1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский политехнический университет Петра Великого
3. —
4. Институт прикладной математики и механики
5. **Высшая школа кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА 3,4**

«**Исследование принципов разработки средств защиты»**

по дисциплине «Теория и системы управления информационной безопасностью**»**

1. Выполнили

студенты гр. 3651003/50801 Д.И. Кожушок

А.А. Корольков

Ю.В. Романюк

* 1. Руководитель Е.В. Жуковский

1. Санкт-Петербург
2. 2019

**1. Цель работы**

Изучение принципов разработки средств антивирусной защиты. Изучение технологий и механизмов ОС Windows, используемых для реализации функций защиты. Получение навыков проектирования и разработки многокомпонентного программного обеспечения.

**2. Формулировка задания**

Лабораторная работа выполняется в группах по 3 человека. В ходе выполнения лабораторной работы необходимо спроектировать и разработать программное обеспечение (ПО), реализующее основные функции, выполняемые современным антивирусным программном обеспечением.

Разработать следующие подсистемы:

1. Графический интерфейс. Возможность ввода настроек всех подсистем и их отображение. Информирование об инцидентах ИБ. Логирование информации.
2. Подсистема фильтрации. Внедрение в запускаемые процессы DLL.
3. Подсистема статического анализа. Определение упаковки и попытка распаковать, проверка валидности подписи. Использование классификатора ВПО.
4. Подсистема защиты реального времени. Реализовать DLL, которая перехватывала вызовы опасных функций. И блокировала опасные операции (на основе MITRE ATT&CK), например, попытки внедрения кода в чужие процессы (WriteProcessMemory, CreateThread).
5. Подсистема сетевой защиты. Реализация возможности использования правил Snort/Suricata для блокировки сетевых пакетов.
6. Подсистема обновления и облачного сканирования. Построение классификатора ВПО на основе полученной трассы выполнения программы из песочницы.

**3. Результаты работы**

1. Графический интерфейс.

В ходе выполнения работы было реализовано несколько модулей. Модуль фонового логирования событий и модуль пользовательского интерфейса.

Модуль логирования собирает информацию от различных модулей и сохраняет в формате json, для удобного дальнейшего использования.

Пользовательский интерфейс позволяет совершать следующие функции:

* Первоначальная установка остальных модулей
* Запуск и остановка модулей динамической и статической проверки запускаемых файлов, а также файлового и сетевого фильтра.
* Просмотр Json-форматированных логов.
* Добавление и удаление элементов в файловый, реестровый фильтр
* Генерация и подключение новых сетевых правил фильтрации
* Вывод включенных правил сетевой фильтрации
* Статический анализ выбранного файла
* Отправка выбранного файла на обычный сервер проверки файлов
* Проверка наличия обновлений статического анализатора
* Запуск модулей реализованных в 5 лабораторной работе
* Отображение статуса всех модулей и запущенных процессов

