ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и технологий Высшая школа киберфизических систем и управления

Отчет о прохождении учебной практики «Практика по получению первичных профессиональных умений и навыков, в том числе первичных умений и навыков научно-исследовательской деятельности»

Шкалин Кирилл Павлович
(Ф.И.О. обучающегося)

3 курс, группа № 3532704/90501
(номер курса обучения и учебной группы)

27.03.04 «Управление в технических системах»
(направление подготовки (код и наименование)

Место прохождения практики:

ВШ КФСУ

Сроки практики: с 15.06.2022 г. по 08.07.2022 г.

Руководитель практической подготовки от ФГАОУ ВО «СПбПУ»:

Сальников Вячеслав Юрьевич, Доцент – высшая школа киберфизических систем и управления
(Ф.И.О., уч. степень, должность)

Оценка:

Руководитель практической подготовки
от ФГАОУ ВО «СПбПУ»: Сальников В.Ю

Обучающийся: Шкалин К.П

Дата: 08.07.2022

Оглавление

Введение

Темой практической работы является защита программного обеспечения от отладчика. А именно применения антиотладочных техник для защиты приложения.

Защита программного обеспечения – это предотвращение возможности нелегального копирования, распространения и использования программного продукта. Об актуальности этой проблемы говорит внушительный процент пиратской (нелегальной) продукции, широко распространяющейся в сети Интернет. Из-за несовершенства законодательных систем многих стран привлечь к ответственности или найти распространителя нелицензионного продукта проблематично. Поэтому наилучшим решением будет применять средства программной защиты.

Наиболее распространенный способ защиты – серийный номер. Для каждой единицы программного продукта существует свой уникальный код активации. Тем не менее, такой метод можно обойти. Для этого взломщики используют программы отладчики.

Отладчик – программа, предназначенная для поиска ошибок в других программах. Отладчик позволяет выполнять пошаговую трассировку, отслеживать, устанавливать или изменять значения переменных в процессе выполнения кода, устанавливать и удалять контрольные точки или условия остановки. Все перечисленные возможности созданы для упрощения процесса разработки программного обеспечения, но также они активно применяются злоумышленниками для исследования чужих программ с целью обхода ограничений и защиты.

Соответственно антиотладочные приемы – участки кода, которые позволяют программе понять, что она выполняется под отладчиком, а следовательно, что за ней пристально наблюдают. На основе этой информации программа решает прервать ли ей свое выполнение (что не очень эффективно) или пустить ход выполнения по альтернативному пути, чтобы запутать взломщика.

Как итог, целью практической работы является разработка таких техник, которые дадут программе понять, выполняется она под отладчиком или нет.

Для достижения поставленной цели в ходе практической работы были проделаны следующие этапы:

- Создание демонстрационной программы, на которой будут тестироваться антиотладочные приемы.
- Разработка антиотладочных приемов, основанных на замере временных интервалов между выполнением инструкций.
- Разработка антиотладочных приемов, основанных на анализе информации о процессах в системе.
- Внедрение разработанных техник в программу.
- Анализ и сравнение эффективности применяемых методов.

Теоретическое описание антиотладочных техник

Замер времени выполнения

Методы, использующие замер времени выполнения инструкций программы, хорошо справляются с «отловом» пошаговой отладки программы. При отладке программы взломщик, обычно, ставит точки останова, а затем пошагово выполняет следующие инструкции, просматривая содержимое регистров и стека. Из-за такого пошагового выполнения программы время между инструкциями увеличивается в несколько раз. Так, например, при нормальной работе процессор выполняет участок кода программы за 5–10 миллисекунд (это, конечно, зависит от размера участка кода). Так вот, если программа обнаружит, что этот же кусок кода процессор выполняет за несколько секунд или даже минут, то тут можно с уверенностью сказать, что программа выполняется под отладчиком.

Следовательно, в различных участках кода, а особенно там, где происходит нажатие на кнопку или проверка ключа, будем засекать временной интервал. Если интервал будет слишком велик, то будем, делать вывод, что программа работает под отладчиком.

Есть несколько различных способов замерить время выполнения участка программы. Рассмотрим их и выберем наиболее удобный и точный вариант.

1. Использовать ассемблерную инструкцию rdtsc. Эта инструкция считывает текущее значение счетчика меток времени процессора (64-битный MSR) в регистры EDX:EAX. Регистр EDX загружается старшими 32 битами MSR, а регистр EAX загружается младшими 32 битами. В коде такую проверку можно представить следующим образом (рисунок 1)

```
bool is_debugger_present()
{
    int elapsed = 1024;
    __asm
    {
        rdtsc
        xchg    esi, eax
        mov     edi, edx
        rdtsc
        sub     eax, esi
        sbb     edx, edi
        jne     rdtsc_being_debugged
        cmp     eax, elapsed
        jnbe    rdtsc_being_debugged
    }
    return false;
rdtsc_being_debugged:
    return true;
}
```

