УДК 004

Изолированная среда для запуска недоверенных процессов и кода в контексте информационной безопасности веб-ориентированных приложений

Фатеенков Данила Витальевич
Приамурский государственный университет имени Шолом-Алейхема
Студент

Аннотация
В статье рассматривается процесс внедрения модуля для создания изолирующей среды в веб-ориентированное приложение. Модуль предназначен для обеспечения безопасности запуска сторонних процессов на стороне сервера и недопуска утечек памяти, ошибок ожидания и слишком долго выполнения процесса. Также в статье описаны возможные проблемы при работе с недоверенными процессами в веб-приложениях и к чему они могут привести, приведены примеры создания тех или иных ошибок на различных языках программирования. Итогом является модуль, обеспечивающий безопасность работы сервера с процессами на различных языках программирования.
Ключевые слова: изолированная среда, режим песочницы, информационная безопасность, веб-фреймворки, C, работа с процессами операционной системы, Docker

Isolated environment for running untrusted code and processes in the context of information security of web-oriented applications

Fateenkov Danila Vitalievich
Sholom-Aleichem Priamursky State University
Student

Abstract
The article discusses the process of implementing a module for creating an isolation environment in a web-oriented application. The module is designed to provide security for launching third-party processes on the server side and prevent memory leaks, waiting errors and too long process execution. The article also describes possible problems when working with untrusted processes in web applications and what they can lead to, gives examples of creating these or those errors in different programming languages. The result is a module that provides security of server work with processes in different programming languages.
Keywords: isolated environment, sandboxing, information security, web frameworks, C, working with operating system processes, Docker

1. Введение

1.1 Актуальность

Обеспечение безопасности работы серверных приложений и сред является в настоящее время одной из наиболее актуальных задач. Связано это, в первую очередь, с быстрым развитием веб-ориентированной сферы: ежедневно в сети передаются терабайты информации между клиентами и серверами различных приложений. Информация также может быть представлена в различном виде: текстовая, медиа, техническая и др.

Стоит также отметить, что не все сервера создаются или арендуются с целью передачи информации между пользователями. На стороне сервера также можно осуществлять запуск приложений и процессов обработки различных данных. Это может быть актуально для приложений различных направленностей: от дата-центров, предназначенных для обработки больших объёмов данных, до обучающих систем (например, обучение пользователей сайта программированию). Запуск сторонних процессов (например, компиляция исходного кода некоторого приложения с последующим запуском) является серьёзной уязвимостью в безопасности системы, которая может привести к нежелательным последствиям.

Один из способов обеспечить безопасность во время запуска недоверенных процессов – внедрение изолированной среды (также называется «песочницей»), в которой будет осуществляться запуск необходимых процессов. Разработка такого модуля для сайта является достаточно комплексной, но при этом актуальной задачей в контексте информационной безопасности.

1.2 Обзор исследований

А.С. Кравченко и О.Р. Горкина в своей работе рассмотрели изолирующую среду с теоретической точки зрения и описали принцип работы данной технологии [1].

Н.В. Филиппов и Н.В. Киреева провели анализ безопасного запуска кода в изолированной среде web-приложений [2].

А.В. Исайчева, В.С. Тиунов, А.М. Плюхин и Е.В. Монид в статье рассмотрели основные методы анализа, используемые в современных антивирусных решениях - статический, динамический и эвристический [3]. В статье также описаны недостатки каждого из методов и одним из результатов анализа является вывод о необходимости использования изолирующих сред.

М.К. Солонько и К.А. Краснов рассмотрели программную платформу виртуализации Docker [4]. Также в статье рассмотрен процесс создания и запуска контейнера на конкретном примере.

Н.В. Новикова рассмотрела в своей статье методы управления памятью ОС Linux без доступа [5]. В статье описаны семантический и несемантический доступы управления памятью.

