Внешний аудит безопасности корпоративных сетей

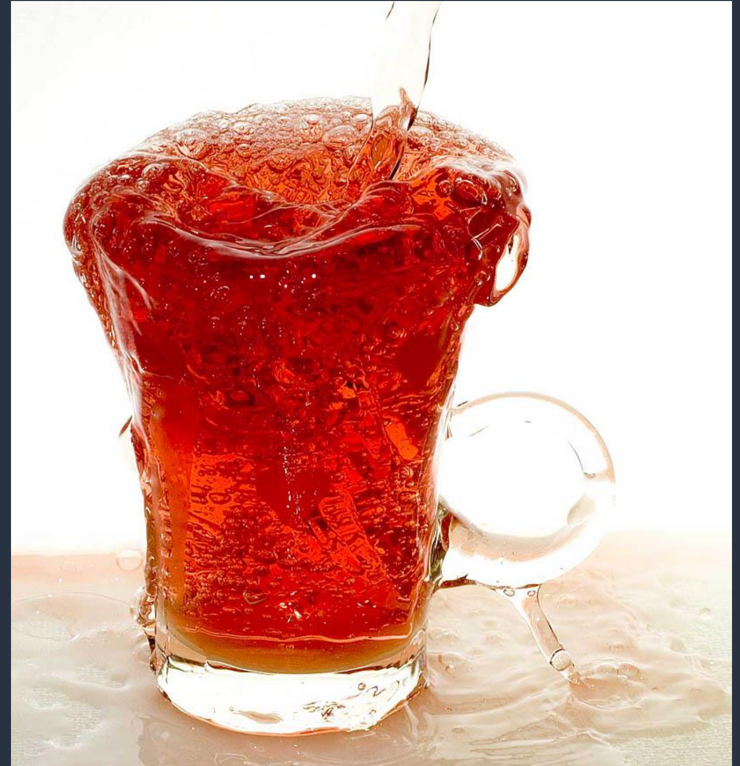
Лекция 8
Уязвимости памяти. Переполнения буфера

Семён Новосёлов

2021

Переполнения буфера

- **Переполнение стека**
- Переполнение кучи



Переполнение стека. Пример

```
#include <stdio.h>
#include <string.h>

void hello(char* str){
    char buf[128];
    strcpy(buf, str);
    printf("Hello, %s\n", buf);
}

int main(int argc, char* argv[]){
    if(argc < 2){
        printf("Enter argument!\n");
        return 1;
    }
    hello(argv[1]);
    return 0;
}
```

Код hello.c

- программа принимает аргумент извне и записывает в массив buf
- при этом в коде нет проверки выхода за границы массива
- локальные переменные хранятся в стеке
- перезаписывается память в стеке за массивом

Как выполняются программы?

Упрощенно:

1. ОС загружает код из исполняемого файла в память
2. выделяет память под стек и другие данные
3. передает управление на точку входа (Entry Point)

Распределение памяти для hello.c

Start Address	End Address	Permissions	Name
0x0000561d049f5000	0x0000561d049f8000	r-x	/media/sf_vbox-shared/bof/hello
0x0000561d049f8000	0x0000561d049fa000	rwx	/media/sf_vbox-shared/bof/hello
0x00007f771e575000	0x00007f771e5a1000	r-x	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f771e5a2000	0x00007f771e5a4000	rwx	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f771e5a4000	0x00007f771e5a5000	rwx	
0x00007fffed5ad000	0x00007fffed5ce000	rwx	[stack]
0x00007fffed5da000	0x00007fffed5dd000	r--	[vvar]
0x00007fffed5dd000	0x00007fffed5de000	r-x	[vdso]
0xffffffffff600000	0xffffffffff601000	--x	[vsyscall]

Edb: Memory Regions (Linux, x64)

Машинный код

- двоичный код программы: последовательность кодов команд (опкодов) процессора
- для удобства вместо чисел используется запись на языке ассемблера
- система команд зависит от архитектуры
  - Intel x64/x32: Intel® 64 and IA-32 Architectures Software Developer's Manual
  - Arm: Arm® A64 Instruction Set Architecture