Рисунок 1 – использование rdtsc

2. API операционной системы Windows содержит несколько функций работы со временем. Так, например, можно использовать функции GetSystemTime и GetLocalTime. Первая функция получает текущую системную дату и время в формате всемирного кодированного времени (UTC). Вторая функция получает текущую локальную дату и время. После чего можно воспользоваться функцией SystemTimeToFileTime, которая представит дату и время в виде структуры с двумя 4-байтовыми числами. Которые потом можно легко перевести в одно 8-байтовое число, благодаря структуре ULARGE_INTEGER. Соответственно временной интервал будет представлять из себя разницу между двумя числами.

```
unsigned __int64 FiletimeToUint64(const FILETIME& fileTime)
{
    ULARGE_INTEGER uiTime;
    uiTime.LowPart = fileTime.dwLowDateTime;
    uiTime.HighPart = fileTime.dwHighDateTime;
    return uiTime.QuadPart;
}

bool is_debugger_present_GetTime()
{
    int elapsed = 1024;

    SYSTEMTIME firstCheckTime;
    GetSystemTime(&firstCheckTime); //GetLocalTime

    //YOUR CODE HERE

    SYSTEMTIME secondCheckTime;
    GetSystemTime(&secondCheckTime); //GetLocalTime

    FILETIME ftConstructorCallTime, ftCheckCallTime;
    if (!SystemTimeToFileTime(&firstCheckTime, &ftConstructorCallTime))
        return false;
    if (!SystemTimeToFileTime(&secondCheckTime, &ftCheckCallTime))
        return false;

    return (FiletimeToUint64(ftCheckCallTime) - FiletimeToUint64(ftConstructorCallTime)) > elapsed;
}
```

Рисунок 2 – применение GetSystemTime и GetLocalTime

3. Также API Windows поддерживает функции счетчиков, которые возвращают количество миллисекунд (микросекунд) с момента запуска системы. Это соответственно функции GetTickCount и QueryPerformanceConter.

```
bool is_debugger_present_GetCount(int elapsed)
{
    DWORD first = GetTickCount();

    //YOUR CODE HERE

    return (GetTickCount() - first) > elapsed;
}
```

Рисунок 3 – применение GetTickCount

```
double PCFreq = 0.0;
__int64 CounterStart = 0;

void StartCounter()
{
    LARGE_INTEGER li;
    if (!QueryPerformanceFrequency(&li))
        std::cout << "QueryPerformanceFrequency failed!\n";

    PCFreq = double(li.QuadPart) / 1000.0;

    QueryPerformanceCounter(&li);
    CounterStart = li.QuadPart;
}
double GetCounter()
{
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return double(li.QuadPart - CounterStart) / PCFreq;
}
```

Рисунок 4 – применение QueryPerformanceCounter

В данной работе я остановлюсь на таком наборе способов замера времени. На мой взгляд, удобнее всего использовать функции счетчики. А конкретно GetTickCount, так как точности до миллисекунд нам более, чем достаточно, ведь при трассировке время выполнения инструкций будет больше секунды.

Получение информации о процессах в системе

В следующей технике мы просто попытаемся найти процесс отладчика в системе. Для этого мы воспользуемся функцией FindWindow, которая. Эта функция возвращает дескриптор окна верхнего уровня, которое соответствует переданному имени класса окна.

```
BOOL findWindow_check()
{
    const int n = 6;
    const char* window_classes[n] = {
    "OLLYDBG",
    "WinDbgFrameClass",
    "ID",
    "Zeta Debugger",
    "Rock Debugger",
    "ObsidianGUI",
    };

    for (int i = 0; i < n; ++i)
    {
        if (FindWindowA(window_classes[i], NULL) != NULL)
            return true;
    }
    return false;
}
```

Рисунок 5 – применение FIndWindow

Другой способ – узнать родительский процесс, который запустил наше приложение. Для этого необходимо воспользоваться функцией NtQueryInformationProcess. Для работы с этой функцией к ней необходимо динамически линковаться. Для это нужно вызвать функцию LoadLibraryA(“ntdll.dll”), которая загружает указанный модуль в адресное пространство вызывающего процесса. После чего нужно вызвать функцию GetProcAddress, которая возвращает адрес экспортированной функции из указанной библиотеки динамической компоновки (DLL). В результате мы можем посмотреть, откуда было запущено наше приложение.

```
typedef NTSTATUS(WINAPI* TNtQueryInformationProcess)(
    IN HANDLE                   ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID                   ProcessInformation,
    IN ULONG                    ProcessInformationLength,
    OUT PULONG                  ReturnLength
    );

struct MY_PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    ULONG_PTR UniqueProcessId;
    ULONG_PTR InheritedFromUniqueProcessId;
};
```

