Актуальные вопросы безопасности современных информационных технологий: Монография / [Р.Р. Бежан и др.]; Под ред. д-ра экон. наук, проф. Е.В. Стельмашонок. – СПб : Изд-во СПбГЭУ, 2015. – 164 с. – ISBN 978-5-7310-3418-0, 500 экз.

Федоров Д.Ю. аспирант СПбГЭУ

ИСПОЛЬЗОВАНИЕ ИНТЕРАКТИВНОГО ДИЗАССЕМБЛЕРА IDA PRO ДЛЯ ОБНАРУЖЕНИЯ МОДИФИКАЦИИ СЕГМЕНТА КОДА ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ В ОПЕРАЦИОННОЙ СИСТЕМЕ WINDOWS

Модифицируемый код с одной стороны противоречит канонам программирования, по которым код – это код, и его следует исполнять, а данные – это данные, и их следует читать, а также при желании модифицировать. Но с другой стороны есть принцип фон Неймана, при грубой трактовке которого нет принципиальной разницы между данными и кодом – все это лишь последовательность байтов и битов.

Код, изменяющий свои собственные инструкции во время выполнения, будем называть самомодифицирующимся.

В операционной системе (ОС) MS-DOS модификация кода во время исполнения – совсем простое дело. Там можно менять содержимое ячейки вне зависимости от содержания в ней данных или кода. В ОС Windows код напрямую модифицировать запрещено и, вроде бы, все пути к модификации собственного кода закрыты. Однако выход есть и не один.

Наиболее популярным (но не самым простым) методом модификации кода во время выполнения программы является изменение сегмента кода – это великолепный прием, позволяющий сокрыть истинные намерения программы. Подобный вид модификации встречается во многих вирусах, защитных механизмах, сетевых червях и прочих программах подобного типа.

В процессе исследования программы, производящей модификацию кода во время выполнения, дизассемблер отображает ее в том виде, в котором она была получена на момент снятия дампа или загрузки исходного файла, рассчитывая на то, что ни одна из машинных команд не изменится в ходе своего выполнения. Если модификацию вовремя не обнаружить, то реконструкция алгоритма будет выполнена неверно.

В связи с этим, возникла задача: на базе интерактивного дизассемблера IDA Pro разработать плагин для поиска модификации сегмента кода во время выполнения программы в ОС Windows.

1. ПРОБЛЕМЫ МОДИФИКАЦИИ СЕГМЕНТА КОДА В ОС WINDOWS

Для выполнения программы она должна быть помещена в память компьютера. В памяти компьютера содержатся и данные, которые могут использоваться исполняемой программой. Нет никакого различия между командами микропроцессора и данными. В определенных ситуациях команды программы могут рассматриваться как данные и, наоборот, данные становятся фрагментами программы. Таким образом, нет никакого препятствия к тому, чтобы программа в процессе выполнения изменяла саму себя [1].

Рассвет эпохи модификации кода во время выполнения программы пришелся на систему MS-DOS, программистами широко использовался подобный код, без которого не обходилась практически ни одна серьезная защита. Да и не только защита, он встречался в компиляторах, компилирующих код в память, распаковщиках исполняемых файлов, полиморфных генераторах и т. д. Во времена отладчиков типа debug.com модификация действительно серьезно затрудняла анализ, однако с появлением IDA Pro и Turbo-Debugger все изменилось.

Уже к середине девяностых годов начался массовый переход пользователей с MS-DOS на Windows 95/Windows NT, и разработчикам пришлось задуматься о переносе накопленного опыта и приемов программирования на новую платформу – от бесконтрольного доступа к памяти и компонентам операционной системы пришлось отвыкать [2].

1.1 Примеры модификации сегмента кода во время выполнения программы в ОС MS Windows

Требование модификации атрибутов доступа к сегменту означает, что программа должна быть скомпилирована с указанием дополнительных атрибутов доступа к сегменту, непредусмотренных по умолчанию, либо в процессе выполнения программы должна быть вызвана функция Windows API VirtualProtect / VirtualProtectEx для добавления этих атрибутов. Выделение необходимого количества памяти для нового кода означает, что данный вид модификации позволяет выделить кусок адресного пространства произвольной длины (в рамках сегмента) под новый код. Иначе модификация производится в рамках статического буфера.

Ниже приведены различные варианты модификации кода во время выполнения [1].

