УДК 004.052.42 DOI: 10.31114/2078-7707-2022-4-32-36

Тенденции внедрения дескрипторов памяти процессоров и анализаторов для верификации программного обеспечения

Семенов С.А.

Публичное акционерное общество «Институт электронных управляющих машин имени И.С. Брука», г. Москва

reply.s@yandex.ru

***Аннотация* — В статье рассмотрены отечественные и зарубежные технологии аппаратной поддержки вычислений процессоров и анализаторов программного обеспечения как методов снижения уязвимостей памяти. Приведены основные архитектурные отличия таких технологий защиты как Эльбрус, CHERI и Arm MTE. Исследованы существующие статические и динамические программные анализаторы на предмет методологии работы и преимуществ по выявлению дефектов в программном коде.**

***Ключевые слова* — дескрипторы памяти, тегированная память, Эльбрус, CHERI, Arm MTE, статический анализ кода, Valgrind, Svace.**

I. Введение

Стремительный рост развития информационных технологий обуславливает повышение сложности программного обеспечения, что влечёт за собой снижение его надежности и увеличение количества ошибок и уязвимостей. Таким образом, спровоцированный рост уровня киберпреступности и кибератак, наряду с проблемой использования аппаратных архитектур, не имеющих средств повышения надежности, ставят вопрос защищенности программ на ведущее место в планах дальнейшего развития важных инфраструктур.

Наряду с известной проблемой разработки безопасных программ, вполне оправданно направлять усилия на исправление уязвимостей, проявление которых носит наиболее вероятный характер. Зарубежные компании, такие как Microsoft Corporation, [1] и Google [2], сообщают, что максимальная доля всех проблем безопасности, вызваны именно нарушениями безопасности памяти. Попытки решать проблему совершенствованием системы государственной стандартизации и сертификации программных продуктов не гарантируют отсутствие потенциальных уязвимостей и подтверждают наличие проблемы создания безопасного кода и эффективного метода поиска ошибок в нём. Очевидным преимущественным подходом обеспечить надежность программы — это не допустить возникновения ошибок. С этой целью широко применяют способы безопасного программирования, направленного на предотвращение появления и устранение уязвимостей [3].

В вопросе безопасного программирования сегодня создано и имеют своё развитие несколько подходов, которые в общей своей сути могут быть отнесены к аппаратным или программным решениям.

II. Обзор аппаратных решений

Много лет разработчики искали эффективное решение, чтобы добиться безопасной реализации «небезопасных языков» как программными, так и программно-аппаратными средствами.

Примером уникального подхода аппаратной поддержки безопасных вычислений в отечественной микропроцессорной технике является архитектура «Эльбрус», основоположником которой является Борис Арташесович Бабаян. Механизм аппаратной защиты памяти в «Эльбрусе» поддерживает компилятор, формирующий код с размером адреса 128 бит, что позволяет исполнять программу с использованием аппаратной поддержки разделения контекста
и контроля доступа к данным.

В архитектуре используется тегированная память. Тег не может быть модифицирован на пользовательском уровне. Для доступа в память используется расширенный до 128 бит указатель, называемый дескриптором. Кроме собственно адреса объекта, дескриптор содержит размер объекта. смещение и специальный тег дескриптора. Это дает возможность аппаратуре контролировать ошибки при обращении в память (например, выход за границу массива). Неинициализированная память метится тегом «пусто», что позволяет отследить использование значения переменных до их инициализации [4].

Создание и использование дескрипторов осуществляются под строгим контролем аппаратуры. Реализуется это требование таким образом, что объекты создаются исключительно операционной системой. При этом в отличие от традиционных указателей, набор операций над дескриптором существенно ограничен. В случае применения к дескриптору некорректной операции (например,

умножения), дескриптор теряет тег дескриптора и превращается в целочисленное значение, доступ к памяти через которое теряется (приводит к прерыванию). На пользовательском уровне сформировать дескриптор из произвольного адреса невозможно [5].

Реализация базируется на семантических основах безопасности языков программирования и включает компоненты, которые обеспечивают контекстную защиту, модульный подход, контроль соответствия данных, кода и интерфейса функции, контроль границ данных и кода и т.д.