Рисунок 6.1 – применение NtQueryInformationProcess

```
BOOL CheckParentProcess()
{
    HMODULE hNtdll = LoadLibraryA("ntdll.dll");

    TNtQueryInformationProcess pfnNtQueryInformationProcess = (TNtQueryInformationProcess)GetProcAddress(hNtdll, "NtQueryInformationProcess");

    HWND hExplorerWnd = GetShellWindow();
    if (!hExplorerWnd)
        return false;

    DWORD dwExplorerProcessId;
    GetWindowThreadProcessId(hExplorerWnd, &dwExplorerProcessId);

    MY_PROCESS_BASIC_INFORMATION ProcessInfo;
    NTSTATUS status = pfnNtQueryInformationProcess(
        GetCurrentProcess(),
        PROCESSINFOCLASS::ProcessBasicInformation,
        &ProcessInfo,
        sizeof(ProcessInfo),
        NULL
    );

    if (!NT_SUCCESS(status))
        return false;

    return (DWORD)ProcessInfo.InheritedFromUniqueProcessId != dwExplorerProcessId;
}
```

Рисунок 6.2 – применение NtQueryInformationProcess

Оба вышепредставленные техники позволяют понять, что программа работает под отладчиком. Я буду использовать их одновременно.

Практическое применение

Демонстрационная программа

Для выполнения задания требуется программа, на которой будут тестироваться антиотладочные техники. В связи с этим была написана простая программа с минимальным интерфейсом (рисунок 7). Программа написана на WinAPI без использования сторонних пакетов.

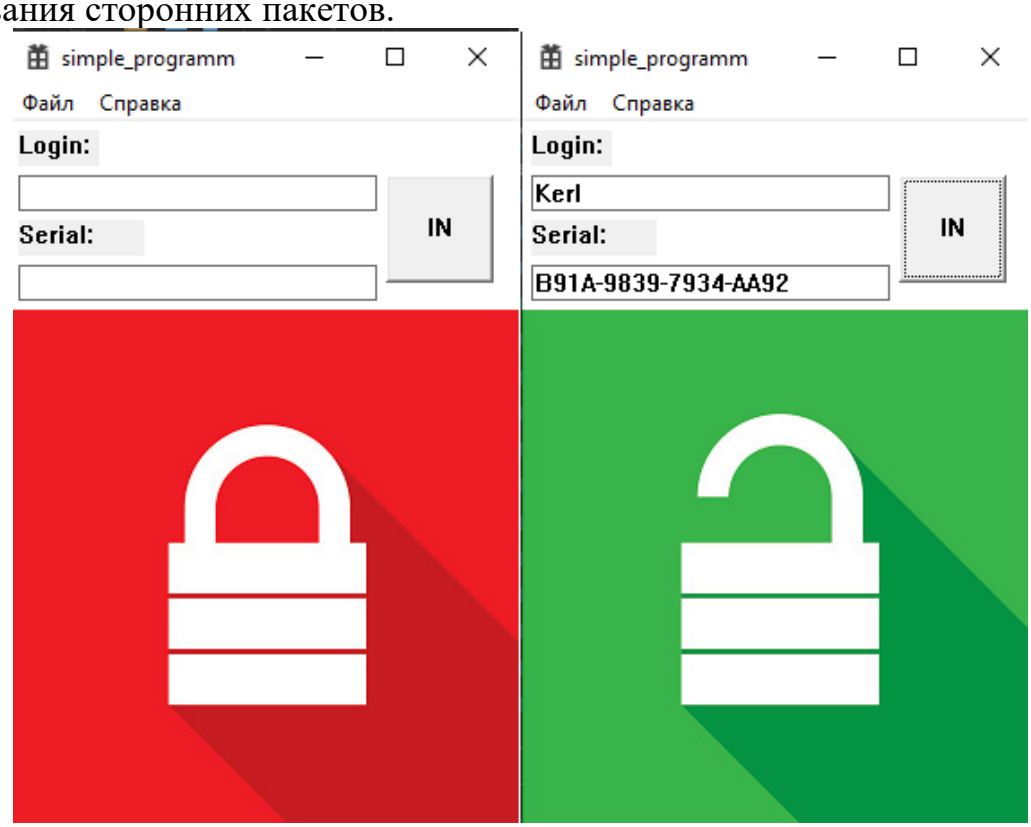


Рисунок 7 – интерфейс программы.

Программа содержит два поля edit: один для логина, другой для серийного ключа. Серийный ключ составляется из имени пользователя в процессе работы программы (Рисунок 8).

```
//For User "Kerl" serial "B91A-9839-7934-AA92"
void generateSerial(const char* userName, char serial[20])
{
    long long sn = 0;
    for (int i = 0; i < strlen(userName); ++i)
    {
        sn += userName[i];
        sn *= 123456;
    }
    sn ^= MASK;
    char hexSn[17];
    sprintf_s(hexSn, "%llX", sn);
    for (int i = 0, j = 0; i < 16; ++i, ++j)
    {
        if (i % 4 == 0 && i > 0)
        {
            serial[j] = '-';
            ++j;
        }
        serial[j] = hexSn[i];
    }
    serial[19] = 0;
}
```

Рисунок 8 – алгоритм получения ключа из логина

В случае, если введенный пользователем ключ не совпадает с тем, которые генерирует программа, то выводится MessageBox, уведомляющий пользователя о неудаче (Рисунок 9).

