# Атаки, основанные на переполнении буфера и контрмеры по их предотвращению

Эссе по курсу "Защита информации", кафедра радиотехники,
Московский физико-технический институт (ГУ МФТИ),
http://www.re.mipt.ru/infsec

Антон Поляков

1 мая 2005 г.

## 1 Введение

В последние 15-20 лет в компьютерном мире происходят очень большие и очень значительные изменения. Меняется система ценностей. Информация становится наиболее важным и ценным ресурсом. Кто владеет информацией, тот владеет всем. В связи с этим, меняются способы и средства ведения "войны". Взломы компьютерных систем приносят убытки в сотни, тысячи раз превышающие стоимость этих самих систем. В связи с этим, на первое место выходит проблема обеспечения безопасности компьютерных программ, их отказоустойчивости. Одной из основных проблем современного программного обеспечения является проблема переполнения буфера. Удаленное переполнение буфера может привести к самым различным последствиям - от аварийного завершения сервиса до получения полного контроля на удаленной машине. Компьютерные вирусы также активно используют атаки, основанные на переполнении буфера для проникновения на уязвимую машину.

## 2 Переполнение буфера

Сначала определим понятия, которые мы будем использовать. Мы будем рассматривать атаки переполнения буфера в среде операционной систе-

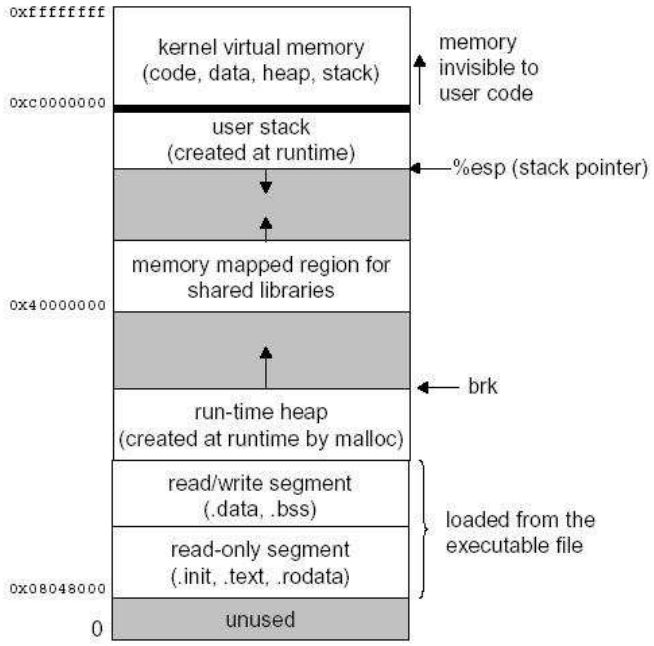


Рис. 1: Память процесса

мы Linux. Под буфером мы подразумеваем некоторый непрерывный участок памяти, в котором могут располагаться данные одного типа. Например, в языке C таковыми являются строки, которые представляют собой массивы символов. Динамические массивы и буферы располагаются в области памяти называемой стеком. Мы будем рассматривать переполнения буфера в стеке на x86 архитектуре

## 2.1 Организация памяти при выполнении программы

Когда процесс загружается в память, для него выделяются три области памяти - текст, данные и стек. В регионе, называемом текст содержатся инструкции программы, обычно эта область памяти помечается как read-only. В области данных хранятся статические переменные.

## 2.2 Стек

Как было указано выше, нас будет интересовать область памяти, занимаемая стеком. В качестве указателя на вершину стек используется регистр

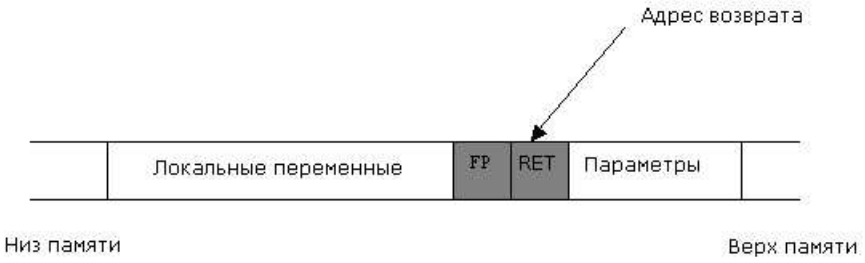


Рис. 2: Стек при вызове функции

SP (stack pointer). Дно стека всегда находится у фиксированного адреса. На архитектуре x86 стек растет вниз, то есть вершина стека находится в ячейках памяти, с меньшим адресом нежели дно стека. Чтобы адресовать локальные переменные в стеке удобно использовать некоторый фиксированный указатель, называемый frame pointer, который является точкой отсчета для каждой функции. Так как при операциях PUSH и POP со стеком его адрес не меняется, то не меняются и расстояния от него до переменных функции. Когда вызывается функция происходят ряд действий:

- в стеке сохраняются значения параметров функции
- в стеке сохраняется предыдущий указатель на вершину стека
- в стеке сохраняется предыдущий указатель на начало кадра стека (FP, frame pointer)
- текущий SP становится FP

Таким образом, при вызове функции стек имеет примерно вид, показанный на Рис. 2.

При завершении функции, считываются из стека сохраненные значения SP и FP и продолжается выполнение вызывающей функции с той команды, на которую указывает SP (или непосредственно со следующей, что зависит от реализации).

## 2.3 Уязвимости переполнения буфера

Суть атак, основанных на переполнении буфера лежит в следующем: мы записываем в буфер некоторой функции данные, выходим за границу буфера и переписываем RET адрес. По окончании выполнения функции, в

регистр SP будет загружен RET адрес, который мы переписали и выполнение начнется с инструкции, на которую указывает этот адрес. Чаще всего, атакующий записывает в буфер бинарный злонамеренный код, а в RET записывает адрес, указывающий на начало этого буфера. Но этот адрес угадать довольно сложно, к тому же он может меняться в зависимости от системы и условий компиляции. Поэтому, чтобы повысить вероятность попасть в буфер, злоумышленник записывает некоторое количество пустых циклов в начало буфера и только затем злонамеренный код. Пустая операция, или NOP (not operation) используется для организации временной задержки при выполнении кода.С использованием NOP'ов нам не надо указывать в RET точный адрес начала буфера, достаточно попасть в один из NOP'ов после чего, с некоторой задержкой, будет выполнен код, следующий за NOP'ами

# 3 Защита от атак переполнения буфера

Вкратце рассмотрев суть проблемы переполнения буфера, обратимся к методам если не полного предотвращения, то хотя бы предупреждения таких атак.

## 3.1 Неисполнимый стек

Одним из самых очевидных решений проблем переполнение буфера является запрещение исполнения какого-либо кода из стека. Так как злонамеренный код может попасть в программу через какой-то буфер, то он обязан располагаться в стеке, а не в сегменте кода (текста). Поэтому, запретив исполнение кода из стека мы автоматически избавляемся от проблемы переполнения буфера (точнее, мы можем быть уверены, что выполнение кода, введенного в буфер не произойдет). Но у этого решения есть несколько минусов

- некоторые программы, например GNU C Complier (gcc) используют выполнение кода в стеке. GCC использует исполнение кода в стеке для так называемых *trampoline*-функций. Так называются небольшие куски кода, создаваемые во время исполнения. Они необходимы компилятору для правильной работы
- Linux использует исполнимый пользовательский стек для обработки сигналов

- Функциональные языки программирования и некоторые другие программы полагаются на исполнимый стек для генерации кода 'на лету'

Эти проблемы решаются в конкретных реализациях функциональности неисполнимого стека по-разному.

### 3.1.1 Openwall patch

Известный патч от Solar Designer (Openwall проект) для Linux ядра обеспечивает неисполнимый стек для приложений. Проблема с gcc и обработкой сигналов решается путем разрешения "кратковременного"исполнения кода из стека. Те при работе gcc динамически обнаруживается необходимость в исполнении кода из стека и на необходимое время такая возможность дается. Тоже самое происходит и с обработчиками сигналов. Но это не решает проблему целиком. Во-первых, во время работы обработчика сигнала может произойти переполнение буфера и, как следствие, исполнение кода. Во-вторых злоумышленник может располагать код не в стеке, а, например, в куче (heap) и просто переписать RET адрес таким образом, чтобы он указывал на нужный адрес или просто нарушить поток исполнения программы.