1.3 Цель исследования

Цель – обеспечить безопасность запуска процессов и недоверенного кода на стороне сервера в веб-ориентированном приложении за счёт разработки и внедрения модуля создания изолирующих сред.

2. Материалы и методы

Для реализации поставленной цели используется язык программирования С.

3. Результаты и обсуждения

Перед описанием принципа работы изолирующей среды для запуска недоверенного кода необходимо рассмотреть взаимодействие языка программирования Си с процессами.

Начать стоит с контроля работы запущенных в данный момент процессов. В Си контроль основан на сигналах стандарта POSIX (набор стандартов, описывающих интерфейсы между ОС и прикладной программой). Сигнал это, исходя из официального описания в POSIX, уведомление процесса о каком-либо событии (завершение работы, создание нового процесса и др.). Всего существует 28 сигналов в Unix-подобных системах, но в рамках данной работы ниже приведены только наиболее используемые в разработке ПО:

1. SIGINT (2): Сигнал прерывания. Этот сигнал по умолчанию приводит к завершению процесса.
2. SIGTERM (15): Сигнал завершения, который часто используется для запроса гармоничного (то есть не аварийного) завершения процесса. Процесс может перехватить этот сигнал и выполнить необходимые действия перед завершением.
3. SIGKILL (9): Сигнал немедленного завершения. Процесс не имеет возможности перехватить или игнорировать этот сигнал, и он приводит к немедленному завершению процесса.
4. SIGUSR1 (10) и SIGUSR2 (12): Сигналы пользовательского использования. Эти сигналы не имеют фиксированного значения и могут использоваться процессами для своих нужд.
5. SIGHUP (1): Сигнал, который обычно посылается процессам при завершении сеанса терминала.
6. SIGCHLD (17): Сигнал, посылаемый родительскому процессу, когда один из его дочерних процессов завершается. Может использоваться для обработки завершения дочерних процессов.

За работу с сигналами отвечает библиотека “signal.h”. Данная библиотека содержит объявления и макросы для работы с сигналами. Сигнал может быть, как синхронным с помощью вызова “raise”, так и асинхронным. Стоит учитывать, что все описанные в рамках данной статьи сигналы предназначены для взаимодействия с процессами в Unix-подобных системах. То есть описанные далее примеры взаимодействия с процессами и реализации изоляции могут не подойти для реализации в Windows или Mac OS.

Самый простой пример обработки сигнала - это завершение работы процесса через комбинацию клавиш “Ctrl+C” (то есть вызов сигнала SIGINT). Обработка сигналов строится на работе функции “signal”, которая определяется следующим образом:

```
void (*signal(int signum, void (*handler)(int)))(int);
```

где signum – номер сигнала, а handler – обработчик события. Работает функция следующим образом: будет вызван указанный обработчик handler, когда сигнал с номером signum поступает.

Также немаловажным будет отметить функцию “raise” – функция посылает сигнал текущему процессу (то есть в котором она вызвана):

```
int raise(int signum);
```

Помимо библиотеки “signal.h” используются в рамках данной работы также следующие заголовочные файлы:

1. grp.h: Функции для работы с группами пользователей.
2. sched.h: работа с приоритетами процессов (управление планировщиком задач).
3. unistd.h: библиотека для работы с системными вызовами.
4. time.h: работа со временем (используется для замера времени выполнения процесса).
5. stdlib.h: включает стандартные функции для управления памятью, преобразованием строк и др.
6. resource.h: функции для работы с ресурсами процессора.
7. wait.h: содержит функции для ожидания завершения дочерних процессов и получения их статусов.

Список используемых библиотек неполный, представлены только наиболее важные заголовочные файлы Си в контексте рассматриваемой темы данной работы.

В рамках данной работы будут рассмотрены три типа ошибок, которые могут возникнуть во время выполнения того или иного процесса:

1. Утечка памяти.
2. Слишком долгое выполнение программы.
3. Ошибка бездействия.