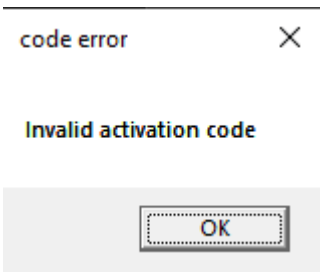


Рисунок 9 – MessageBox

Когда пользователь, ввел логин и ключ, он нажимает на кнопку IN, чтобы программа одобрила ему вход. После нажатия на кнопку IN в функцию обработчик событий WndProc попадает сообщение WM_COMMAND (0x0111). После этого, в зависимости от параметра сообщения, выполняется какой-либо участок кода. Если параметр соответствует ID кнопки, то выполняется проверка ключа (Рисунок 10). (Понимание этого нам понадобится позже, при работе в отладчике)

```
case WM_COMMAND:
    {
        int wmId = LOWORD(wParam);
        // Разобрать выбор в меню:
        switch (wmId)
        {
        case IDM_ABOUT:
            DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
            break;
        case IDM_EXIT:
            DestroyWindow(hWnd);
            break;
        case MCM_IN:
        {
            char userName[65];
            GetWindowTextA(hEdLogin, userName, 64);

            char correctSerial[20];
            generateSerial(userName, correctSerial);

            char enteredSerial[20];
            GetWindowTextA(hEdPassword, enteredSerial, 20);
            //enteredSerial[19] = 0;

            if (strcmp(enteredSerial, correctSerial) == 0)
            {
                ShowWindow(hStUnlock, SW_SHOW);
                ShowWindow(hStLock, SW_HIDE);
            }
            else
            {
                ShowWindow(hStLock, SW_SHOW);
                ShowWindow(hStUnlock, SW_HIDE);
                MessageBoxA(hWnd, "Invalid activation code", "code error", MB_OK);
            }
        }
        break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
        }
```

Рисунок 10 – код с проверкой ключа

Обход защиты без применения антиотладочных техник

Попробуем обойти проверку ключа в программе, которая не содержит антиотладочных приемов. Для этого воспользуемся отладчиком OllyDbg.

Первым делом скомпилируем программу без отладочной информации. После чего загрузим ее в отладчик OllyDbg и начнем выполнение нажатием F9. На экране мы увидим следующую картину (рисунок 11).