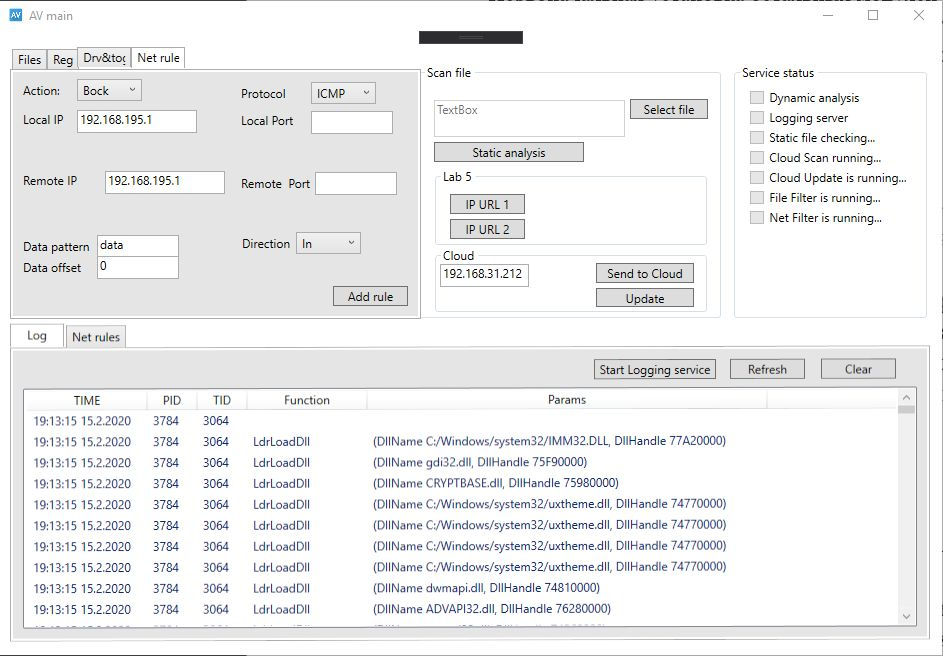


Рисунок 1-пользовательский интерфейс разработанной программы

1. Подсистема фильтрации и подсистема защиты реального времени

Для реализации внедрения в запускаемые процессы DLL c использованием механизма APC был использован и модифицирован проект InjDrv (<https://github.com/wbenny/injdrv>). Он состоит из драйвера, библиотеки реализующей фикции внедрения, внедряемой dll и управляющего приложения.

Для внедрения dll используется установка обработчика события создания процесса (PsSetCreateProcessNotifyRoutineEx). Внедрение Dll происходит на раннем этапе создания процесса, до загрузки kernel.32. Из-за этого невозможно отлавливать использования многих функций. Возможно перехватывать функции реализованные в ntdll или добавлять обработчик необходимой функции уже после момента загрузки нужной библиотеки. В ходе реализации модуля были использованы оба подхода.

В качестве последовательностей опасных функций была использована информация из базы атак MITRE (<https://attack.mitre.org/techniques/T1106/>). В случае нахождения искомой последовательности появляется окно, предупреждающее пользователя о выполнении потенциально опасной функции. Так же происходит логирование всей информации о вызовах функций для последующего вывода в интерфейс.