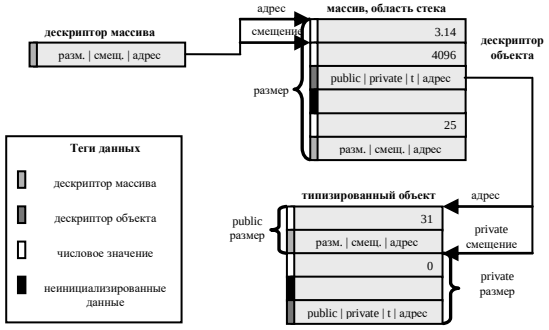

Рис. 1. Использование тегированной памяти для защиты данных

Дескрипторы массивов и объектов, числа и неинициализированные данные легко различимы в памяти по своим тегам. Смещение в дескрипторе массива соответствует позиции указателя, области public и private данных в объекте разделены и защищены размерами, а поле t содержит номер класса объекта (рис. 1). Разработчики современной методологии безопасных вычислений утверждают, что аппаратная поддержка является обязательной, поскольку исключительно программными средствами невозможно защититься от подделки указателей или от нарушения границ объектов. В то же время за формирование контекста каждой точки программы отвечают компилятор, реализующий семантику языка, и редактор связей, объединяющий отдельные единицы компиляции в готовые к выполнению программы. Операционная система обеспечивает выделение памяти, формирование ссылок на объекты и другие важные механизмы [6].

Альтернативным зарубежным решением является совместный исследовательский проект CHERI от SRI International и Кембриджского университета, целью которого является пересмотр основных вариантов проектирования аппаратного и программного обеспечения для значительного повышения безопасности системы. Это универсальная технология для построения доверенных аппаратно-программных платформ, являющаяся аналогом режима защищенных вычислений платформы «Эльбрус». Особенностью является наличие тегированной памяти в размере 1 бита на слово и мандат-дескрипторы, которые в свою очередь содержат маски разрешений, флаги и тип объекта.

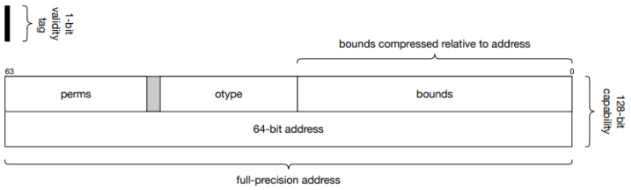

Рис. 2. 128-битное представление дескриптора CHERI включающее 64-битный адрес, метаданные в адресуемой памяти и 1-битный тег вне диапазона

Кембриджский университет объясняет, что CHERI расширяет традиционные аппаратные архитектуры наборов инструкций (ISA) новыми функциями, чтобы обеспечить мелкозернистую защиту памяти и масштабируемую компартментализацию программного обеспечения. В зависимости от варианта архитектуры команд технология спроектирована для 64, 128 и 256 разрядной адресации. По заявлениям SRI International и Кембриджского университета одним из следующих направлений в развитии проекта является его адаптация для открытой архитектуры RISC-V [7]

Стоит отметить, что вышеописанные решения обладают серьезным недостатком, затрудняющим широкое применение этих архитектур для повышения безопасности программы. Это недостаточная совместимость по исходному коду.

Еще одной конкурентной и стремительно развивающейся технологией является Memory Tagging Extension (MTE) от ARM, предоставляющая производительное и масштабируемое аппаратное решение, которое снижает вероятность использования нарушений безопасности памяти, присутствующие в коде, написанном на «небезопасных языках». Реализация этого механизма доступа к памяти осуществляется по принципу "ключ-замок" (lock & key). Замок устанавливается на память и доступ к ней потребует ключа. Если ключ подходит к замку, то доступ разрешается. В противном случае формируется сообщение об ошибке [8-9].

Области памяти помечаются путем добавления 4х битов метаданных для каждых 16 байтов физической памяти. Эти четыре бита формируют Tag Granule. Тег и есть реализация замка. Указатели (а, следовательно, и виртуальные адреса) модифицируются таким образом, чтобы в них содержался ключ. Когда память выделяется или освобождается, ей присваивается тег блокировки и тогда весь доступ должен осуществляться по адресу с тем же тегом (ключом).

MTE предоставляет механизм, который позволяет обнаруживать ошибки типа use-after-free и out-of-bounds. Кроме того, он был разработан таким образом, что для большинства приложений не требуется модификации исходного кода. Доказательством эффективности такой аппаратной реализации служит новость, о том Google объявил о внедрении MTE от Arm в Android [10]. За счет незначительного отступления от концепции полной защиты памяти в пользу вероятностной защиты с

большой степень защищенности технология MTE решает проблему совместимости по исходному коду.