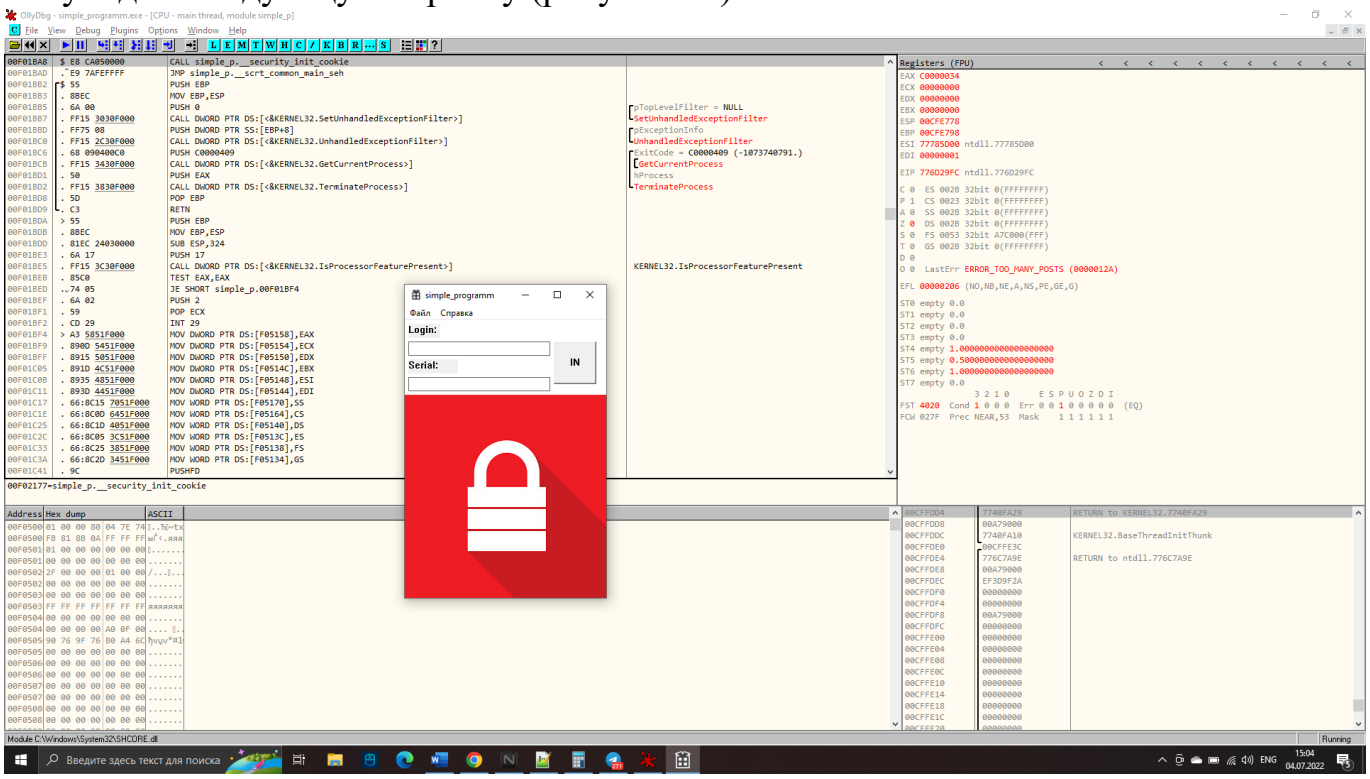
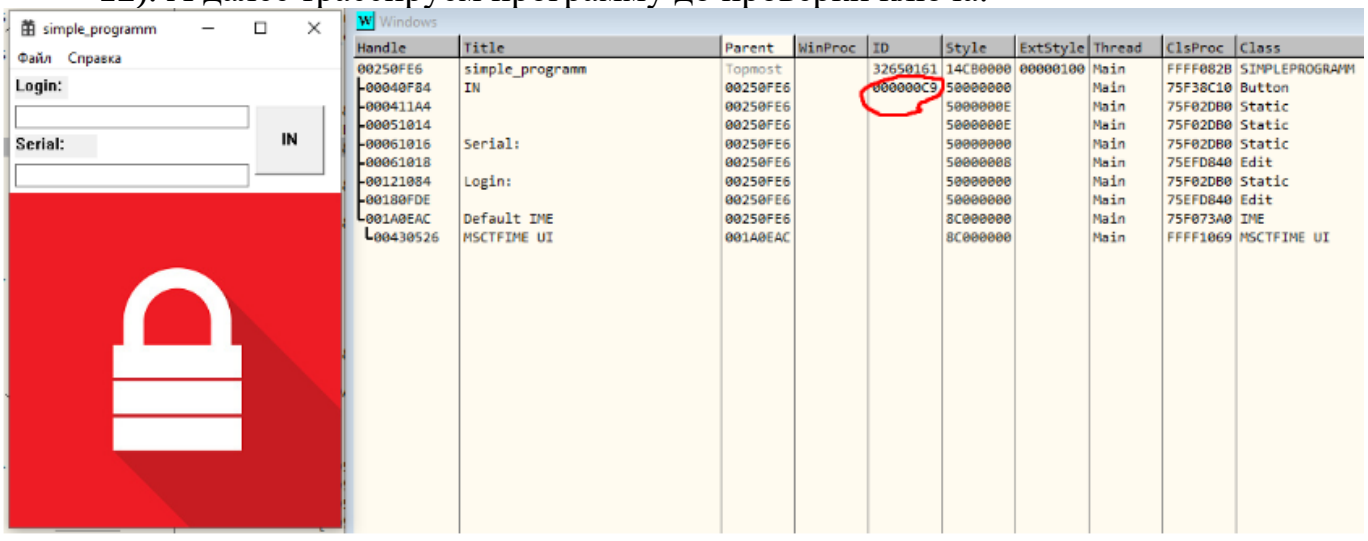


Рисунок 11 – запуск программы в отладчике

На следующем этапе необходимо расставить точки останова там, где будет производиться проверка ключа. Для этого есть несколько способов:

1. Найти вызов системной функции GetMessage и поставить точку останова с условием, что сообщение будет равно WM_COMMAND. Дальше смотрим значение wParam, там оно должно быть равно значению ID кнопки IN. Соответственно ID можно посмотреть в OllyDbg во вкладке windows (рисунок 12). А далее трассируем программу до проверки ключа.



Handle	Title	Parent	WinProc	ID	Style	ExtStyle	Thread	ClsProc	Class
00250FE6	simple_programm	Topmost		32650161	14CB0000	00000100	Main	FFFF082B	SIMPLEPROGRAMM
00040F84	IN	00250FE6		000000C9	50000000		Main	75F38C10	Button
000411A4		00250FE6			5000000E		Main	75F02DB0	Static
00051014		00250FE6			5000000E		Main	75F02DB0	Static
00061016	Serial:	00250FE6			50000000		Main	75F02DB0	Static
00061018		00250FE6			50000008		Main	75EFD840	Edit
00121084	Login:	00250FE6			50000000		Main	75F02DB0	Static
00180FDE		00250FE6			50000000		Main	75EFD840	Edit
001A0EAC	Default IME	00250FE6			8C000000		Main	75F073A0	IME
00430526	MSCTFIME UI	001A0EAC			8C000000		Main	FFFF1069	MSCTFIME UI

Рисунок 12 – вкладка windows отладчика OllyDbg

2. Второй способ основан на том, что в случае ввода неправильного ключа, программа выводит MessageBox. Поэтому мы поставим точку останова на вызов системной функции MessageBox и просмотрим код над ней. Ближайшая операция условного перехода, скорее всего будет тем, что мы хотим найти.

