

АНАЛИЗ И ЭКСПЛУАТАЦИЯ УЯЗВИМОСТИ ТИПА: ПЕРЕПОЛНЕНИЕ БУФЕРА

Д. Д. Комиссаренко^{1, а}

¹ *Національний технічний університет України «Київський політехнічний інститут»*

Аннотация

С помощью реверс инжиниринга произведен анализ поведения видео-плеера BlazedDVD 5.0 Pro при воздействии на присущую ему уязвимость типа: переполнение буфера. Разобраны основные принципы работы регистров, структуры памяти приложения, написаны скрипты, которые будут эксплуатировать уязвимость переполнения буфера.

Ключевые слова: уязвимость, эксплойт, переполнение буфера

Введение

Информационная безопасность включает в себя много различных подразделов. В данной работе рассмотрен один из них – это Reverse Engineering. Обратная разработка (обратный инжиниринг, реверс-инжиниринг) – это исследование некоторого устройства или программы, а также документации на него с целью понять принцип его работы; например, чтобы обнаружить недокументированные возможности (в том числе «программные закладки»), сделать изменение, или воспроизвести устройство, программу или иной объект с аналогичными функциями, но без копирования как такового. Актуальность работы заключается в том, что именно с возможностью использования атак на переполнение буфера для удаленного внедрения вредоносного кода связаны непрекращающиеся дискуссии и шумиха вокруг атак этого класса. Данная уязвимость будет существовать до тех пор, пока программисты будут пользоваться стандартными функциями для копирования данных в буфер, а значит, данная уязвимость будет существовать еще очень долго.

1. Определение уязвимости переполнение буфера

Переполнение буфера (Buffer Overflow) – явление, возникающее, когда компьютерная программа записывает данные за пределами выделенного в памяти буфера [1]. Переполнение буфера обычно возникает из-за неправильной работы с данными, полученными извне, и памятью, при отсутствии жесткой защиты со стороны подсистемы программирования (компилятор или интерпретатор) и операционной системы. В результате переполнения могут быть испорчены данные, расположенные следом за буфером (или перед ним). Переполнение буфера является наиболее популярным способом взлома компьютерных систем, так как большинство языков высокого уровня используют технологию стекового кадра – размещение

данных в стеке процесса, смешивая данные программы с управляющими данными (в том числе адреса начала стекового кадра и адреса возврата из исполняемой функции). Переполнение буфера может вызывать аварийное завершение или зависание программы, ведущее к отказу обслуживания (Denial of Service, DoS), позволяют злоумышленнику загрузить и выполнить произвольный машинный код от имени программы и с правами учетной записи, от которой она выполняется [2].

2. Пример уязвимой программы

Скомпилировав следующую программу, её можно использовать для генерации ошибок переполнения буфера. Первый аргумент командной строки программа принимает как текст, которым заполняется буфер.

```
/* overflow.c – демонстрирует процесс переполнения буфера */
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char buffer[10];
    if (argc < 2)
    {
        fprintf(stderr, "ИСПОЛЬЗОВАНИЕ: %s_строка\n", argv[0]);
        return 1;
    }
    strcpy(buffer, argv[1]);
    return 0;
}
```

Строки в 10 и более символов будут вызывать переполнение, хотя это может и не приводить к ошибке сегментации.

Эта программа может быть переписана следующим образом, с использованием функции strncpy для предотвращения переполнения. Однако, следует учитывать, что простое отбрасывание лишних данных, как в этом примере, также может приводить к нежелательным последствиям, в том числе к повышению привилегий (например, если по данным

^аkemstan@mail.ru

определяются права пользователя, и после отображения они совпадут с другими, хранящимися в системе). Как правило, требуется более тщательная обработка таких ситуаций – например, выделение буфера нужного размера.

```
/* better.c – демонстрирует, как исправить
ошибку */
#include <stdio.h>
#include <string.h>
#define BUFFER_SIZE 10
int main(int argc, char *argv[])
{
    char buffer[BUFFER_SIZE];
    if (argc < 2)
    {
        fprintf(stderr, "ИСПОЛЬЗОВАНИЕ: %s_строка\n",
            argv[0]);
        return 1;
    }
    strncpy(buffer, argv[1], BUFFER_SIZE);
    buffer[BUFFER_SIZE-1] = '\0';
    return 0;
}
```

3. Способы защиты от переполнения буфера в SEH

SEH – это сокращение от Structured Exception Handling и переводится как структурная обработка исключения. Исключение – это событие к которому приводит ненормальное (неправильное) поведение кода, например, исключение, возникает при обращении к невыделенной области памяти (и как следствие возникает нарушение доступа) или при делении на ноль. Исключения бывают двух типов:

- Аппаратные, которые генерирует (возбуждает) процессор;
- Программные, которые могут генерировать операционная система или приложение.

