.: Technology Insight: *Intel Next Generation Microarchitecture*. Codename Ivy Bridge, IDF 2011.
[27]. Yu F. Enable/Disable Supervisor Mode Execution Protection. 2011, Доступно по ссылке: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=de5397ad5b9ad22e2401c4dacdf1bb3b19c05679, 19.05.2025.
[28]. Corbet, J. Supervisor mode access prevention. 2012. Доступно по ссылке: http://lwn.net/Articles/517475/, 19.05.2025.
[29]. Spengler B. SSTIC 2016 Keynote. Rennes, France, 2016, Доступно по ссылке: https://grsecurity.net/SSTIC2016.pdf#page=3, 19.05.2025.
[30]. LWN Randomizing structure layout. 2017, Доступно по ссылке: https://lwn.net/Articles/722293/, 19.05.2025.
[31]. Serebryany, K., Bruening, D., Potapenko, A., & Vyukov, D. AddressSanitizer: A Fast Address Sanity Checker. USENIX ATC, 2012.

[32]. The Kernel Address Sanitizer (KASAN). Доступно по ссылке: www.kernel.org/doc/html/v4.10/dev-tools/kasan.html, 19.05.2025.
[33]. Elver M. KFENCE: A low-overhead sampling-based memory safety error detector. 2020, Доступно по ссылке: https://lore.kernel.org/all/20201103175841.3495947-1-elver@google.com/, 19.05.2025.
[34]. ARM, ARMv8.5-A Memory Tagging Extension.
[35]. Konovalov A. kasan: add hardware tag-based mode for arm64. 2020, Доступно по ссылке: https://lwn.net/Articles/838211/, 19.05.2025.
[36]. Zhao Q. Security Improvements in GCC. Linux Plumbers Conference, 2021, Доступно по ссылке: https://lpc.events/event/11/contributions/1001/, 19.05.2025.
[37]. Cook K. hardening: Introduce CONFIG_ZERO_CALL_USED_REGS. Доступно по ссылке: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=a82adfd5c7cb, 19.05.2025.
[38]. Jelinek J. Object size checking to prevent (some) buffer overflows. 2004, Доступно по ссылке: https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html, 19.05.2025.
[39]. Cook K. mm: Hardened usercopy. 2016, Доступно по ссылке: https://lwn.net/Articles/691012/, 19.05.2025.
[40]. Sidhpurwala H. Hardening ELF binaries using Relocation Read-Only (RELRO). Доступно по ссылке: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro, 19.05.2025.
[41]. Cowan C., Pu C., Maier D., Hintony H., Walpole J., Bakke P., Beattie S., Grier A., Wagle P., Zhang Q. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. Proceedings of the 7th conference on USENIX Security Symposium, 1998.
[42]. Wagle P., Cowan C. StackGuard: Simple Stack Smash Protection for GCC. 2003.
[43]. Etoh H., Yoda K. Propolice: Protecting from stack-smashing attacks. Technical Report, IBM Research Division, Tokyo Research Laboratory, 2000.
[44]. Yu R., Nin F. D., Zhang Y., Huang S., Kaliyar P., Zakto S., Conti M., Portokalidis G., Xu J. Building Embedded Systems Like It's 1996. Proceedings 2022 Network and Distributed System Security Symposium, DOI: 10.14722/ndss.2022.24031.
[45]. Bierbaumer, B., Kirsch, J., Kittel, T., Francillon, A., Zarras, A. Smashing the Stack Protector for Fun and Profit. SEC 2018, IFIP Advances in Information and Communication Technology, vol 529.2 2018 Springer, Cham. DOI: https://doi.org/10.1007/978-3-319-99828-2_21.
[46]. Depuydt H., Gülmez M., Nyman T., Mühlberg J. T. Do we still need canaries in the coal mine? Measuring shadow stack effectiveness in countering stack smashing. 2024, DOI: 10.48550/arXiv.2412.16343.
[47]. Delalleau G. Large memory management vulnerabilities. CanSecWest, 2005. Доступно по ссылке: https://cdn.atraining.ru/docs/memory_vulns_delalleau.pdf, 19.05.2025.
[48]. Wojtczuk R. Exploiting large memory management vulnerabilities in Xorg server running on Linux. 2010. Доступно по ссылке: https://invisiblethingslab.com/resources/misc-2010/xorg-large-memory-attacks.pdf, 19.05.2025.
[49]. Qualys Security Advisory: The Stack Clash. 2017. Доступно по ссылке: https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt, 19.05.2025.
[50]. Torvalds L. mm: keep a guard page below a grow-down stack segment. 2010. Доступно по ссылке: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=320b2b8de12698082609ebbc1a17165727f4c893, 19.05.2025.
[51]. Dickins H. mm: larger stack guard gap, between vmas. 2017. Доступно по ссылке: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=1be7107fbe18eed3e319a6c3e83c78254b693acb, 19.05.2025.
[52]. Law J. Stack clash mitigation, 2017. Доступно по ссылке: https://gcc.gnu.org/legacy-ml/gcc-patches/2017-07/msg00556.html, 19.05.2025.
[53]. Guelton S., Ledru S., Stone J. Bringing Stack Clash Protection to Clang / X86 – the Open Source Way. 2021. Доступно по ссылке: https://blog.llvm.org/posts/2021-01-05-stack-clash-protection/, 19.05.2025.
[54]. Lutomirski A. Virtually mapped stacks with guard pages (x86, core). 2016. Доступно по ссылке: https://lkml.org/lkml/2016/6/15/1064, 19.05.2025.
[55]. Cook K. VLA removal for v4.20-rc1. 2018, Доступно по ссылке: https://lkml.org/lkml/2018/10/28/189, 19.05.2025.
[56]. Tomlin A. sched: Always check the integrity of the canary. Доступно по ссылке: https://lore.kernel.org/all/1410527779-8133-1-git-send-email-atomlin@redhat.com/, 19.05.2025.
[57]. Pax Team, PaX - kernel self-protection. H2HC, 2012. Доступно по ссылке: https://pax.grsecurity.net/docs/PaXTeam-H2HC12-PaX-kernel-self-protection.pdf, 19.05.2025.