В итоге мы должны получить следующее (рисунок 13). По адресу 00F013F2 находится операция сравнения. В окне регистров можно увидеть, что регистр EAX содержит значение, введенное пользователем, а ECX корректное значение пароля. То есть мы уже имеем правильный пароль. Но, если мы хотим, чтобы программа одобряла вход для любого пароля, то можно в следующей строке заменить инструкцию JNZ на JMP, и тогда вход будет происходить в независимости от пароля.

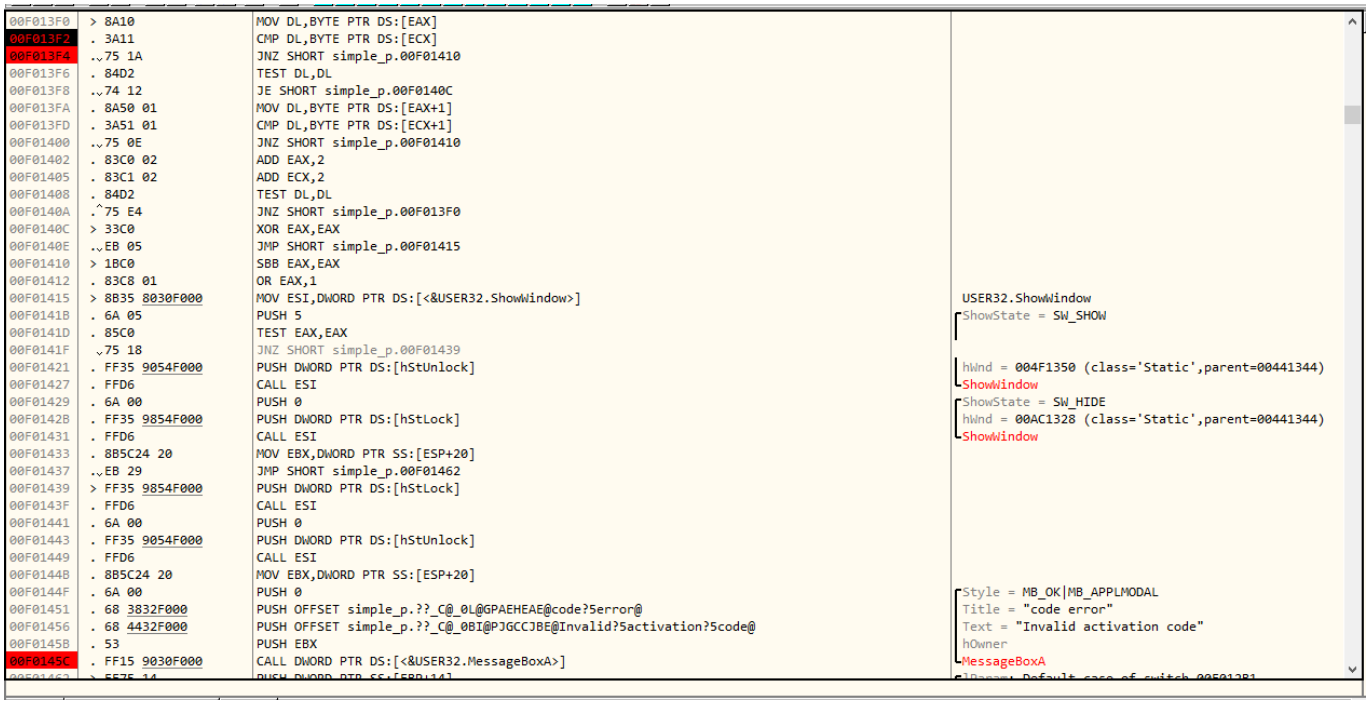


Рисунок 13 – программа, остановленная на моменте проверки ключа

Из примера видно, что обойти защиту этой программы очень просто. А теперь внедрим в программу антиотладочные техники и посмотрим, как это помешает взломщику.

Внедрение антиотладочных техник

Добавим проверку времени выполнения инструкций в код проверки ключа. Для этого воспользуемся функцией GetTickCount. В нескольких местах будем выполнять замер времени выполнения тех или иных функций и, если время будет превышать секунду, будем выводить MessageBox с соответствующим сообщением. Хочу отметит, что вывод диалогового окна с сообщением о том, что отладчик обнаружен, не имеет никакого смысла с точки зрения защиты приложения. Мы будем так поступать исключительно для наглядности работы антиотладочного приема.

Код программы изменится следующим образом (рисунок 14).