У каждого исключения есть свой код, их определения можно посмотреть в файлах `winbase.h` и `winnt.h`. Например, исключение, связанное с нарушением доступа имеет следующее определение и код:

```
#define EXCEPTION_ACCESS_VIOLATION
STATUS_ACCESS_VIOLATION
#define STATUS_ACCESS_VIOLATION
((DWORD) 0xC0000005L)
```

Любое исключение, как аппаратное, так и программное, можно обработать. Ниже представлен синтаксис обработчика исключения:

```
__try {
    // защищенный код,
    // который помещается в SEH-фрейм
}
__except (фильтр исключений) {
    // обработчик исключений
}
```

Если при выполнении кода из блока `__try{...}`, возникнет исключение, то операционная система перехватит его и приступит к поиску блока `__except`. Найдя его, она передаст управление фильтру исключений. Фильтр исключений может получить код исключения и на основе этого кода принять решение,

передать управление обработчику или же сказать системе, чтобы она искала предыдущий по вложенности блок `__except`. Фильтр исключений может возвращать одно из трех значений (идентификаторов), которые определены в файле `except.h`:

```
EXCEPTION_EXECUTE_HANDLER 1
EXCEPTION_CONTINUE_SEARCH 0
EXCEPTION_CONTINUE_EXECUTION -1
```

Идентификатор `EXCEPTION_EXECUTE_HANDLER` говорит системе, что для этого блока `__try` есть обработчик исключения и он готов обработать это исключение. Идентификатор `EXCEPTION_CONTINUE_SEARCH` говорит системе, чтобы она искала предыдущий по вложенности блок `__except`, а идентификатор `EXCEPTION_CONTINUE_EXECUTION` говорит ей, чтобы она снова продолжала выполнение с того места кода, который вызвал исключение. Рассмотрим пример с обработчиком исключений:

```
<except.c>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
void
except_sample() {
    __try {
        printf("start_of_ __try_\r\n");
        // возбуждаем исключение
        *(char *)0 = 0;
        // этот кусок кода никогда не выполнится
        printf("end_of_ __try_\r\n");
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        printf("!!!_exception_!!!\r\n");
    }
    printf("end_of_except_sample\r\n");
}
int
main(int argc, char **argv)
{
    except_sample();
    printf("end_of_main\r\n");
    return 0;
}
```

```
C:\samples>except.exe
start of __try
!!! exception !!!
end of except_sample
end of main
C:\samples>
```

В этом примере в качестве фильтра исключения сразу подставляется идентификатор `EXCEPTION_EXECUTE_HANDLER`, тем самым перехватываются все исключения, которые могут возникнуть при выполнении кода из блока `__try{...}`. Помимо обработчиков исключения есть еще не менее важная вещь – обработчик завершения. Смысл его в том, что независимо от того, что произойдет в защищенном участке кода, обработчик завершения будет выполнен. Другими словами, вне зависимости от того выполнится защищенный код целиком без ошибок или все же произойдет ошибка (исключение), обработчик завершения получит управление. Главное отличие обработчика завершения от обработчика исключения заключается в том, что фильтры и обработчики исключений получают управление

непосредственно от операционной системы и нагрузка на компилятор минимальна. Теперь посмотрим, как операционная система находит обработчики исключений и как передает им управление. Каждый защищенный блок кода, получив управление, прописывает свой обработчик в специальную область, называемую TIB (Thread Information Block), и при этом сохраняет старый обработчик. Происходит это следующим образом: В TIB по смещению 0 отведена специальная область для сохранения адреса текущего фильтра исключений. Компилятор перед защищенным блоком вставляет специальный код, приблизительно следующего вида:

```
push _Handler; помещаем в стек адрес нашего фильтра
mov eax, FS:[0]; записываем в eax адрес структуры предыдущего обработчика
push eax; записываем его в стек
mov FS:[0], esp; устанавливаем в TIB адрес нашей структуры
```

Этот код формирует в стеке структуру (SEH-фрейм) вида:

```
struct EXCEPTION_REGISTRATION {
    EXCEPTION_REGISTRATION *prev;
    DWORD *handler;
};
```

В этой структуре указывается адрес предыдущей (по вложенности) структуры и адрес фильтра. Таким образом, формируется цепочка структур обработчиков исключений, как показано на рис. 1

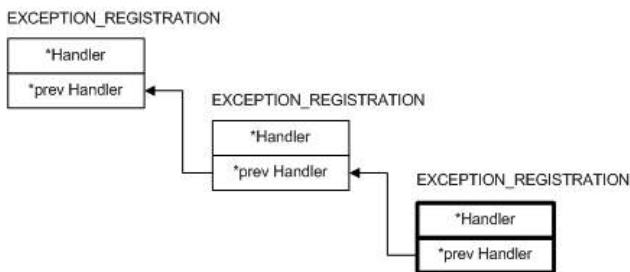


Рис. 1. Цепочка структур EXCEPTION_REGISTRATION.