Для получения информации об используемой памяти процессом можно использовать функцию “getrusage()”. Данная функция в качестве аргументов принимает константу, ссылающуюся на процесс (можно ссылаться как на исходный, так и на связанный с исходном процесс), и ссылку на объект “rusage”, который будет сохранять информацию о работе процесса.

“getrusage” предоставляет информацию не только об используемой процессом памяти, но также и много других данных о работе программы (например, количество операций блокового ввода и вывода, количество принятых сигналов и др.). Представлен данный объект в виде отдельной

структуры, которая сохранена в виде отдельного шаблона, то есть прописывать каждый параметр нет необходимости:

```
#include <sys/resource.h>
#include <stdio.h>
int main() {
    struct rusage usage;
    getrusage(RUSAGE_SELF, &usage);
    printf("Memory usage: %ld KB\n", usage.ru_maxrss);
    return 0;
}
```

Данный пример выводит информацию о максимальном потреблении памяти запущенного процесса (определяется через константу RUSAGE_SELF) в килобайтах (см. рис. 1).

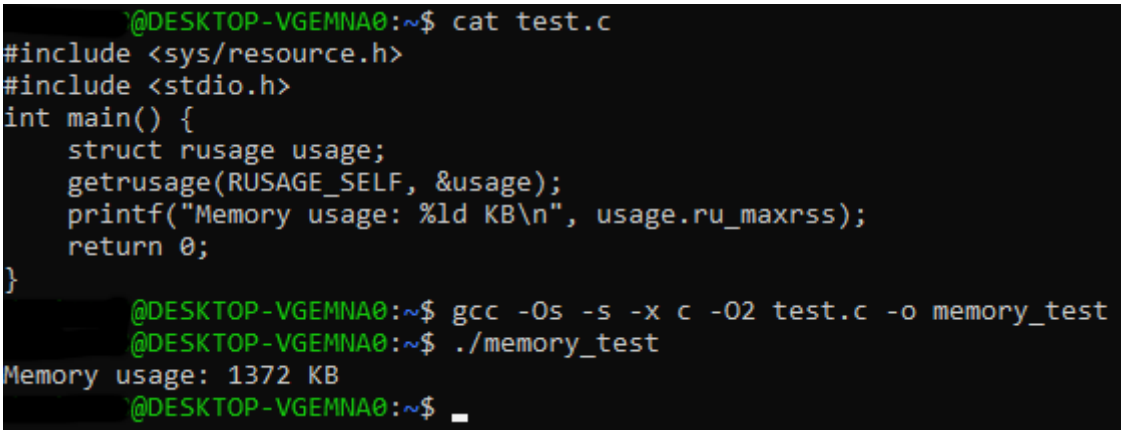

Рисунок 1. Пример запуска программы определения потребления памяти

Этот код необходимо доработать и установить ограничения по объёму используемой памяти для выполнения процесса.

В качестве примера будет рассматриваться процесс, потребляющий 3 Мб оперативной памяти компьютера (например, алгоритм сортировки подсчётом массива с количеством значений в 1000 элементов). Для ограничения потребляемой памяти будет использоваться rlimit – структура, содержащая все необходимые параметры для работы с памятью в процессах:

```
struct rlimit memory_limit;
memory_limit.rlim_cur = 1024 * 1024;
memory_limit.rlim_max = 1024 * 1024;
if (setrlimit(RLIMIT_AS, &memory_limit) != 0) {
  perror("setrlimit for memory");
  return 1;
}
```

Ограничение по памяти в данном случае достаточно сильное – можно использовать в рамках созданного процесса только 1 Мб оперативной памяти. Приведённая часть кода досрочно завершит выполнение программы, которая

будет запущена в рамках данного процесса. Процесс завершит свою работу, выведя сообщение о том, что дочерний (то есть алгоритм сортировки подсчётом) был завершён сигналом SIGSEGV (завершение с дампом памяти, то есть неправильное обращение к памяти устройства).