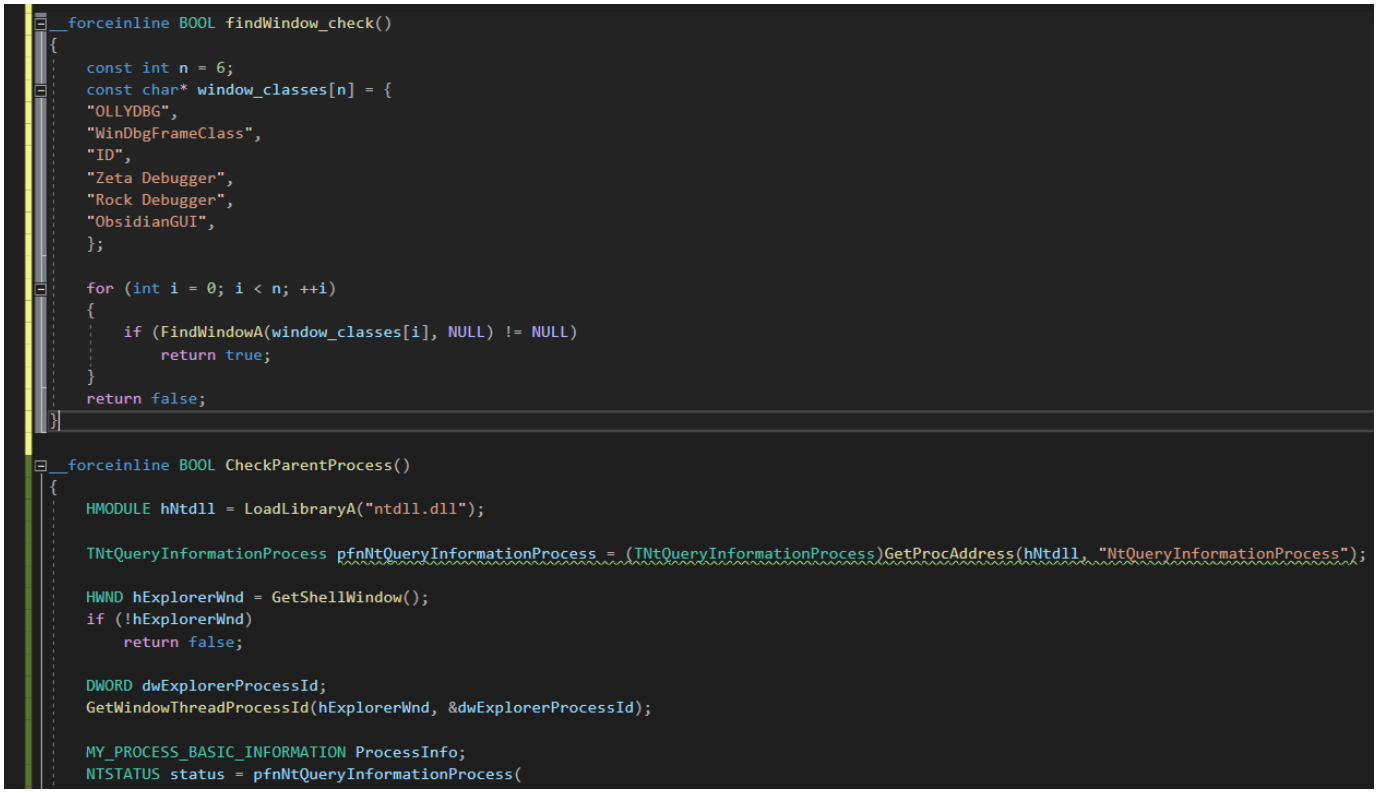
```
case MCM_IN:
{
    DWORD first = GetTickCount();
    char userName[65];
    GetWindowTextA(hEdLogin, userName, 64);

    if (first - GetTickCount() > 1000)
    {
        MessageBoxA(hWnd, "Debugger detected", "find debugger", MB_OK);
        return 0;
    }

    first = GetTickCount();

    char correctSerial[20];
    generateSerial(userName, correctSerial);

    char enteredSerial[20];
    GetWindowTextA(hEdPassword, enteredSerial, 20);
    //enteredSerial[19] = 0;
    if (first - GetTickCount() > 1000)
    {
        MessageBoxA(hWnd, "Debugger detected", "find debugger", MB_OK);
        return 0;
    }

    first = GetTickCount();
    if (strcmp(enteredSerial, correctSerial) == 0)
    {
        if (first - GetTickCount() > 1000)
        {
            MessageBoxA(hWnd, "Debugger detected", "find debugger", MB_OK);
            return 0;
        }
```

Рисунок 14 – код демонстрационной программы, содержащий антиотладочный прием

Попробуем теперь воспользоваться алгоритмом обхода защиты из предыдущего пункта. Теперь при трассировке программы с целью найти участок кода с проверкой условия, мы попадаем в условие, которое выводит MessageBox о том, что отладчик обнаружен и завершает попытку входа (рисунок 15). Также хочу сказать, что добавление проверок времени несколько засорило ассемблерный код, отчего читать и ориентироваться в нем стало сложнее.