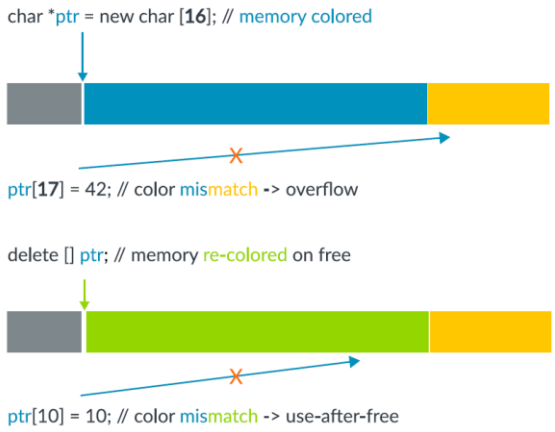

Рис. 3. Пример блокировки и доступа к памяти

Общей тенденцией в развитии перечисленных архитектурных решений является нацеленность разработчиков на оценку и изучение следующих вопросов:

- влияние технологии на производительность программного обеспечения;
- совместимость существующего программного обеспечения исходному коду;
- выбор оптимальной микроархитектуры;
- повышение безопасности кода.

III. Программные анализаторы

Исключительно программными средствами поддержки отладки и технической реализации безопасной работы с памятью являются программные анализаторы. Считается, что их работа приводит к резкому снижению производительности программ, но использование анализаторов — это объективная потребность для обеспечения безопасности современных программ и в связи с увеличением сложности и объемности последних популярность анализаторов будет только расти. Целью их применения стоит поиск дефектов различного типа включая переполнение буфера, утечку памяти, использование неинициализированных переменных, использование объектов после удаления, несогласованность ошибки деления на ноль, возвращение адреса локальных переменных [11].

Технологии поиска дефектов программы лежат в плоскости трёх основных направлений своего развития. Первая группа — это системы автоматического поиска ошибок с помощью статического анализа исходного кода, которые можно применять на самых ранних этапах разработки, что делает исправление дефектов максимально дешевым. Второе – это системы динамического анализа бинарного кода, позволяющие многократно запускать заданную программу на автоматически генерируемом наборе входных данных и отслеживать ситуации возникновения дефектов. К последней группе, относятся технологии защиты памяти приложения и его окружения от эксплуатации имеющихся в коде уязвимостей во время работы программы [12].

Что касается статического анализа кода, то пусть он и не выявляет все недостатки в программах, но вместе с тем имеет относительную точность, эффективность и скорость при выявлении ошибок, комплексность тестирования и возможность обнаружения ошибок, которые могут не выявляться при динамическом тестировании [13].

Неполный список известных программных средств инструментирования, осуществляющих эффективный анализ и фаззинг, составляют Valgrind, S2E, Avalanche, Mayhem, Xandra, Mechanical Phish, PVS-Studio. Инструменты обеспечивают возможности анализа слабых мест в исходном коде, программ пользователя, библиотек, ядра, драйверов и возможность анализа бинарного кода. К примеру, один из известных и мощных инструментов с открытым исходным кодом Valgrind исследует поведение приложения в ходе его работы. Он транслирует бинарные файлы в промежуточное представление. Это позволяет отслеживать вызовы функций выделения и освобождения памяти, какие участки памяти зарезервированы, не выходят ли они за границу выделенной области и т.п.

Побочными и не уникальными эффектами от применения подобных продуктов являются:

- медленная скорость работы программы;
- неоднозначный результат анализа ошибок;
- сложность анализа работы инструмента.

Не последним остается проблема полноты тестирования возможных сценариев работы программы [14].

Отдельного внимания заслуживает инструмент Svace [15], осуществляющий статический поиск дефектов, используя 2 вида анализаторов: легковесный и основной. Легковесный анализатор осуществляет поиск дефектов просматривая абстрактное синтаксическое дерево. Основной, в свою очередь, осуществляет межпроцедурный потоково- и контекстно-чувствительный анализ. Таким образом, по заявлениям разработчика, обеспечивается высокое качество анализа, масштабируемость, удобный интерфейс просмотра предупреждений и другие преимущества.

Большая часть статических анализаторов измеряет качество программного кода на основе соответствующих метрик, включающие такие потенциальные ошибки как повторение участков кода, неиспользуемые или неизменные параметры, неинициализированные переменные, стиль программирования и т.д. [3]. Разные инструменты используют различные алгоритмы поиска дефектов и

способы борьбы с ложными срабатываниями. Нередко для более эффективного выявления ошибок анализаторы программного кода могут использоваться совместно с применением аппаратных решений.

IV. Заключение

В работе приведен обзор существующих решений по ограничению проявления уязвимостей памяти в программном обеспечении. Показаны основные отличия, недостатки каждой технологии и тенденции развития аппаратных и программных методов поиска дефектов.