[58]. Popov A. STACKLEAK: A Long Way to the Linux Kernel Mainline. Linux Security Summit 2018, Vancouber, Canada, 2018.
[59]. Cho H., Park J., Kang J., Bao T., Wang R., Shoshitaishvili Y., Doupé A., Ahn G.-J. Exploiting Uses of Uninitialized Stack Variables in Linux Kernels to Leak Kernel Pointers. WOOT 2020. Доступно по ссылке: https://www.usenix.org/conference/woot20/presentation/cho, 19.05.2025.
[60]. MISRA, MISRA C:2012 – Guidelines for the use of the C language in critical systems. MIRA Ltd, Nuneaton, Warwickshire CV10 0TU, UK (Mar 2013).
[61]. Pax Team, PaX - GCC plugins galore. H2HC, 2013. Доступно по ссылке: https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf, 19.05.2025.
[62]. Cook K. gcc-plugins: Add structleak for more stack initialization. 2017, Доступно по ссылке: https://lore.kernel.org/kernel-hardening/20170113220256.GA57663@beast/T/#u, 19.05.2025.
[63]. Zhao Q. Security Improvements in GCC. Linux Plumbers Conference, 2021. Доступно по ссылке: https://lpc.events/event/11/contributions/1001/, 19.05.2025.
[64]. Zhao Q. add -ftrivial-auto-var-init and variable attribute uninitialized to gcc. GCC Patch Mailing List, February 2021, Доступно по ссылке: https://gcc.gnu.org/pipermail/gcc-patches/2021-February/565514.html, 19.05.2025.
[65]. Cook K. security: Implement Clang's stack initialization. 2019. Доступно по ссылке: https://lore.kernel.org/all/20190411180117.27704-4-keescook@chromium.org/, 19.05.2025.
[66]. Pax Team, RANDKSTACK. Доступно по ссылке: https://pax.grsecurity.net/docs/randkstack.txt, 19.05.2025.
[67]. Reshetova E. Randomize kernel stack offset upon syscall. 2019, Доступно по ссылке: https://lwn.net/Articles/785484/, 19.05.2025.
[68]. Shacham, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls. ACM CCS, 2007, DOI: 10.1145/1315245.1315313.
[69]. Garnier T. mm: SLAB freelist randomization. 2016, Доступно по ссылке: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c7ce4f60ac199fb3521c5fcd64da21cee801ec2b, 19.05.2025.
[70]. Cook K. mm: Add SLUB free list pointer obfuscation. 2017, Доступно по ссылке: https://lore.kernel.org/all/20170802180609.GA66807@beast/T/#m5fc98c10905ea25cf6123e723bca8162226d53c4, 19.05.2025.
[71]. Cook K. slab: Introduce dedicated bucket allocator. 2024, Доступно по ссылке: https://lwn.net/Articles/978976/, 19.05.2025.
[72]. Potapenko A. add init_on_alloc/init_on_free boot options. 2019, Доступно по ссылке: https://lore.kernel.org/all/20190628093131.199499-2-glider@google.com/T/#u, 19.05.2025.
[73]. Ruiqi G. Randomized slab caches for kmalloc(). 2023, Доступно по ссылке: https://lore.kernel.org/all/20230714064422.3305234-1-gongruiqi@huaweicloud.com/, 19.05.2025.
[74]. Abadi M., Budiu M., Erlingsson Ú., Ligatti J. Control-flow integrity. Proceedings of the 12th ACM conference on Computer and communications security. Alexandria VA USA: ACM, Nov. 2005, pp. 340–353. DOI: 10.1145/1102120.1102165.
[75]. Abadi M., Budiu M., Erlingsson Ú., Ligatti J. Control-flow integrity principles, implementations, and Applications. ACM Trans. Inf. Syst. Secur., vol. 13, no. 1, pp. 1–40, Oct. 2009, DOI: 10.1145/1609956.1609960.
[76]. Clercq R. D., Verbauwhede I. A survey of Hardware-based Control Flow Integrity (CFI). 2017.
[77]. Intel Corporation Control-flow Enforcement Technology Specification. 2021.
[78]. Shanbhogue V., Gupta D., Sahita R. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy. Phoenix AZ USA: ACM, Jun. 2019, pp. 1–11. DOI: 10.1145/3337167.3337175.
[79]. Tice C., Roeder T., Collingbourne P., Checkoway S., Erlingsson Ú., Lozano L., Pike G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. USENIX Security 14, 2014, pp. 941–955. Доступно по ссылке: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice, 19.05.2025.
[80]. Bento F. M. Control-Flow Integrity for the Linux kernel: A Security Evaluation. 2019.
[81]. Unified Extensible Firmware Interface Specification. Version 2.7, May 2017, Доступно по ссылке: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_7.pdf, 19.05.2025.