### 3.1.2 GRSecurity patch

Другой вариант - набор патчей PaX от проекта grsecurity.org. Эти патчи позволяют организовать неисполнимые страницы памяти (в частности,стек). В наборе PaX содержится патч EMUTRAP. Его задачей является эмулирование необходимых последовательностей инструкций, которые генерируются во время исполнения. Но PaX эмулирует данные инструкции только для работы gcc и обработчиков сигналов ядра (однако, есть и еще возможные варианты, приложения, например - wine). Когда процесс пытается исполнить код из неисполнимой памяти, PAGEEXEC и SEGMEXEC логика (другая часть PaX) генерирует ошибку нарушения доступа к страницам памяти, а, обработчик ошибок, в свою очередь, передает управление EMUTRAP, который и выполняет дальнейшую работу по эмулированию необходимой функциональности. Сам EMUTRAP архитектурно-зависимый и в данный момент существует только для архитектуры IA-32.

## 3.2 Защита от переписывания RET адреса

Существуют программы, которые помогают предотвратить перезапись RET адреса функции. Хотя, они и не предотвращают переполнения буфера, однако, усложняют задачу атакующему.

### 3.2.1 `StackShield`

`StackShield` - это свободно распространяемый патч для GCC, который предотвращает попытку перезаписи RET. Перед входом в функцию, `StackShield` сохраняет адрес возврата (RET) в безопасном месте (обычно - в начале сегмента данных). Когда функция завершает свое исполнение, значение из стека RET сравнивается с тем, которое лежит в сегменте данных и, при несовпадении, происходит аварийное завершение программы.

### 3.2.2 `StackGuard`

`StackGuard` - другой патч для GCC, который так же ставит своей целью предотвращение перезаписи RET адреса. Он делает это путем добавления сразу после RET адреса некоторого ключевого слова ("*canaryword*"). Таким образом, невозможно переписать RET не переписав при этом ключевое слово. На выходе из функции проверяется значение ключевого слова, сохраненного в стеке и, при изменении, происходит аварийный останов.

## 3.3 Проверка границ массивов при компиляции

Существуют патчи для GCC, а так же другие компиляторы, которые могут осуществлять проверку выхода за границы массивов при компиляции программы. Примером может служить Compaq C Compiler for Tru64 UNIX. Однако, этот компилятор поддерживает только проверку явного выхода за границы массива, например:

```
char buffer[256];
int i=280;
buffer[i]=bad_code;
```

Этот код при компиляции вызовет ошибку, так как индекс i лежит за границами выделенной для буфера памяти. Однако, все же остается возможность неявного переполнения, например так:

```
char buffer[256];
```

```
void f(char* p)
{
    int bad_code;
    p[280]=bad_code;
}

f(buffer);
```

При таком вызове функции f произойдет передача указателя на начало буфера `buffer` и проверка границ не сработает. Эту проблему решает патч Ричарда Джонса (Richard Jones) и Пола Келли (Paul Kelly) для GCC. При использовании этого патча, с каждым указателем ассоциируется дескриптор массива, который содержит в себе информацию о границах данного массива. При использовании указателя делается ассоциативный запрос и находятся допустимые границы. Однако, такой способ дорого сказывается на производительности кода. В среднем, производительность падает в 10-30 раз.

### 3.4 Интерпретируемый код

Возможно, одной из причин столь большой популярности интерпретируемых языков программирования в последнее время, является как раз проблема с небезопасной работой с памятью в "классических"(транслируемых) языках программирования. Самыми известным интерпертируемыми языками программирования являются Java, Python, C# и тд. В основе защищенности этих языков лежит идея отделения программы непосредственно от памяти путем введения виртуальной машины. Так, в Java эту роль выполняет JVM (Java Virtual Machine). Программа, написанная на Java компилируется в промежуточный байт-код, который при запуске интерперетируется и исполняется виртуальной машиной. При таком способе исполнения, возможна проверка всех операций с памятью и, в случае выхода за границу массива приложение получает ArrayOutOfBounds Exception. Однако традиционно интерпертируемые языки проигрывают по производительности транслируемым.

## Список литературы

[1] *Aleph One.* Smashing The Stack For Fun And Profit `http://destroy.net/machines/security/P49-14-Aleph-One`

[2] *Sandeep Grover*. Buffer Overflow Attacks and Their Countermeasures `http://www.linuxjournal.com/article/6701`

[3] *Brad Spengler*.Documentation for the PaX project `http://pax.grsecurity.net/docs/`

[4] The OpenWall project `http://www.openwall.com/linux/`

[5] Stack Shield. `http://www.angelfire.com/sk/stackshield/index.html`

[6] StackGuard: Simple Stack Smash Protection for GCC `http://gcc.fyxm.net/summit/2003/Stackguard.pdf`