Пример

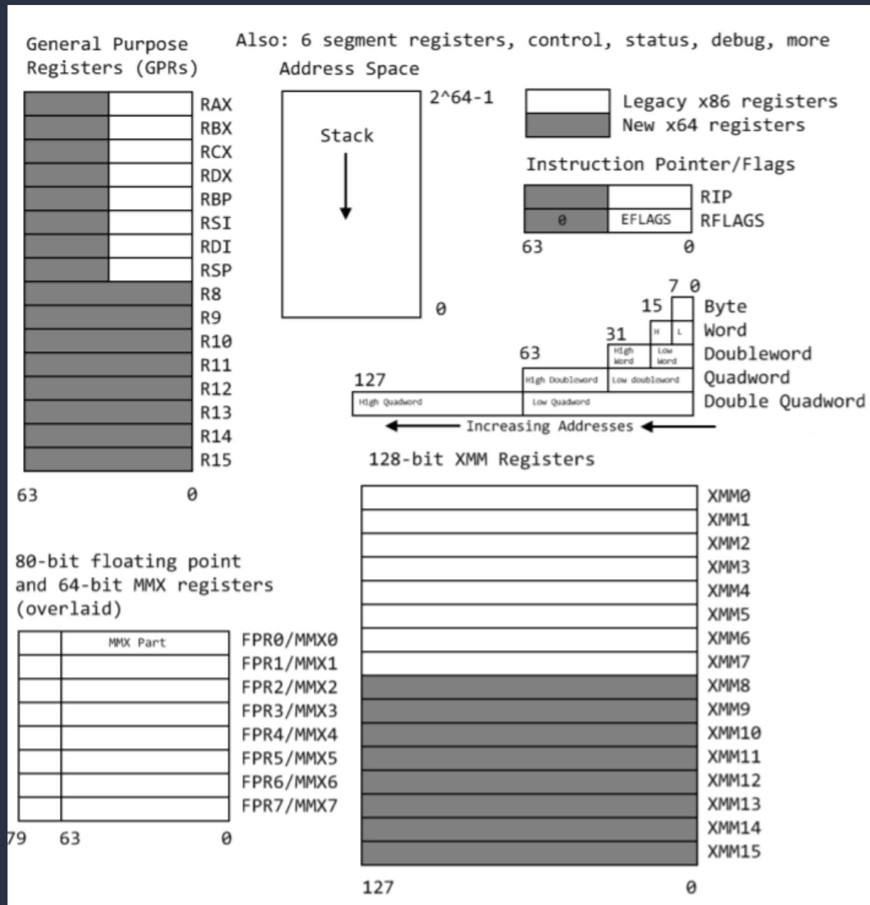
```
000055d6:ac1551d0  f3                     db 0xf3
000055d6:ac1551d1  0f                     db 0x0f
000055d6:ac1551d2  1e                     db 0x1e
000055d6:ac1551d3  fa                     cli
000055d6:ac1551d4  55                     push rbp
000055d6:ac1551d5  48 89 e5               mov rbp, rsp
000055d6:ac1551d8  48 83 ec 10            sub rsp, 0x10
000055d6:ac1551dc  89 7d fc               mov [rbp-4], edi
000055d6:ac1551df  48 89 75 f0            mov [rbp-0x10], rsi
000055d6:ac1551e3  83 7d fc 01            cmp dword [rbp-4], 1
000055d6:ac1551e7  7f 13                  jg 0x55d6ac1551fc
000055d6:ac1551e9  48 8d 3d 1f 0e 00 00   lea rdi, [rel 0x55d6ac15600f]   ASCII "Enter argument!"
000055d6:ac1551f0  e8 8b fe ff ff         call 0x55d6ac155080
000055d6:ac1551f5  b8 01 00 00 00         mov eax, 1
000055d6:ac1551fa  eb 18                  jmp 0x55d6ac155214
000055d6:ac1551fc  48 8b 45 f0            mov rax, [rbp-0x10]
000055d6:ac155200  48 83 c0 08            add rax, 8
000055d6:ac155204  48 8b 00               mov rax, [rax]
000055d6:ac155207  48 89 c7               mov rdi, rax
000055d6:ac15520a  e8 7a ff ff ff         call hello!hello
000055d6:ac15520f  b8 00 00 00 00         mov eax, 0
000055d6:ac155214  c9                     leave
000055d6:ac155215  c3                     ret
```

```c
int main(int argc, char* argv[]){
  if(argc < 2){
    printf("Enter argument!\n");
    return 1;
  }
  hello(argv[1]);
  return 0;
}
```

машинный код функции main

Архитектура IA-64

- регистры: наиболее быстрая память в процессоре
- как правило процессор сначала загружает данные в регистры, а затем обрабатывает
- используются для передачи параметров в функции
- RIP - указатель на текущую выполняемую инструкцию в программе
- RSP - указатель на вершину стека