[82]. Hagl J., Mann O., Pirker M. Securing the Linux Boot Process: From Start to Finish. Proceedings of the 7th International Conference on Information Systems Security and Privacy, 2021, pp. 604–610. DOI: 10.5220/0010313906040610.
[83]. Safford D. An Overview of The Linux Integrity Subsystem. Whitepaper, Доступно по ссылке: https://cyfuture.dl.sourceforge.net/project/linux-ima/linux-ima/Integrity_overview.pdf, 19.05.2025.
[84]. Zohar M. EVM. 2011, Доступно по ссылке: https://lore.kernel.org/all/1309377038-4550-1-git-send-email-zohar@linux.vnet.ibm.com/, 19.05.2025.
[85]. Garrett M. security: Add a static lockdown policy LSM. Доступно по ссылке: https://lore.kernel.org/linux-security-module/20190820001805.241928-4-matthewgarrett@google.com/, 19.05.2025.
[86]. Edge J. A seccomp overview, 2015. Доступно по ссылке: https://lwn.net/Articles/656307/, 19.05.2025.
[87]. Arcangeli A. seccomp for 2.6.11-rc1-bk8. 2005, Доступно по ссылке: https://lwn.net/Articles/120192/, 19.05.2025.
[88]. Salaün M. Landlock: From a security mechanism idea to a widely available implementation. 2024, Доступно по ссылке: https://landlock.io/talks/2024-06-06_landlock-article.pdf, 19.05.2025.
[89]. Salaün M. Landlock LSM. 2021, Доступно по ссылке: https://lore.kernel.org/linux-security-module/20210422154123.13086-1-mic@digikod.net/, 19.05.2025.

## *Информация об авторах / Information about authors*

Денис Валентинович ЕФРЕМОВ – старший научный сотрудник. Сфера научных интересов: формальная верификация, статический и динамический анализ.

Denis Valentinovich EFREMOV – senior researcher. Research interests: formal verification, static and dynamic analysis.

Александр Константинович ПЕТРЕНКО – доктор физико-математических наук, профессор, заведующий отделом Технологий программирования ИСП РАН, профессор кафедр Системного программирования ВМК МГУ и ФКН НИУ ВШЭ. Научные интересы: формальные методы программной инженерии, языки спецификаций и моделирования, верификация.

Alexander Konstantinovich PETRENKO – Dr. Sci. (Phys.-Math.), Prof., Head of Software Engineering Department of the Ivannikov Institute for System Programming, Russian Academy of Sciences, Professor of MSU and the Faculty of Computer Science, NUY HSE. Research their use in software development and verification.

Борис Аронович ПОЗИН – доктор технических наук, профессор, главный научный сотрудник Института системного программирования им. В.П. Иванникова РАН, профессор базовой кафедры ЗАО ЕС-лизинг в МИЭМ НИУ ВШЭ, технический директор ЗАО ЕС-лизинг. Сфера научных интересов: программная инженерия, системы обеспечения жизненного цикла доверенного программного обеспечения, автоматизированное тестирование программ.

Boris Aronovich POZIN – Dr. Sci. (Tech.), Prof., Chief Researcher at the Ivannikov Institute for System Programming of the Russian Academy of Sciences, Professor of the Basic Department of CJSC EC-Leasing at the Higher School of Economics, Technical Director of CJSC EC-Leasing. Research interests: software engineering, life cycle ensuring systems for trusted software, automated software testing.

Виталий Адольфович СЕМЕНОВ – доктор физико-математических наук, профессор, заведующий отделом системной интеграции и прикладных программных комплексов Института системного программирования им. В.П. Иванникова РАН с 2015 года. Сфера научных интересов: модельно-ориентированные методологии и инструменты программной инженерии для создания цифровых платформ и мультидисциплинарных программных комплексов, визуализация и компьютерная графика, технологии информационного

моделирования в архитектуре и строительстве, проектное управление и календарно-сетевое планирование.

Vitaly Adolfovich SEMENOV – Dr. Sci. (Phys.-Math.), Prof., Head of the Department of System Integration and Multi-disciplinary Applied Systems of the Ivannikov Institute for System Programming of the RAS since 2015. Research interests: model-driven methodologies and CASE toolkits for creating digital platforms and advanced applied systems, visualization and computer graphics, building information modeling, project management and scheduling.