Но если изменить ограничение и поставить, например, 10 Мб, то работа процесса будет завершена успешно (см. рис. 2).

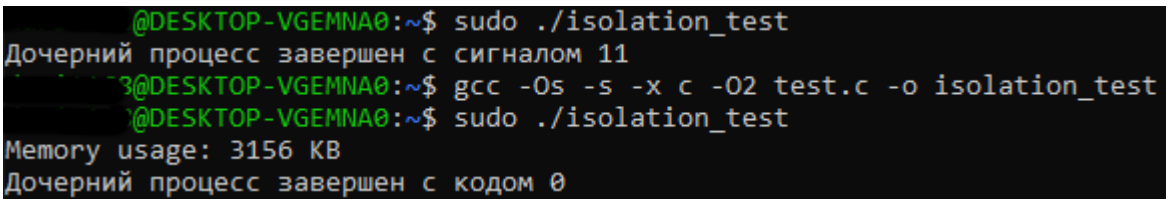

Рисунок 2. Пример работы ограничения оперативной памяти для запускаемого процесса

Вызов процесса осуществляется через execvp:

```
     char *script_argv[] = {"/bin/bash", "-c", "./memory_test",
NULL};
     execvp(script_argv[0], script_argv);
```

Проделанных операций недостаточно для установки ограничений на запускаемый процесс с недоверенным кодом. “execvp” создаёт новый процесс, на который не распространяются установленные ограничения. Для исправления этой ситуации была сделана привязка запуска процесса к пользователю “isouser”, который был специально создан для создания ограничений по памяти и времени выполнения процессов:

```
     if (setuid(getpwnam("isouser")->pw_uid) != 0) {
          perror("setuid");
          exit(EXIT_FAILURE);
     }
```

Этот код использует функцию getpwnam() для получения информации о пользователе по его имени, и затем использует setuid() для установки эффективного UID процесса. Важно отметить, что для использования setuid() требуются права суперпользователя.

Также был написан алгоритм определения новых запущенных процессов через пользователя “isouser”:

```
     FILE *cmd_output = popen("pgrep -u isouser", "r");
     if (cmd_output == NULL) {
          perror("popen");
          exit(EXIT_FAILURE);
     }
```

Далее остаётся перебрать полученные процессы и завершить превысившие хоть одно из ограничений. Если хоть один из процессов превысил ограничение, то также необходимо будет завершить все остальные процессы и передать сигнал на выход программы:

```
int pid;
while (fscanf(cmd_output, "%d", &pid) == 1) {
     if (kill(pid, SIGTERM) == -1) {
          perror("kill");
          exit(EXIT_FAILURE);
     }
}
```

Данный цикл перебирает полученные процессы, превышающие допустимые ограничения, и посылает сигнал завершения.

Аналогично можно сделать с ограничением по времени исполнения процесса:

```
struct rlimit time_limit;
time_limit.rlim_cur = 5;
time_limit.rlim_max = 5;
if (setrlimit(RLIMIT_CPU, &time_limit) != 0) {
     perror("setrlimit for time");
     exit(EXIT_FAILURE);
}
```

Таким образом получаем функцию “isolated_function”:

```
int isolated_function(void *arg) {
     struct rlimit time_limit;
     time_limit.rlim_cur = 5;
     time_limit.rlim_max = 5;
     if (setrlimit(RLIMIT_CPU, &time_limit) != 0) {
               perror("setrlimit for time");
               return 1;
     }
     struct rlimit memory_limit;
     memory_limit.rlim_cur = 1024 * 1024 * 10;
     memory_limit.rlim_max = 1024 * 1024 * 10;
     if (setrlimit(RLIMIT_AS, &memory_limit) != 0) {
               perror("setrlimit for memory");
               return 1;
     }
     char *script_argv[] = {"/bin/bash", "-c", "sudo python3
main.py", NULL};
     execvp(script_argv[0], script_argv);
     perror("execvp");
     return 1;
}
```

Это только одна из функций в изолирующей среде, которая отвечает за установку ограничений на выполнение исходного процесса и передачи значений функции в процесс, который создаётся через “execvp”. Также в изолирующей среде учитывается пользователь и процессы, запущенные через него.