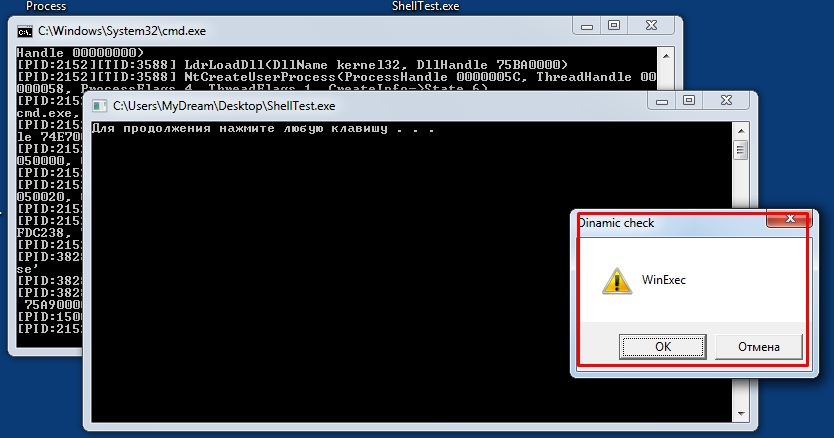


Рисунок 2- окно, предупреждающее пользователя

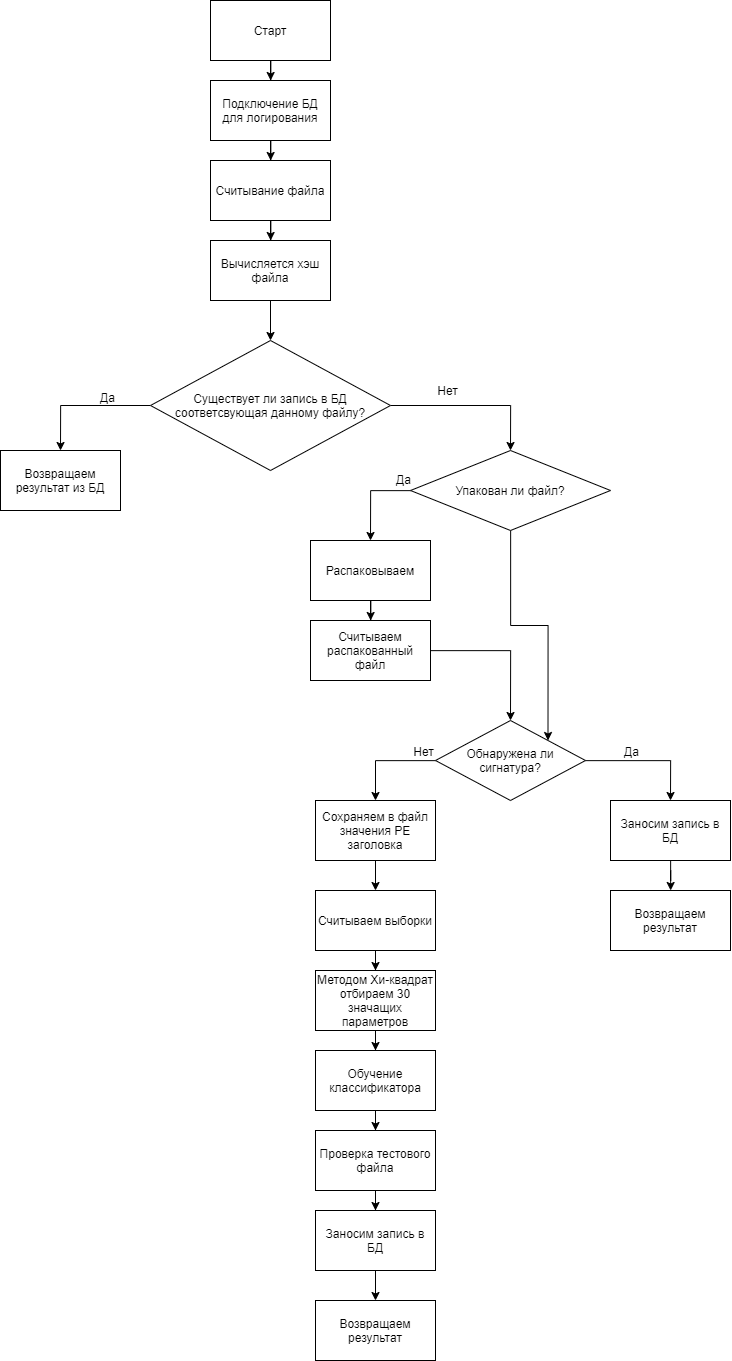
1. Подсистема статического анализа.

При запуске статического анализатора в качестве аргумента передается путь к проверяемому файлу.

На рисунке 3 представлена блок схема статического анализатора.

В качестве проверки на упаковку файла используется UPX-упаковщик.

Для обучения классификатора используется алгоритм RandomForest. Для отбора значащих параметров для обучения используется метод хи-квадрат.



*Рисунок 3 - Блок-схема статического анализатора*

1. Подсистема сетевой защиты.

Была реализована возможность использования правил, схожих со Snort/Suricata, для блокировки сетевых пакетов.

Синтаксис правила Snort:

<Действие> <Протокол> <IP-адреса отправителей> <Порты отправителей> <Оператор направления> <IP-адреса получателей> <Порты получателей> (ключ\_1 : значение\_1; ключ\_2 : значение\_2; ... ключ\_N : значение\_N;)

Был доработан сетевой драйвер фильтр, осуществляющий анализ пакетов. Для анализа используются следующие поля пакета: ip-адрес отправителя, порт отправителя, ip-адрес получателя, порт получателя, протокол, данные.

Правила записываются в ключ реестра. Для каждого правила создается следующая структура:

* действие: блокировка, логирование;
* направление: входящий, исходящий трафик;
* ip-адрес отправителя;
* маска отправителя;
* порт отправителя;
* ip-адрес получателя;
* маска получателя;
* порт получателя;
* полезная нагрузка.

Возможные действия при классификации пакетов: разрешить, блокировать. Callout – набор функций в драйвере, которые проводят инспекцию пакетов. Они имеют специальную функцию, выполняющую классификацию пакетов. Эта функция может принят следующее решение:

* разрешить (FWP\_ACTION\_PERMIT);
* блокировать (FWP\_ACTION\_BLOCK).