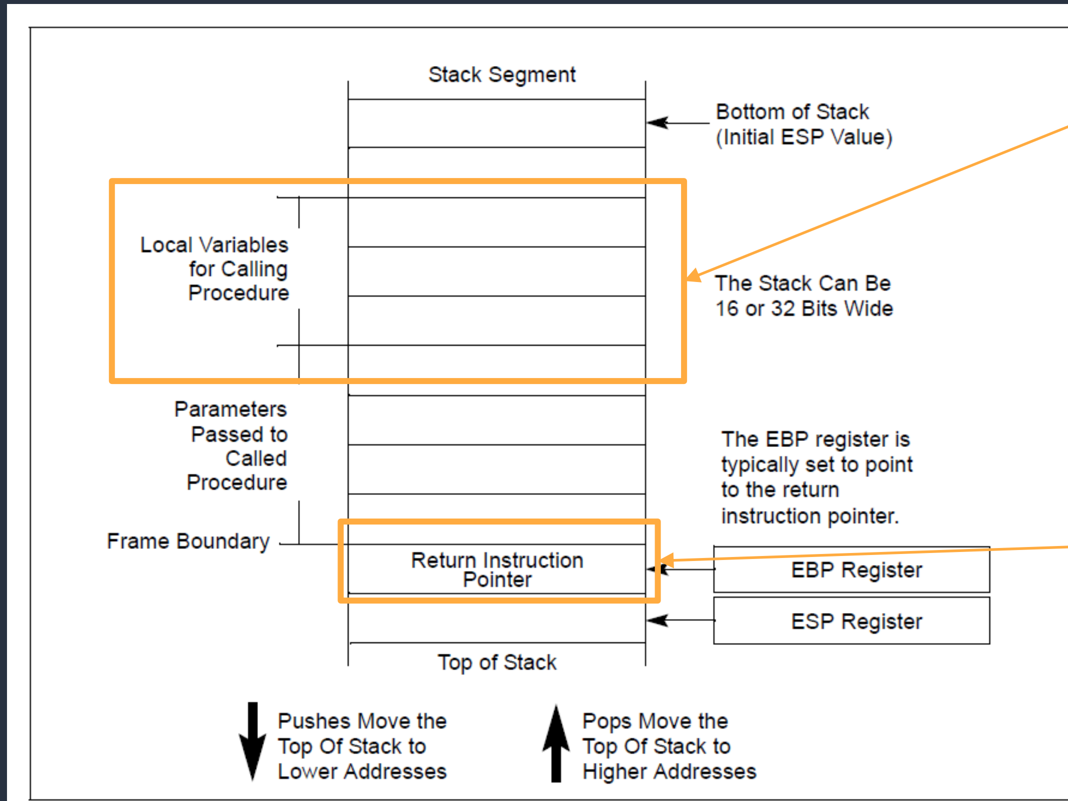
Структура стека

- Стек – это зарезервированная область памяти приложения, используемая для работы с динамическими данными
- Принцип LIFO – первым извлекается элемент, размещенный последним.
- Растет в сторону меньших адресов.
- Основные действия со стеком:
    - push: помещение элемента в стек
    - pop: извлечение элемента из стека
    - при выполнении команд процессор увеличивает/уменьшает регистр RSP

Инструкции call и ret

- call: вызов функции
  - помещает в стек по адресу в регистре rsp (то есть в вершину стека), текущее значение регистра rip (счетчика команд) - адрес возврата
  - переходит по адресу, указанному в качестве операнда
- ret: возврат из функции:
  - производит обратные действия к call
  - извлекает из стека значение по адресу в регистре rsp и переходит по нему

Структура стека



здесь хранится буфер buf из hello() в hello.c

помещается в стек при вызове hello() с помощью call

указатель на main(), перезаписывается внешними данными при переполнении буфера buf в hello.c

Можно подставить свой адрес и при вызове **ret** программа перейдет по нему

Эксплуатация

Предполагаем в начале, что нет защитных механизмов.

1. записать рабочий машинный код (шелкод) в буфер
   - база: https://www.exploit-db.com/shellcodes
2. перезаписать адрес возврата указав на адрес начала кода в буфере
3. адреса можно узнать с помощью отладчика

Инструменты отладки

- EDB - графический отладчик, удобно использовать для пошаговой отладки
- GDB - для анализа coredump

Пример. Вывод /etc/passwd через hello.c

1. Возьмём шелкод для вывода /etc/shadow: https://www.exploit-db.com/shellcodes/49547
   и заменим shadow на passwd, чтобы не нужны были права root.
2. Подбираем размер буфера (buf_size):
   должен перезаписываться только адрес возврата на main()
3. Формируем буфер buf вида:
   [shellcode][A x (buf_size - shellcode_size - 8)][buff_addr]
   - A x (buf_size - shellcode_size - 8) - можно заменить на любые данные, нужного размера
4. Передаём на вход программы hello “[buf]”

Как узнать адрес буфера?