Одним из этапов работы среды является обработка сигналов на выходе. Для этого реализован следующий этап программы:

```
int status;
waitpid(pid, &status, 0);
if (WIFEXITED(status)) {
     printf("%d\n", WEXITSTATUS(status));
} else if (WIFSIGNALED(status)) {
     printf("%d\n", WTERMSIG(status));
}
```

С помощью функции “waitpid” среда получает сигнал работы процесса. Данная функция является системным вызовом в C, который используется для ожидания завершения дочернего процесса. Этот вызов позволяет контролировать выполнение дочернего процесса и получать информацию о его завершении.

Прототип системного вызова выглядит следующим образом:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

В качестве примера работы изолирующей среды можно рассмотреть возможный случая запуска процесса, вызывающих ошибку утечки памяти. ТО есть это процесс, в котором есть критическая ошибка, приводящая к тому, что алгоритм занимает слишком много места в оперативной памяти.

Это, в контексте разработки и поддержки веб-ориентированных приложений, можно считать одной из самых опасных ошибок, которую можно допустить при работе со сторонними программами и процессами. Меньшая из проблем, к которой может привести данная ошибка, это зависание сервера. Худшее, что может произойти через несколько минут после допуска утечки памяти – временная неработоспособность сервера (что может быть критично в условиях поддержки веб-ориентированных систем, особенно тех, которые регулярно собирают и обрабатывают определённую информацию).

В рамках данной работы был написан код на Python, специально создающий утечку памяти во время запуска:

```
class ML:
    def __init__(self, data):
        self.data = data

def create_memory_leak():
    leaked_obj = []
    for i in range(10):
```

```
            leaked_obj.append(ML([1] * (10 ** 9)))
        global_obj.extend(leaked_obj)

if __name__ == "__main__":
    global_obj = []
    create_memory_leak()
```

Стоит определить место и причину утечки памяти в этом коде: изначально процесс создаёт 10 объектов класса ML (сокр. Memory Leak), которые добавляются в список “leaked_obj”. Каждый объект ML содержит атрибут data, представляющий собой список из 1 миллиарда элементов (каждый элемент - число 1). Затем список “leaked_obj” добавляется к глобальному списку “global_obj” с использованием метода extend.

Объекты ML в совокупности создадут утечку памяти, которая приведёт к ошибке “Memory Error”. Также стоит отметить работу сборщика мусора в Python в контексте данной программы: сборщик мусора должен очищать память после выполнения процесса в программе, но данный алгоритм приведёт к тому, что даже после завершения процесса память не будет очищена. Связано это с тем, что все созданные объекты были добавлены в глобальный список global_obj и они останутся в памяти, так как на них все еще существуют ссылки (см. рис. 3).

Самый простой способ исправить данную утечку памяти – очистить глобальный список после завершения работы процесса:

```
if __name__ == "__main__":
    global_objects = []
    create_memory_leak()

    global_objects.clear()
```