1.1.1 Модификация сегмента кода с использованием функции WriteProcessMemory

Один из способов модификации кода во время исполнения – это использование API-функции WriteProcessMemory. С ее помощью можно писать данные в адресное пространство процесса. Область, куда предполагается писать, должна быть доступна для записи, в противном случае записи не произойдет. Прежде всего, с помощью данной функции вы исправляете код текущего процесса, но не можете увеличить объём памяти, чтобы добавить новый код.

```
.586P
.MODEL FLAT,STDCALL
PROCESS_VM_OPERATION  =     0008H
PROCESS_VM_WRITE      =     0020H
PROCESS_VM_OW     =     PROCESS_VM_OPERATION OR
PROCESS_VM_WRITE

includelib E:\masm32\lib\user32.lib
includelib E:\masm32\lib\kernel32.lib
EXTERN OpenProcess@12:NEAR
EXTERN WriteProcessMemory@20:NEAR
EXTERN GetCurrentProcessId@0:NEAR
;---------------------------------------------
_DATA SEGMENT
OPC DB 0C3H
_DATA ENDS
_TEXT SEGMENT
START:
	CALL GetCurrentProcessId@0
;в EAX идентификатор текущего процесса
	PUSH EAX
	PUSH 1
;дескриптор может наследоваться
	PUSH PROCESS_VM_OW
;желаемый уровень доступа к процессу
	CALL OpenProcess@12
;в EAX дескриптор открытого процесса
	PUSH 0
;игнорируем параметр
	PUSH 1
;кол-во байт, которые будут записаны
```

```
        PUSH OFFSET OPC
;указатель на буфер-источник данных
        PUSH OFFSET RETE
;адрес в памяти процесса, куда собираемся писать
        PUSH EAX
;дескриптор процесса, в память которого мы собираемся
;писать
        CALL WriteProcessMemory@20
;По адресу RETE записывается код C3H. Если это не
;сделать, то будет выполняться бесконечный цикл, и
;программа никогда не закончит свою работу (без воздействий
;извне)

RETE:
        JMP  RETE
        RETN
_TEXT ENDS
END START
```

1.1.2 Модификация сегмента кода с использованием функции VirtualProtectEx

Воспользуемся API-функцией VirtualProtectEx и разрешим доступ к нужным байтам (страницам, на которых располагаются байты), а затем воспользуемся командой MOV: по адресу RETE запишем байт C3H.

```
.586P
.MODEL FLAT,STDCALL
PROCESS_VM_OPERATION  =     0008H
PROCESS_VM_WRITE      =     0020H
PROCESS_VM_OW         =
PROCESS_VM_OPERATION OR PROCESS_VM_WRITE
PAGE_WRITECOPY       = 8
PAGE_EXECUTE              = 10h
includelib "D:\Program Files\Microsoft Visual Studio
.NET\Vc7\PlatformSDK\lib\User32.Lib"
includelib "D:\Program Files\Microsoft Visual Studio
.NET\Vc7\PlatformSDK\lib\Kernel32.Lib"
;импортируемые функции
EXTERN OpenProcess@12:NEAR
EXTERN FlushInstructionCache@12:NEAR
EXTERN VirtualProtectEx@20:NEAR
```

```
EXTERN GetCurrentProcessId@0:NEAR
;---------------------------------------------
_DATA SEGMENT
HANDLE  DD ?
NN      DD ?
_DATA ENDS
_TEXT SEGMENT
START:
      CALL GetCurrentProcessId@0
;открыть текущий процесс
      PUSH EAX
      PUSH 1
      PUSH PROCESS_VM_OW
      CALL OpenProcess@12
;разрешить копирование байта по адресу RETE
      MOV  HANDLE,EAX
      PUSH OFFSET NN
;адрес переменной, которая получит старый атрибут
;первой из страниц (если их несколько)
      PUSH PAGE_WRITECOPY
;устанавливаем атрибут
      PUSH 1
;размер изменяемой области
      PUSH OFFSET RETE
;адрес области памяти, атрибут которой мы будем изменять
      PUSH EAX
;дескриптор процесса, память которого мы модифицируем
      CALL VirtualProtectEx@20
;изменяем байт по адресу RETE
      LEA  EAX,RETE
      MOV  BYTE PTR [EAX],0C3H
;возвращаем байту первоначальный атрибут
      PUSH OFFSET NN
      PUSH PAGE_EXECUTE
      PUSH 1
      PUSH OFFSET RETE
      PUSH HANDLE
      CALL VirtualProtectEx@20
;сбрасываем кэш
      PUSH 1
;размер изменяемой области
```

```
        PUSH OFFSET RETE
;адрес области, которую мы изменили
        PUSH HANDLE
;дескриптор процесса, память которого мы изменяем
        CALL FlushInstructionCache@12
;необходимо, чтобы очистить буфер, содержащий команды.
;если этого не сделать, вероятно, что процессор будет
;использовать для выполнения старые команды, "не заметив"
;изменения в памяти
RETE:
        JMP  RETE
        RETN
_TEXT ENDS
END START
```

Успешное выполнение приведенных примеров подтверждает, что модификация кода программы во время ее выполнения в ОС Windows возможна.