4. Практическая часть

Использоваться для отладки приложения будет дебагер ImmunityDebugger. Для написания эксплоита будет использоваться скриптовый язык Perl и ассемблер для (получения опп-кода). Известно, что переполнение буфера происходит при приеме PLF-файла, для того, чтобы определить, действительно ли буфер переполняется при приеме файла, составляется специальный PLF-файл, который вызовет переполнение. Запускаем BlazedDVD под дебагером (рис. 2).

Плеером принимаем наш сгенерированный файл (рис. 3).

В результате видно, что регистр EIP перезаписан, и ещё был перезаписан нашими значениями 0x41414141 указатель на SEH. Очевидны 2 вектора возможной эксплуатации данной уязвимости: через

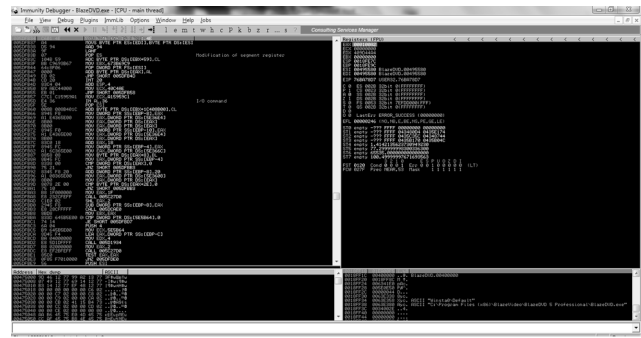


Рис. 2. Запуск приложения под дебагером



Рис. 3. Перезапись EIP и SEH

перезапись EIP, и через перезапись SEH [5]. В исследовательских и учебных целях будет произведена эксплуатация через перезапись SEH.

4.1. Эксплуатация через перезапись SEH

Переполнение SEH происходит следующим образом (рис. 4):

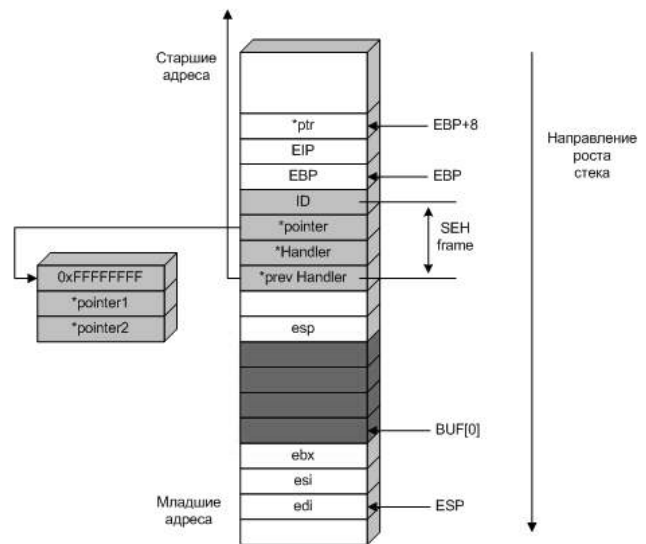


Рис. 4. Изображение стека

Между буфером BUF и регистрами EIP и EBP находятся блоки SEH-фреймов и блок с адресом вершины стека. Блок SEH-frame есть ни что иное, как структура EXCEPTION_REGISTRATION, о которой рассказывалось ранее. Первое поле ID необходимо для раскрутки (unwind) и в самом первом звене цепочки SEH оно принимает значение 0xFFFFFFFF, и сигнализирует об окончании, т.е. является признаком последнего звена. Каждая последующая функция, которая

имеет обработчик исключений, устанавливают ID в 0, и увеличивает это значение на единицу с каждым последующим вложенным блоком `__try`.