Также есть фильтры (filters) – правила, указывающие, в каких случаях вызывается тот или иной callout.

Регистрация callout:

NTSTATUS WfpRegisterCallout()

{

FWPS\_CALLOUT Callout = { 0 };

Callout.calloutKey = WFP\_SAMPLE\_ESTABLISHED\_CALLOUT\_V4\_GUID;

Callout.flags = 0;

Callout.classifyFn = FilterCallback;

Callout.notifyFn = NotifyCallback;

Callout.flowDeleteFn = FlowDeleteCallback;

return FwpsCalloutRegister(DeviceObject, &Callout, &RegCalloutId);

}

classifyFn – классифицирующая функция, notifyFn – функция добавления\удаления фильтра, flowDeleteFn – закрытие обрабатываемого потока.

После регистрации callout его необходимо присоединить к определенному уровню:

NTSTATUS WfpAddCallout() {

FWPM\_CALLOUT callout = { 0 };

callout.flags = 0;

callout.displayData.name = L"CalloutName";

callout.calloutKey = \*CalloutGuid;

callout.applicableLayer = FWPM\_LAYER\_INBOUND\_IPPACKET\_V4;

return FwpmCalloutAdd(EngineHandle, &callout, NULL, &AddCalloutId);

}

applicableLayer — уникальный идентификатор уровня, на который добавляется callout. "v4" в названии идентификатора означает версию протокола Ipv4.

После того, как callout успешно добавлен в систему, нужно создать фильтр, то есть указать, в каких случаях будет вызываться наш callout, а именно - его классифицирующая функция. Новый фильтр создается функцией WfpAddFIlter.

В FWPM\_FILTER есть структура FWPM\_FILTER\_CONDITION. Поле layerKey заполняется GUID’ом уровня (layer), к которому мы хотим присоединиться. В поле fieldKey нужно указать, что хотим контролировать — порт, адрес, приложение. Значение fieldKey определяет, значения какого типа будут в структуре FWP\_CONDITION\_VALUE, входящей в FWPM\_FILTER\_CONDITION. В данном случае в ней содержится ipv4-адрес. Поле matchType определяет, каким образом будет производиться сравнение значения в FWP\_CONDITION\_VALUE с тем, что пришло по сети.

NTSTATUS \_WfpAddFIlter(…){

FWPM\_FILTER filter = { 0 };

FWPM\_FILTER\_CONDITION condition[1] = { 0 };

UINT64 filterWeight = FWPM\_AUTO\_WEIGHT\_MAX;

FWP\_V4\_ADDR\_AND\_MASK AddrRangeMask = { 0 };

condition[0].matchType = FWP\_MATCH\_EQUAL;

condition[0].conditionValue.type = FWP\_V4\_ADDR\_MASK;

condition[0].conditionValue.v4AddrMask = &AddrRangeMask;

if (direction == INBOUND\_DIRECTION)

{

condition[0].fieldKey = FWPM\_CONDITION\_IP\_LOCAL\_ADDRESS;

filter.layerKey = FWPM\_LAYER\_INBOUND\_IPPACKET\_V4;

filter.action.calloutKey = WfpInboundCalloutGUID;

} else {

condition[0].fieldKey = FWPM\_CONDITION\_IP\_REMOTE\_ADDRESS;

filter.layerKey = FWPM\_LAYER\_OUTBOUND\_IPPACKET\_V4;

filter.action.calloutKey = WfpOutboundCalloutGUID;

}

filter.displayData.name = WFP\_FILTER\_NAME;

filter.subLayerKey = WfpSublayerGUID;

filter.weight.type = FWP\_UINT64;

filter.weight.uint64 = &filterWeight;

filter.numFilterConditions = 1;

filter.filterCondition = condition;

filter.action.type = FWP\_ACTION\_CALLOUT\_TERMINATING;

return FwpmFilterAdd(engineHandle, &filter, NULL, pFilterId);

}

Рассмотрим функцию, которая выполняет классификацию. Таких функций реализовано две, одна обрабатывает входящий трафик, другая исходящий. Функции принимают структуру FWPS\_INCOMING\_VALUES0\* Values, в которой содержится вся необходимая информация о пакете: IP-адрес отправителя и получателя, порты и протокол.