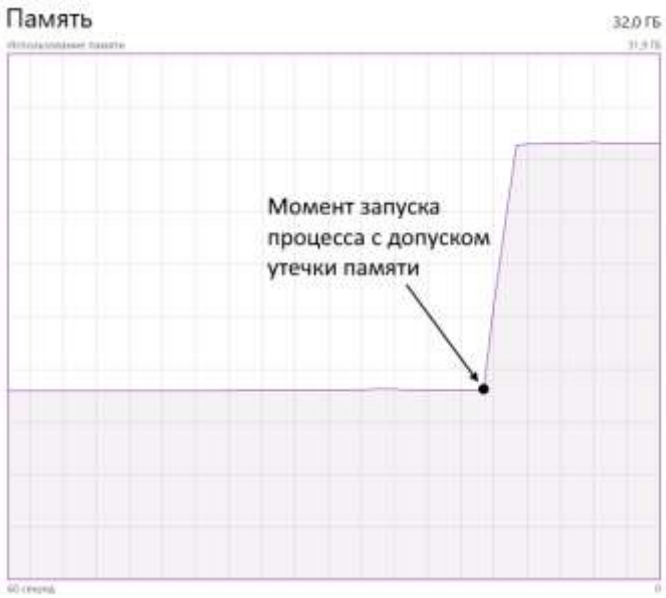

Рисунок 3. Результат запуска вышеописанного кода, отображение в диспетчере задач Windows (до запуска потребление ОЗУ – 11.5 Гб; после запуска – 26.6 Гб)

Чтобы убедиться в работоспособности скрипта, создающего утечку памяти, можно открыть диспетчер процессов и посмотреть работу программы (см. рис. 4).

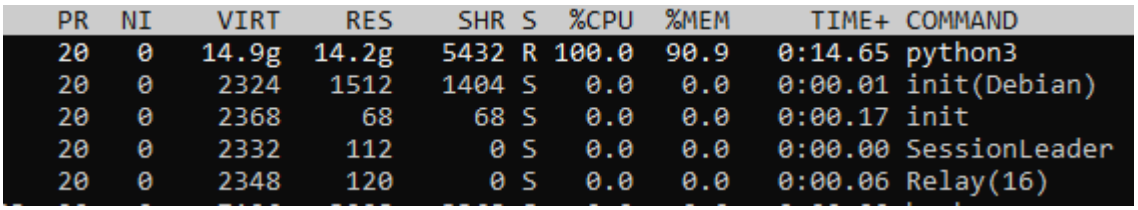

Рисунок 4. Результат запуска вышеописанного кода, отображение в top (Debian, первый процесс в списке)

Если запустить данный процесс в изолирующей среде, то он будет мгновенно прерван и в результате будет выведено сообщение о том, что процесс был завершён сигналом SIGTERM (см. рис. 5).

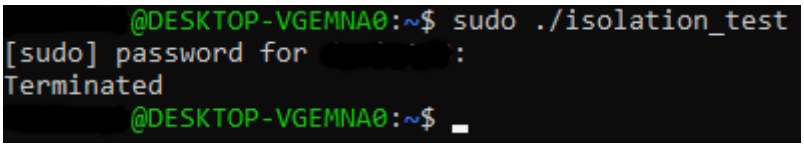

Рисунок 5. Результат работы изолирующей среды (вывод сигналов на момент тестирования и отладки работы не был задействован)

В реальных условиях работы Web-приложения допуск утечки памяти является критической проблемой, которая, как правило, приводит к нежелательным результатам (см. рис. 6).