Литература

[1] Matt Miller. Microsoft Security Response Center. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. [презентация] 2019. – URL: https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf (Дата обращения: 05.05.2022).

[2] J.V. Stoep. Queue the Hardening Enhancements. / J.V. Stoep, Android Security & Privacy Team and Chong Zhang, Android Media Team // Google Security Blog – 2019. URL: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (Дата обращения: 05.05.2022).

[3] Воротникова, Т. Ю. Надежный код: статический анализ программного кода как средство повышения надежности программного обеспечения информационных систем / Т. Ю. Воротникова // Информационные технологии в УИС. – 2020. – № 2. – С. 22-27.

[4] Мустафин Т.Р. Безопасная среда исполнения критических приложений во встраиваемых системах на базе вычислительных средств семейства «Эльбрус» / Т.Р. Мустафин, А.И. Алехин, Е.М. Кравцунов, Б.О. Макаев // Радиопромышленность. 2019. Т. 29. № 1. С.16-22.

[5] Микропроцессоры и вычислительные комплексы семейства «Эльбрус».: [учеб. пособие] / А.К. Ким, В.И. Перекатов, С.Г. Ермаков [и др.]; под ред. А.Кривцова – Санкт-Петербург.: Питер, 2013. 272 с.

[6] Волконский, В. Ю. Безопасная реализация языков программирования на базе аппаратной и системной поддержки / В. Ю. Волконский // Вопросы радиоэлектроники. – 2008. – Т. 4. – № 2. – С. 98-141.

[7] R.N.M. Watson, An Introduction to CHERI. Technical Report. / R.N.M. Watson, S.W. Moore, P. Sewell, P.G. Neumann // 2019 – URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf (Дата обращения: 05.05.2022).

[8] ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. Konstantin Serebryany. SECURITY. SUMMER 2019 VOL. 44, NO. 2

[9] https://developer.arm.com/Architectures/A-Profile%20Architecture# Technical-Information (Дата обращения: 05.05.2022).

[10] https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html (Дата обращения: 05.05.2022).

[11] Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Бородин А.Е., Белеванцев А.А.. Труды ИСП РАН, том 27, вып. 6, 2015 г., с. 111-134

[12] Технологии статического и динамического анализа уязвимостей программного обеспечения. Аветисян А.И., Белеванцев А.А., Чукляев И.И. Вопросы кибербезопасности №3(4) – 2014.

[13] Обзор инструментов статического анализа программного кода. Кусаинов А.Р., Глазырина Н.С. Colloquium-Journal. 2020. #32(84), Technical Sciences. С.48-52.

[14] Силаков, Д. Качество программного кода. От выявления стилистических огрехов к поиску ошибок / Д. Силаков // Системный администратор. – 2014. – № 3(136). – С. 80-84.

[15] А.Е. Бородин, А.А. Белеванцев. Статический анализатор Svace как коллекция анализаторов разных уровней сложности. Труды ИСП РАН, том 27, вып. 6, 2015 г., с. 111-134

Trends in the Implementation of Processor Memory Descriptors and Analyzers for Software Verification

S.A. Semenov

Public Joint Stock Company "Institute of Electronic Control Machines named after I.S. Bruk", Moscow

reply.s@yandex.ru

***Abstract* — In the issue of secure coding, several approaches have been created and developed, they can be either hardware or software solutions. An example of a unique approach of hardware support for secure coding is domestic microchip technology using Elbrus architecture. The hardware memory protection mechanism supports a compiler that generates code with an address size of 128 bits, which allows you to run a program using hardware support for context separation and data access control. The architecture supports tagged memory. Tag-level permission system do not allow user to modify the tag.**

An alternative solution is the CHERI project from SRI International and the University of Cambridge. A special feature of this architecture is one bit in a word and mandatory descriptors, which contain permission mask, flag

and object type. CHERI extends conventional hardware instruction-Set architectures (ISA's) with new features to enable fine-grained memory protection and highly scalable software compartmentalization. Another competitive technology is ARM Memory Tagging Extension (MTE), which provides a productive and scalable hardware solution, that reduces the likelihood of using memory security breaches in code, written in memory-unsafe programming languages. This mechanism of memory access is called the locks-and-keys.

Software analyzers are software-only tools that support debuggers and memory-safe programming. They are used to search for various defect types, including buffer overflow, use of uninitialized variable, memory leaks, inconsistent/division by zero error, returning address of local variable and objects use after deletion. Software defect detection technologies are static analyzers, dynamic analyzers and application memory protection technologies from exploiting vulnerabilities in the code while the program is running.