1.2 Интерактивный дизассемблер и отладчик IDA Pro

IDA Pro (https://www.hex-rays.com) – это интерактивный дизассемблер и отладчик одновременно. Она позволяет превратить бинарный код программы в ассемблерный текст, который может быть применен для анализа работы программы.

Три кита, на которых держится анализ исполняемого кода в IDA Pro - это:

- мощное средство анализа исполняемого кода (дизассемблер сам комментирует код и распознает стандартные библиотечные и системные функции), встроенное в дизассемблер. IDA Pro никогда не делает слишком “самоуверенных” предположений. Привилегия на эвристический анализ предоставляется пользователю;
- пользователю предоставляется возможность участвовать в этом анализе, уточнять параметры тех или иных объектов программы, делать исправления;
- наличие библиотеки для написания собственных плагинов (IDA SDK) и встроенный язык программирования (скриптовый язык IDC), весьма близкий по своей структуре к классическому языку С, позволяют значительно наращивать функциональность данного продукта.

IDA используется для анализа вирусов (antivirus companies), исследования защит систем (software security auditing), обратной инженерии (reverse engineering). Хотя IDA и не является декомпилятором (decompiler), она содержит отладчик (debugger) и может анализировать программы на высоком уровне [3].

2. РЕАЛИЗАЦИЯ

Основываясь на том факте, что модификация кода во время выполнения не препятствует трассировке, и что для отладчика она полностью прозрачна, к интерактивному дизассемблеру IDA Pro был написан плагин [4] для проверки наличия модификации сегмента кода в программе во время ее выполнения.

Основными критерием при работе плагина являются: изменение атрибутов сегмента кода во время выполнения программы и наличие функций, способных произвести запись в память процесса.

Для реализации поставленной задачи был применен метод перехвата вызовов API-функций, работающих с атрибутами сегментов, и API-функций, позволяющих произвести запись в память процесса.

2.1 Особенности работы плагина

Для хранения истории изменений атрибутов сегмента кода была создана следующая структура:

```
struct about_seg_struct {
      char seg_name[MAXSTR];             // Имя сегмента
      int seg_attr[NUM_CHANGE_ATTR];  // Атрибуты
                                                  // сегмента
      int attr_counter;                         // Счетчик изменений
      int seg_type;                              // Тип сегмента
      bool change_flag;                       // Флаг изменений
      ea_t seg_addr_start;                   // Адрес начала сегмента
      ea_t seg_addr_end;                     // Адрес завершения
                                                  // сегмента
};
```

При запуске плагина в эту структуру заносятся начальные значения атрибутов. Этот процесс отражен в следующем коде:

```
// Получаем число сегментов
 num_segments = get_segm_qty();
// В цикле проходим через все сегменты и сохраняем их
```

```
// начальные атрибуты
 for (int i = 0; i < num_segments; i++)
 {
	char segName[MAXSTR];
	segment_t *seg = getnseg(i);
	// Получаем имя сегмента
	get_segm_name(seg, segName, sizeof(segName)-1);
	qstrncpy(normal_seg_attr[i].seg_name, segName, MAXSTR);

	normal_seg_attr[i].seg_type = seg->type;
	normal_seg_attr[i].seg_attr[0] = seg->perm;
	normal_seg_attr[i].seg_addr_start = seg->startEA;
	normal_seg_attr[i].seg_addr_end = seg->endEA;
	normal_seg_attr[i].func_flag[0] = true;
	normal_seg_attr[i].attr_counter = 1;
	normal_seg_attr[i].change_flag = false;
}
```

Дальнейшая работа плагина основана на перехвате вызовов API-функций. Рассмотрим алгоритм, используемый в плагине, для анализа перехваченных функций:

1. если перехвачен вызов API-функции VirtualProtect(Ex) (см. Раздел 2.6. Пример анализа перехваченной функции) – получаем из стека аргументы, переданные функции в момент ее вызова: устанавливаемый атрибут, адрес области памяти, атрибут которой планируется изменять и размер изменяемой области.
2. если перехвачен вызов API-функции WriteProcessMemory – получаем из стека аргументы, переданные функции в момент ее вызова: адрес в памяти процесса, куда намеревались писать, количество байтов, которое планировалось записать.