Каждый пакет отправляется на проверку и сравнивается на соответствие всем имеющимся правилам.

Процесс выполнения проверки:

1. (Ip-адрес отправителя из правила & маска из правила) сравниваются с (ip-адресом отправителя пакета& маска из правила), аналогично для адреса получателя.
2. Если ip-адреса совпадают, выполняется проверка на соответствие портов.
3. Проверка по полезной нагрузке.

Если какое-либо условие выполняется, то происходит блокировка пакета.

switch (cfg->action){

case IPV4\_BLOCK\_TRAFFIC:

decision = FWP\_ACTION\_BLOCK;

break;

Если выбрано действие протоколирование, то информация логируется:

switch (cfg->action){

case IPV4\_LOG\_TRAFFIC:{

switch (pOutboundIpv4Protocol->pHeader->proto){

case TCP\_PROTOCOL:{

DbgPrint("[tcp] OUT %u.%u.%u.%u <- %u.%u.%u.%u\n", pRemoteIp->S\_un.S\_un\_b.s\_b1, pRemoteIp->S\_un.S\_un\_b.s\_b2, pRemoteIp->S\_un.S\_un\_b.s\_b3, pRemoteIp->S\_un.S\_un\_b.s\_b4, pLocalIp->S\_un.S\_un\_b.s\_b1, pLocalIp->S\_un.S\_un\_b.s\_b2, pLocalIp->S\_un.S\_un\_b.s\_b3, pLocalIp->S\_un.S\_un\_b.s\_b4);

} break;

case UDP\_PROTOCOL:…

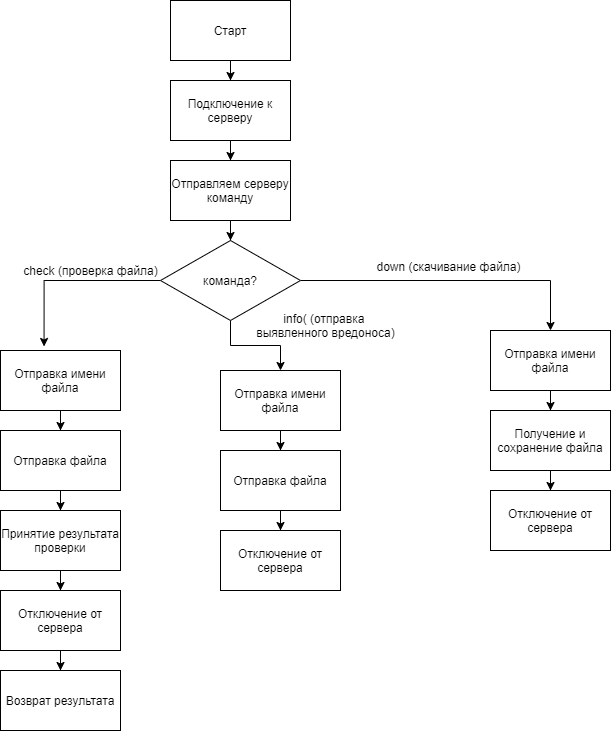
case ICMP\_PROTOCOL:…

decision = FWP\_ACTION\_PERMIT;

1. Подсистема обновления и облачного сканирования.