Different tools use different algorithms to find defects and ways to deal with false positives. Often, for more effective error detection, program code analyzers and hardware solutions are used together.

***Keywords* — memory descriptors, tagged memory, Elbrus, CHERI, Arm MTE, static code analysis, Valgrind, Svace.**

REFERENCES

[1] Matt Miller. Microsoft Security Response Center. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. [presentation] 2019. – URL: https://msrnd-cdn-stor.azureedge.net/bluehat/bluehatil/2019/assets/doc/Trends%2C%20Challenges%2C%20and%20Strategic%20Shifts%20in%20the%20Software%20Vulnerability%20Mitigation%20Landscape.pdf (Access date: 05.05.2022).

[2] J.V. Stoep. Queue the Hardening Enhancements. / J.V. Stoep, Android Security & Privacy Team and Chong Zhang, Android Media Team // Google Security Blog – 2019. URL: https://security.googleblog.com/2019/05/queue-hardening-enhancements.html (Access date: 05.05.2022).

[3] Vorotnikova, T. YU. Nadezhnyj kod: staticheskij analiz programmnogo koda kak sredstvo povysheniya nadezhnosti programmnogo obespecheniya informacionnyh sistem (Reliable code: static analysis of program code as a means of improving the reliability of information systems software)/ T. YU. Vorotnikova // Informacionnye tekhnologii v UIS. – 2020. – № 2. – S. 22-27.

[4] Mustafin T.R. Bezopasnaya sreda ispolneniya kriticheskih prilozhenij vo vstraivaemyh sistemah na baze vychislitel'nyh sredstv semejstva «El'brus» (Safe execution environment for critical applications in embedded systems based on computing tools of the Elbrus family) / T.R. Mustafin, A.I. Alekhin, E.M. Kravcunov, B.O. Makaev // Radiopromyshlennost'. 2019. T. 29. № 1. C.16-22.

[5] Mikroprocessory i vychislitel'nye kompleksy semejstva «El'brus» (Microprocessors and computer systems of the Elbrus family).: [ucheb. posobie] / A.K. Kim, V.I. Perekatov, S.G. Ermakov [i dr.]; pod red. A.Krivcova – Sankt-Peterburg.: Piter, 2013. 272 s.

[6] Volkonskij, V. YU. Bezopasnaya realizaciya yazykov programmirovaniya na baze apparatnoj i sistemnoj podderzhki (Safe implementation of programming languages based on hardware and system support) / V. YU. Volkonskij // Voprosy radioelektroniki. – 2008. – T. 4. – № 2. – S. 98-141.

[7] R.N.M. Watson, An Introduction to CHERI. Technical Report. / R.N.M. Watson, S.W. Moore, P. Sewell, P.G. Neumann // 2019 – URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf (Access date: 05.05.2022).

[8] ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. Konstantin Serebryany. SECURITY. SUMMER 2019 VOL. 44, NO. 2

[9] https://developer.arm.com/Architectures/A-Profile%20Architecture #Technical-Information (Access date: 05.05.2022).

[10] https://security.googleblog.com/2019/08/adopting-arm-memory-tagging-extension.html (Access date: 05.05.2022).

[11] Staticheskij analizator Svace kak kollekciya analizatorov raznyh urovnej slozhnosti (Svace static analyzer as a collection of analyzers of different complexity levels). Borodin A.E., Belevancev A.A.. Trudy ISP RAN, tom 27, vyp. 6, 2015 g., c. 111-134

[12] Tekhnologii staticheskogo i dinamicheskogo analiza uyazvimostej programmnogo obespecheniya (Technologies for static and dynamic analysis of software vulnerabilities). Avetisyan A.I., Belevancev A.A., CHuklyaev I.I. Voprosy kiberbezopasnosti №3(4) – 2014.

[13] Obzor instrumentov staticheskogo analiza programmnogo koda (Overview of Static Code Analysis Tools). Kusainov A.R., Glazyrina N.S. Colloquium-Journal. 2020. #32(84), Technical Sciences. S.48-52.

[14] Silakov, D. Kachestvo programmnogo koda. Ot vyyavleniya stilisticheskih ogrekhov k poisku oshibok (The quality of the software code. From identifying stylistic flaws to finding errors) / D. Silakov // Sistemnyj administrator. – 2014. – № 3(136). – S. 80-84.

[15] A.E. Borodin, A.A. Belevancev. Staticheskij analizator Svace kak kollekciya analizatorov raznyh urovnej slozhnosti (Svace static analyzer as a collection of analyzers of different complexity levels). Trudy ISP RAN, tom 27, vyp. 6, 2015 g., c. 111-134