Решение о наличии или отсутствии модификации кода в программе принимается на основании истории изменения атрибутов сегментов. Процесс проверки изменения атрибутов представлен в следующем коде:

```
// В цикле проходим через все сегменты
for(int i = 0; i < num_segments; i++)
{
// В цикле проходим по всем изменениям в сегменте
 for(int j = 0; j < normal_seg_attr[i].attr_counter; j++)
 {
  // Сравниваем атрибуты
```

```
  if((normal_seg_attr[i].seg_attr[j]==0x04)||
    (normal_seg_attr[i].seg_attr[j] == 0x08))
    start_flag = true;

if(start_flag&&((normal_seg_attr[i].seg_attr[j]==0x10)||
     (normal_seg_attr[i].seg_attr[j] == 0x20)))
       {
// Записываем сведения о наличии модификации в файл
       char str_error[MAXSTR];
       qsnprintf(str_error,sizeof(str_error)-1,"Change attribute in segment
       <%s> and type <%d>!\n",
       normal_seg_attr[i].seg_name,normal_seg_attr[i].seg_type);
       ewrite(fp2, str_error, strlen(str_error));
       }
       if((normal_seg_attr[i].seg_attr[j]==0x40)||
         (normal_seg_attr[i].seg_attr[j] == 0x80))
         {
         // Записываем сведения о наличии модификации в файл
char str_error[MAXSTR];
          qsnprintf(str_error,sizeof(str_error)-1,"Change attribute in
          segment <%s> and type <%d>!\n",
          normal_seg_attr[i].seg_name,normal_seg_attr[i].seg_type);
         ewrite(fp2, str_error, strlen(str_error));
         //
         }
       }
       start_flag = false;
       }
```

2.2 Правила компиляции плагина

Для создания проекта и компиляции плагина можно воспользоваться средой разработки MS Visual Studio 2005 либо ее более ранними версиями.

Рассмотрим процесс настройки окружения в этой среде:

1. Переходим **File->New->Project…** (Ctrl-Shift-N)
2. Разворачиваем каталог **Visual C++ Projects**, переходим в подкаталог **Win32**, и затем выбираем **Win32 Project**. Задаем любое понравившееся имя проекта и нажимаем OK.

3. Далее появится Win32 Application Wizard, нажимаем **Application Settings** и выбираем **Windows Application**, затем выбираем **Empty Project**. Нажимаем **Finish**.
4. В **Solutions Explorer** переходим в каталог **Source Files** и выбираем **Add->Add New Item...**
5. Выбираем **C++ File (.cpp)** и подходящее имя файла. Нажимаем **Open**.
6. Переходим в **Project->имя_проекта Properties...**
7. Выбираем следующие настройки:
   Configuration Properties->General: выбираем **Configuration Type** Динамическая Библиотека (.dll)
   C/C++->General: **Detect 64-bit Portability Issue** отмечаем **No**
   C/C++->General: устанавливаем **Debug Information Format** в Disabled
   C/C++->General: в **Additional Include** добавляем путь к каталогу SDK include, например, C:\IDA\SDK\Include
   C/C++->Preprocessor: добавляем __NT__;__IDP__ к **Preprocessor Definitions C/C++->Code Generation**: отключаем **Buffer Security Check** и **Basic Runtime Checks**, устанавливаем **Runtime Library** в Single Threaded
   C/C++->Advanced: устанавливаем **__stdcall**
   Linker->General: выбираем **Output File**, заменяем a .exe на a .plw
   Linker->General: добавляем путь к libvc.wXX в **Additional Library Directories**, например, C:\IDA\SDK\libvc.w32
   Linker->Input: добавляем ida.lib в **Aditional Dependencies**
   Linker->Debugging: отмечаем **No** в **Generate Debug Info**
   Linker->Command Line: добавляем /EXPORT:PLUGIN
   Build Events->Post-Build Event: устанавливаем **Command-line** в idag.exe. Нажимаем **OK**.

2.3 Установка плагина

Для загрузки плагина в IDA необходимо скомпилированный .plw файл поместить в каталог plugins. При каждом запуске IDA сканирует этот каталог и загружает плагины.