Следующее поле, которое на рисунке обозначено как `*pointer`, является указателем на область памяти, в которой содержится структура вида:

```
long ID — принимает значение 0xFFFFFFFF
long *pointer1
long *pointer2
```

Этой структурой оперирует фильтр исключений. Указатель `pointer1` содержит адрес блока кода, который возвращает одно из трех значений идентификатора фильтра исключений:

```
EXCEPTION_EXECUTE_HANDLER,
EXCEPTION_CONTINUE_SEARCH
```

или

```
EXCEPTION_CONTINUE_EXECUTION.
```

Следующий указатель (`pointer2`) указывает на начало блока обработчика исключений. Последующие поля `*Handler` и `*prev_Handler` указывают соответственно на фильтр исключений и на предыдущий SEH-фрейм. Очевидно, что, переполняя буфер, возможно изменять SEH-фрейм как угодно. Например подменить адрес фильтра исключений и через исключение передать ему управление. Для эксплуатации уязвимости через перезапись SEH-фрейма так же необходимо определить, какими символами из паттерна перезаписывается SEH-фрейм. Для этого повторяется процедура создания паттерна с помощью скрипта `pattern_create` (Metasploit), и нахождение символа с которого начинается перезапись SEH. Результат работы скрипта `pattern_create` показал, что перезапись SEH-фрейма начинается с 609 позиции. Следовательно, символами на позициях 609 — 612 мы перезаписываем `*prev_Handler`, а так как нам необходимо перезаписать `*pointer`, то перезаписываем `*prev_Handler` следующей инструкцией `\xeb\x06\x90\x90`. Первые два байта — джамп вперед на 6 байт (2 байта `\x90` (NOP) и 4 байта адреса на SEH), то есть, за обработчик на наш шеллкод. В момент вызова фильтра исключений адрес SEH-фрейма находится в стеке по адресу `esp+8`. Отсюда следует, что для того, чтобы передать управление на наш шеллкод, нам необходимо найти в памяти следующие инструкции: `pop reg, pop reg, ret`. Данный поиск вернул нам «0x616299F9». Теперь, когда у нас все готово, мы можем приступить к созданию итогового эксплоита:

```
#!/usr/bin/perl
$junk = "A"x 608;
$seh = "\xeb\x06\x90\x90";
$SEH = pack('V', 0x616299F9);
$nop = "\x90"x 24;
$shell = "\x29\xc9\x83\xe9\xdd\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x7d\xe6\xe7\x4e\x83\xeb\xfc\xe2\xf4\x81\x0e\xa3\x4e\x7d\xe6\x6c\x0b\x41\x6d\x9b\x4b\x05\xe7\x08\xc5\x32\xfe\x6c\x11\x5d\xe7\x0c\x07\xf6\xd2\x6c\x4f\x93\xd7\x27\xd7\xd1\x62\x27\x3a\x7a\x27\x2d\x43\x7c\x24\x0c\xba\x46\xb2\xc3\x4a\x08\x03\x6c\x11\x59\xe7\x0c\x28\xf6\xea\xac\xc5\x22\xfa\xe6\xa5\xf6\xfa\x6c\x4f\x96\x6f\xbb\x6a\x79\x25\xd6\x8e\x19\x6d\xa7\x7e\xf8\x26\x9f\x42\xf6\xa6\xeb\xc5\x0d\xfa\x4a\xc5\x15\xee\x0c\x47\xf6\x66\x57\x4e\x7d\xe6\x6c\x26\x41\xb9\xd6\xb8\x1d\xb0\x6e\xb6\xfe\x26\x9c\x1e\x15\x16\x6d\x4a\x22\x8e\x7f\xb0\xf7\xe8\xb0\xb1\x9a\x85\x86\x22\x1e\xc8\x82\x36\x18\xe6\xe7\x4e";
$sploit = $junk.$seh.$SEH.$nop.$shell;
open(FILE, ">_trash.plf");
print FILE $sploit;
close(FILE);
print "Gotovo";
```

Тестирование показало, что данный эксплоит генерирует PLF-файл, который вызывает переполнение буфера и путем перезаписи SEH-фрейма выполняет произвольный код на целевой системе.

5. Выводы

В ходе выполнения работы были рассмотрены организация памяти в компьютерной системе, выполнение программного кода на машинном уровне, рассмотрено понятие переполнение буфера и эксплуатация данной уязвимости путем перезаписи адреса фильтра в SEH-фрейме.

Перечень использованных источников

1. Фостер Д. М., Прайс М. Защита от взлома: сокет, эксплоиты, шелл-код — 2006. — 784 с.
2. Фостер Д. М. Разработка средств безопасности и эксплоитов — 2007. — 432 с.
3. Панов А. В. Реверсинг и защита программ от взлома. — 2006. — 256 с.
4. Sikorski M., Honig A. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. — 2012. — 513 p.
5. Eilam E. Reversing: Secrets of Reverse Engineering — 2005. — 404 p.