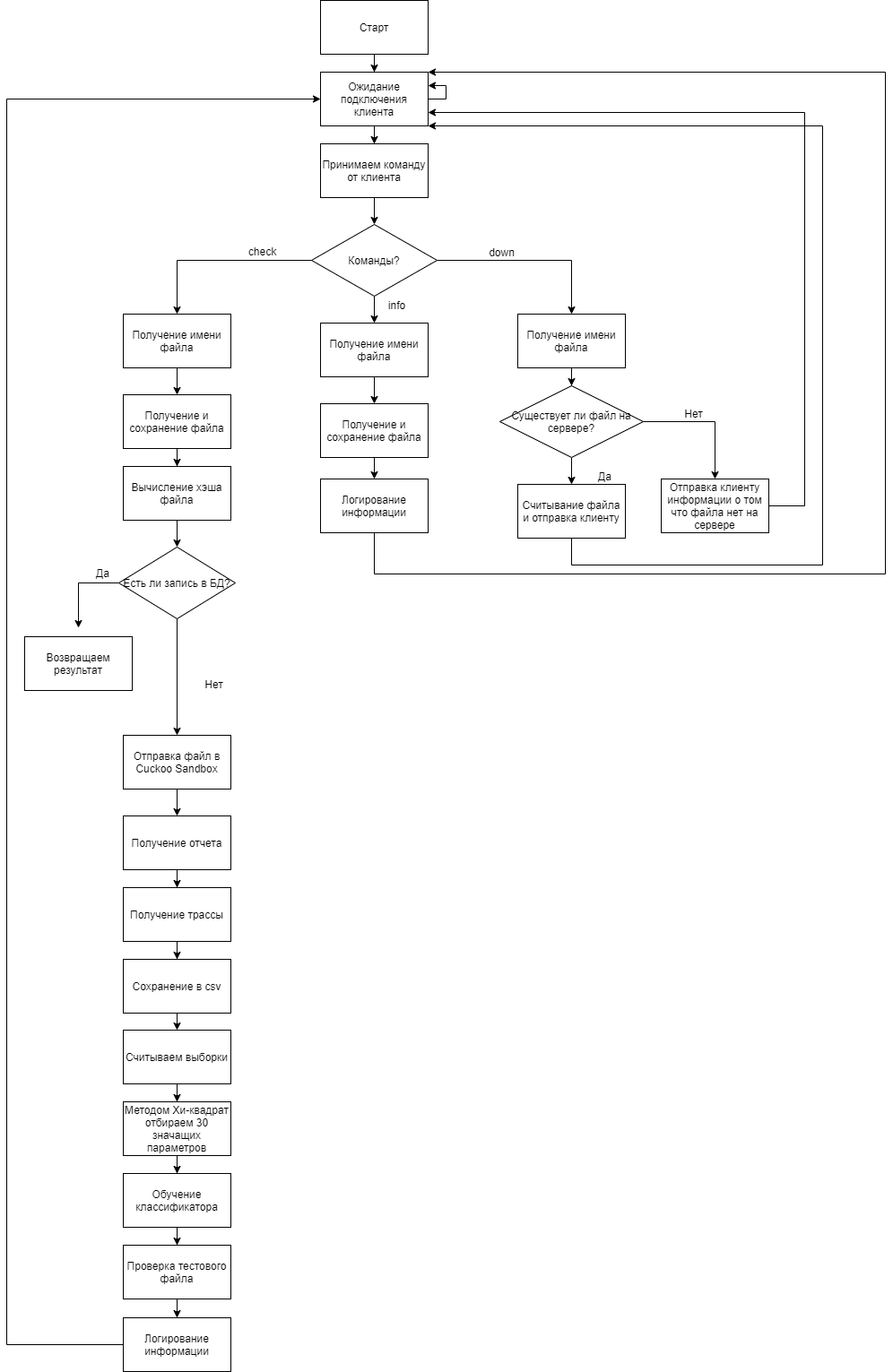
Было разработано клиент-серверное приложение для облачного сканирования.

Клиент обладает следующим функционалом: отправка файла на проверку, скачивание файла с сервера и отправка выявленного вредоносного файла на сервер. На рисунке 4 представлена блок схема работы клиента.



*Рисунок 4 - Блок-схема работы клиента облачного сканирования*

Сервер обладает следующим функционалом: логирование принимаемой информации от клиента, получение трассировки в Cuckoo Sandboox, анализ трассы в классификаторе, скачивание и отправка файлов. На рисунке 5 представлена блок-схема работы сервера.



*Рисунок 5 - Блок-схема работы сервера*

Также было разработано клиент-серверное приложение для обновлений. На стороне клиента и сервера хранятся БД со списком установленных модулей и их хэшей. Для обновлений клиент загружает с сервера актуальную БД, сравнивает со своей и если хэши отличаются, то загружает требуемые файлы с сервера.

**4. Выводы**

В ходе выполнения лабораторной работы были изучены принципы разработки средств антивирусной защиты. Были разработаны компоненты антивирусного приложения. В следующей лабораторной работе будет выполнено объединение компонент в единое антивирусное приложение.