2.4 Руководство по запуску плагина

Запуск плагина осуществляется из меню Edit->Plugins->SMCTracer 4.0 (рис.1).

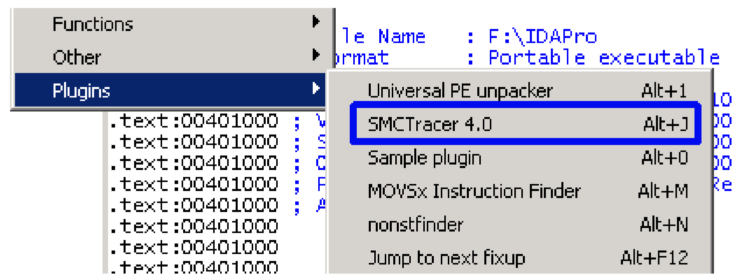

Рисунок 1. Меню загрузки плагина в IDA Pro

Далее, необходимо указать путь к файлу результатов проверки (рис. 2).

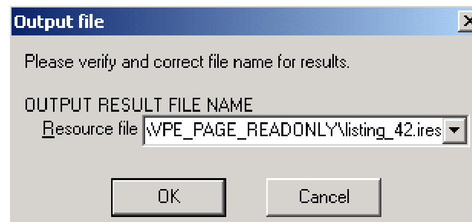

Рисунок 2. Окно задания имени файла с результатами проверки

2.5 Результаты работы плагина

Тестирование плагина проводилось на примерах, рассмотренных в разделе 1.1 (примеры модификации сегмента кода во время выполнения программы в ОС MS Windows). Результаты работы плагина следующие:

1) запуск плагина для программы, в которой присутствует модификация сегмента кода с использование функции WriteProcessMemory

   Detection a call WriteProcessMemory
   (return address: 00401023),
   write data to address: 00401023, size of data: 1
   WriteProcessMemory **write data in segment: <.text>**,
   at 401000 to 401200
   with permission: <Read and Exec>
   and type: **<Code segment>**

2) запуск плагина для программы, в которой присутствует модификация сегмента кода с использованием функции VirtualProtectEx

   Detection a Call VirtualProtectEx
   (return address: 00401028),

new attribute: PAGE_WRITECOPY,
address of modification: 0040105C, size of data: 1
Change attribute in segment 21 is named **_text** permission 8 type 2, function 1 counter 1

Из полученных результатов видно, что созданный плагин сумел обнаружить вызовы API-функций: WriteProcessMemory и VirtualProtectEx. Помимо факта о наличии вызова функции WriteProcessMemory, плагину удалось узнать сегмент, в который происходит запись, тип сегмента и его атрибуты. Для функции VirtualProtectEx плагин указал сегмент, атрибут которого изменяется, и новый атрибут для этого сегмента.