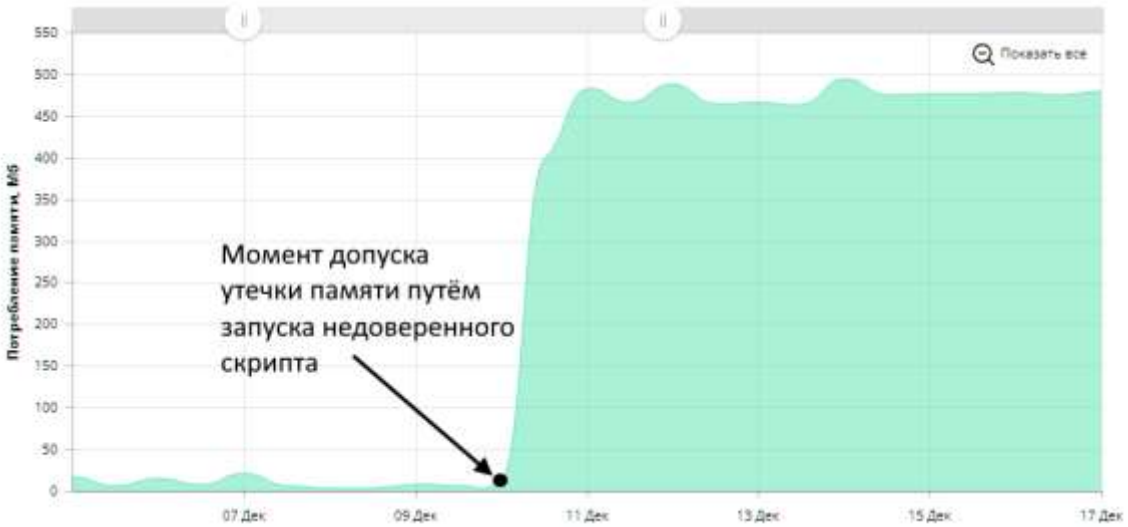

Рисунок 6. Пример запуска Python скрипта с утечкой памяти на сервере одного из хостингов

Для достижения большей защищённости сервера, стоит поместить изолирующую среду вместе с образами компилятора в Docker контейнер. Благодаря контейнеризации можно запустить необходимые процессы в

виртуальной среде, которую можно будет контролировать с помощью изолирующей среды. Ниже приведён пример подготовки контейнера, в котором находится изолирующая среда (рассмотрена только часть скрипта с изолирующей средой):

```
services:
  isolation:
    build:
      context: .
      dockerfile: Dockerfile
    command: ["/services/isolation/isolation_main"]
    volumes:
      - ./isolation:/services/isolation
```

Docker compose подразумевает наличие файла Dockerfile, в котором описаны команды для настройки изолирующей среды:

```
FROM debian:latest
RUN apt-get update && apt-get install cgroup-tools
COPY isolation /services/isolation
WORKDIR /services/isolation
CMD ["/services/isolation/isolation_main"]
```

В результате был получен Docker контейнер с изолирующей средой, в которой можно запускать недоверенные скрипты и процессы.

Выводы

В рамках данной статьи был описан процесс создания изолирующей среды в UNIX-подобной системе (использованный дистрибутив - Debian). Также рассмотрен сценарий, в котором применение изолирующей среды будет наиболее подходящим.

Библиографический список

1. Кравченко А.С., Горкина О.Р. О применении изолированной программной среды для обеспечения информационной безопасности // Техника и безопасность объектов уголовно-исполнительной системы: сборник материалов Международной научно-практической конференции, Воронеж. Иваново: ИПК "ПресСто"; Воронежский институт ФСИН России. 2022. Т.1. С. 240-242.
2. Филиппов Н.В., Киреева Н.В. Анализ возможности безопасного запуска кода в изолированной среде web-приложений // XXIX Российская научно-техническая конференция: Материалы XXIX Российской научно-технической конференции профессорско-преподавательского состава, научных сотрудников и аспирантов университета с приглашением ведущих ученых и специалистов родственных вузов и организаций, Самара. Самара: Поволжский государственный университет телекоммуникаций и

информатики. 2022. С. 51-52.
3. Исайчева А.В. и др. Методы анализа вредоносного кода // Концепции, инструменты и технологии развития современной науки и техники: Материалы XI Всероссийской научно-практической конференции, Ставрополь. Ставрополь: Общество с ограниченной ответственностью "Ставропольское издательство "Параграф". - 2023. С. 77-80.
4. Солонько М.К., Краснов К.А. Docker-контейнеры // Телекоммуникации и информационные технологии. 2020. Т.7. № 1. С. 65-69.
5. Новикова Н.В. Управление памятью операционной системы без доступа // Нанотехнологии: наука и производство. 2022. № 4. С. 6-9.