Рисунок 15 – программа с сообщением об обнаруженном отладчике

Добавим теперь другую группу антиотладочных приемов. Следующие две техники вынесены в отдельные функции, что упростит взломщику их обнаружение и устранение. Поэтому важно пометить эти функции __forceinline, что сделает их встроенными (рисунок 16). При вызове встроенной функции компилятор будет вставлять код функции вместо стандартного вызова. Кроме того, что это усложнить процесс обнаружения антиотладочного приема, это усложнит взломщику понимание ассемблерного кода.



```
__forceinline BOOL findWindow_check()
{
    const int n = 6;
    const char* window_classes[n] = {
    "OLLYDBG",
    "WinDbgFrameClass",
    "ID",
    "Zeta Debugger",
    "Rock Debugger",
    "ObsidianGUI",
    };

    for (int i = 0; i < n; ++i)
    {
        if (FindWindowA(window_classes[i], NULL) != NULL)
            return true;
    }
    return false;
}

__forceinline BOOL CheckParentProcess()
{
    HMODULE hNtdll = LoadLibraryA("ntdll.dll");

    TNtQueryInformationProcess pfnNtQueryInformationProcess = (TNtQueryInformationProcess)GetProcAddress(hNtdll, "NtQueryInformationProcess");

    HWND hExplorerWnd = GetShellWindow();
    if (!hExplorerWnd)
        return false;

    DWORD dwExplorerProcessId;
    GetWindowThreadProcessId(hExplorerWnd, &dwExplorerProcessId);

    MY_PROCESS_BASIC_INFORMATION ProcessInfo;
    NTSTATUS status = pfnNtQueryInformationProcess(
```

Рисунок 16 – встроенные функции для обнаружения отладчика

```
int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                      _In_opt_ HINSTANCE hPrevInstance,
                      _In_ LPWSTR    lpCmdLine,
                      _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    // TODO: Разместите код здесь.

    // Инициализация глобальных строк
    LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadStringW(hInstance, IDC_SIMPLEPROGRAMM, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Выполнить инициализацию приложения:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    HACCEL hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_SIMPLEPROGRAMM));

    MSG msg;

    if (findWindow_check()) {
        MessageBoxA(NULL, "Debugger detected", "find debugger", MB_OK);
        return 0;
    }
```

Рисунок 17 – пример применения функции обнаружения отладчика

В результате применения этих техник программе удается обнаружить отладчик еще только при старте (Рисунок 18).

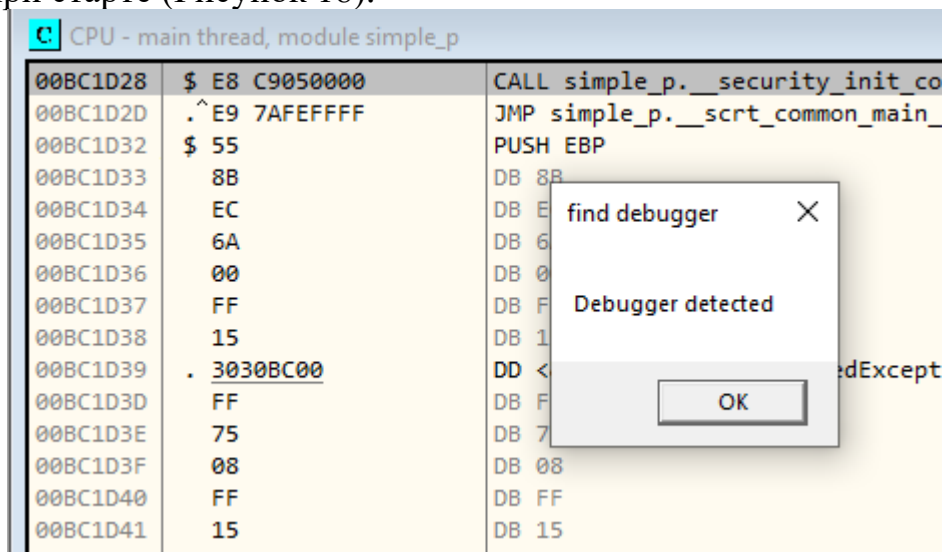


Рисунок 18 – обнаружение отладчика при запуске программы

Все использованные техники смогли обнаружить отладчик.

Вывод

В результате выполнения практической работы было реализовано и применено несколько антиотладочных приемов, основывающихся на замере времени выполнения инструкций. Кроме того, был разработан и применен антиотладочный прием, который ищет процесс отладчика в системе. А также прием, который проверяет свой родительский процесс и делает из этого вывод о работе под отладчиком. Все разработанные техники тестировались на специально написанной демонстрационной программе и отладчике OllyDbg.