2.6 Пример анализа перехваченной функции

Рассмотрим пример анализа перехваченной функции:

```
if(ea == gpa_vp)
{
  // Перехвачен вызов API-функции VirtualProtect
  //
  regval_t rv;
  // Пробуем узнать значение регистра ESP
  // (адрес возврата из функции VirtualProtect)
  if (get_reg_val("esp", &rv))
  {
   ea_t esp = rv.ival;
   invalidate_dbgmem_contents(esp, 1024);
   //
   // Теперь ret_vp содержит значение регистра ESP
   //
   ret_vp = get_long(esp);
   //
   // Из стека “вытаскиваем” параметры, с которыми
   // вызвана функция VirtualProtect:
   // указатель на базовый адрес страницы
   ea_t nameaddr_4 = get_long(esp+4);
   // размер области, атрибут которой изменяется
   ea_t nameaddr_8 = get_long(esp+8);
   // новый атрибут
   ea_t nameaddr_12 = get_long(esp+12);
   // выходной параметр
   ea_t nameaddr_16 = get_long(esp+16);
   //
```

```
  // Сохраняем результаты вызова VirtualProtect в файл
  //
  char log_api_buf[MAXSTR];
  qsnprintf(log_api_buf, sizeof(log_api_buf)-1, "Detection a
  Call VirtualProtect (return address: 00%a), change
  attribut: %s, address of modification: 00%a, size of data:
  %a\n",    ret_vp,    AttrVPEName(nameaddr_12),    nameaddr_4,
nameaddr_12);
  ewrite(fp1, log_api_buf, strlen(log_api_buf));
  //
  // Определяем сегмент, атрибут которого изменился:
  // получаем дескриптор сегмента по адресу
  //
  segment_t* seg_vp = getseg(nameaddr_4);
  if (seg_vp)
  {
   char segName[MAXSTR];
   // Получаем имя сегмента по дескриптору
   get_segm_name(seg_vp, segName, MAXSTR);
   // Следим, чтобы длина, переданная VirtualProtect
   // в качестве параметра, не перекрывала сегменты
   //
   for (int j = 0; j <= nameaddr_8; j++)
   {
    for (int i = 0; i < num_segments; i++)
    {
    //
    // Поверяем, изменился ли атрибут сегмента
    //
     if((normal_seg_attr[i].seg_addr_start<=nameaddr_4+j) &&
(normal_seg_attr[i].seg_addr_end>=nameaddr_4+j) &&
(!normal_seg_attr[i].change_flag))
        {
    //
    // Запоминаем новый атрибут и устанавливаем флаг
    //
        normal_seg_attr[i].change_flag = true;
        normal_seg_attr[i].seg_attr[normal_seg_attr[i].
        attr_counter] = nameaddr_12;
    //
    // Записываем результат в файл
```

```
    //
          char str43[MAXSTR];
          qsnprintf(str43, sizeof(str43)-1, "Change attr
          in segment %d is named %s permission %d type %d, func
          %d counter %d\n",
          i,
normal_seg_attr[i].seg_name,
normal_seg_attr[i].seg_attr[normal_seg_attr[i.
          attr_counter],
          normal_seg_attr[i].seg_type,
          normal_seg_attr[i].func_flag[normal_seg_attr[i].
          attr_counter],
          normal_seg_attr[i].attr_counter);
          ewrite(fp1, str43, strlen(str43));
    //
    // Увеличиваем счетчик числа изменений
    //
          normal_seg_attr[i].attr_counter++;                    }

      }
    }
  }

  // Получаем имя сегмента, из которого была вызвана
  // VirtualProtect
  char segName[MAXSTR];
  segment_t *seg = getseg(ret_vp);
  //
  // Проверяем дескриптор сегмента
  //
  if(seg)
  {
    char perm_seg_buf[MAXSTR];
    get_true_segm_name(seg, segName, MAXSTR);

  // Записываем результат в файл
    qsnprintf(perm_seg_buf, sizeof(perm_seg_buf)-1, "Segment
    for return address: <%s>, at %a to %a with permission:
    <%s> and    type: <%s> \n\n", segName, seg->startEA, seg
    ->endEA, SegPermName(seg->perm), SegTypeName(seg->type));
    ewrite(fp1, perm_seg_buf, strlen(perm_seg_buf));
```

```
  }
  // Продолжаем процесс отладки программы
  continue_process();
 }
 else
 {
  // Если функция получения содержимого регистра
  // вернула ошибку
  msg("Error! in get_reg_val(esp)\n");
  clear_requests_queue();
  request_exit_process();
  run_requests();
 }
}
```

ЗАКЛЮЧЕНИЕ

В результате проделанной работы к интерактивному дизассемблеру IDA Pro был создан плагин. Данный продукт, путем автоматизированной отладки, способен обнаруживать признаки наличия модификации сегмента кода во время выполнения программы.

Сведения, полученные в результате работы плагина применимы при исследованиях компьютерных вирусов, защитных механизмов, сетевых червей и прочих программ, которые используют модификацию сегмента кода во время своего выполнения.

Благодаря огромным возможностям IDA, функциональность созданного плагина легко расширить. В качестве наиболее перспективных направлений расширения возможностей плагина можно выделить следующие:

- обнаружение модификации сегмента данных и стека во время выполнения программы,
- объединение статических и динамических возможностей IDA в единое статическо-динамическое средство обнаружения модификации кода. Таким образом, попытаться разрешить проблему построения полного алгоритма при динамическом исследовании программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ассемблер и дизассемблирование / Пирогов В. Ю. – СПб.: БХВ-Петербург, 2006. – 464 с.

2. Компьютерные вирусы изнутри и снаружи / К. Касперски. – СПб.: Питер, 2006. – 522 с.

3. Фундаментальные основы хакерства. Искусство дизассемблирования / К. Касперски. – М.: СОЛОН-Пресс, 2005. – 448 с.

4. Steve Micallef. IDA PLUG-IN WRITING IN C/C++ [Электронный ресурс] : 2009. – URL: http://www.binarypool.com/idapluginwriting/idapw.pdf (02.11.2015)