- Адрес плавает в зависимости от среды выполнения (переменных окружения и т.п.)
- При первоначальной отладке:
  - посмотреть в отладчике
- При запуске без отладчика:
  - включить coredump (ulimit -c unlimited в Linux)
  - запустить для генерации coredump
  - загрузить в gdb программу вместе с coredump:
    gdb ./hello core
    отладчик выведет адрес addr, на котором программа завершилась
  - найти в памяти возле addr начало шелкода (команда x/100x-20 addr в gdb) и подставить его адрес в буфер

NOP-дорожка

- для снижения точности указания адреса можно поставить в начало шелкода команды NOP с кодом \x90.
- при переходе на такую команду процессор просто переходит к следующей
- пример:
  [NOP x 100][shellcode][...]
  можно сделать ошибку в адресе на 100

Важно: необходимо следить, чтобы шелкод не перезаписывался при выполнении программы, он лежит в стеке, а стек постоянно меняется

Защита от переполнения буфера

Переполнение буфера — архитектурная проблема, связанная с хранением кода и данных в одном адресном пространстве.

Невозможно эффективно предотвратить, но можно ограничить последствия, сделав атаку нерабочей:

- DEP/NX-бит
- ASLR
- Stack-Canary

Защита: Запрет выполнения стека

- Делает стек не выполняемым
- На аппаратном уровне реализована с помощью атрибута страницы памяти (NX/XD – бита)
- На программном уровне поддерживается начиная с ядра Linux версии 2.3.23 и Windows XP SP1 (DEP)
- На 64-битных системах первые параметры передаются через регистры.
- Поэтому атака возврата в библиотеку в чистом виде затруднительна.

Атака: Возврат в библиотеку (Return-to-libc attack)

- Так как стек стал неисполняемым, то передавать управление на него смысла нет.
- Всё ещё можно:
    - передать управление на любой исполняемый участок кода программы
    - вызвать одну из функций любой подключенной библиотеки
- При этом необходимо поместить аргументы этой функции в буфер.

Атака: ROP (Return oriented programming)

- Для атаки используются части кода, которые заканчиваются инструкцией ret (“гаджеты”).
- Формируется цепочка из гаджетов и данных для них, которая выполняет нужный код.
- Пример: нужно записать в регистр eax определённое значение.
  - Находим в коде: pop eax; ret
  - Помещаем в буфер:
    адрес возврата указывающий на pop eax; ret +
    нужное значение +
    адрес следующего кусочка кода

Защита: ASLR (Address space layout randomization)

- Расположение важных структур (стека, кучи и библиотек) по случайным адресам.
- Так как адреса становятся случайными, то нельзя построить цепочку гаджетов или выбрать адрес для возврата в библиотеку.

ASLR. Обход

- В некоторых системах технология реализована не полностью.
- Перебор адресов.
- Использование уязвимостей, позволяющих читать содержимое памяти.
- В многопоточных приложениях адреса не меняются при создании/удалении потоков

Защита: Stack Canary

- Вставляет определённое значение после буфера и проверяет его перед выходом из функции.
- Есть несколько видов защиты:
  - Terminator canaries
  - Random canaries
  - Random XOR canaries
- gcc использует комбинированный подход:
  - 0x00 + 3 случайных байта

Обход Stack canary

- В некоторых случаях возможен перебор: многопоточные сервера и приложения
- Использование уязвимостей, позволяющих читать память в стеке

Защита. Control-Flow Enforcement Technology (CET)

- работает на уровне процессора
- вводит дополнительный “теневой стек”
- при вызове call в него сохраняется состояние вызывающей процедуры
- при вызове ret производится проверка защищает от перезаписи адреса возврата

Процессоры: Intel Tiger Lake

Литература и ссылки

- Google Address/Memory Sanitizer
  - https://github.com/google/sanitizers/wiki/MemorySanitizer
  - https://github.com/google/sanitizers/wiki/AddressSanitizer
- CWE-121: Stack-based Buffer Overflow
  - https://cwe.mitre.org/data/definitions/121.html
- EDB Debugger
  - https://github.com/eteran/edb-debugger
- Д. Фостер, В. Лю. Разработка средств безопасности и эксплойтов. 2011