Низкоуровневое программирование

Лекция 6

Уязвимости форматной строки
Переполнение на стеке
Переполнение в куче
ASLR

Буферы

Для реализации ввода-вывода часто используются буферы – массивы, используемые для хранения временных данных.

Буферы малого размера часто располагаются *на стеке*, рядом с локальными переменными и адресами возврата.

В языке C++ работа с вводом/выводом во многом автоматизирована внутри классов стандартной библиотеки.

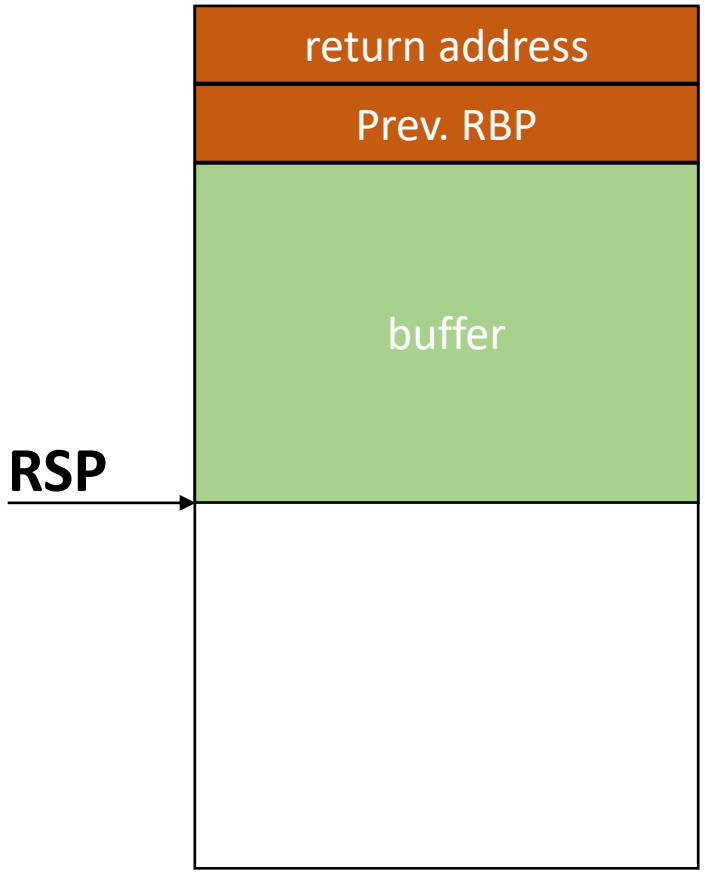
В языке C программист сам должен следить за соответствием вводимых данных и размера буфера.

Многие стандартные функции по умолчанию не производят проверку границ буфера.

Уязвимости форматной строки

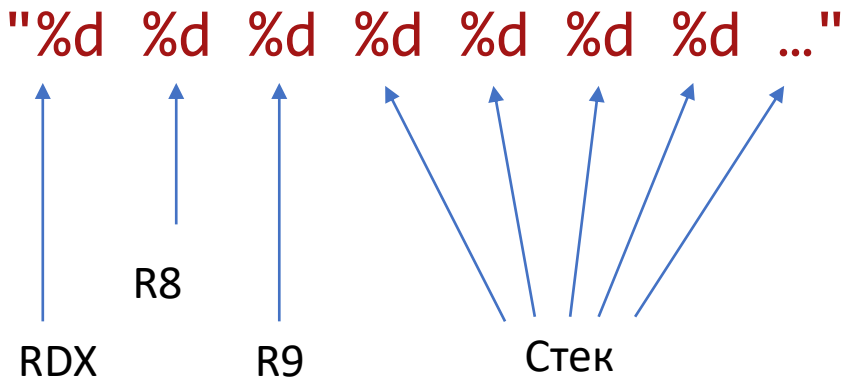
```
void echo() {
    char buffer[100] ;
    scanf("%s", buffer);
    printf(buffer);
}

int main() {
    echo();
}
```

Что будет, если ввести “%d %d %d %d %d %d %d”?

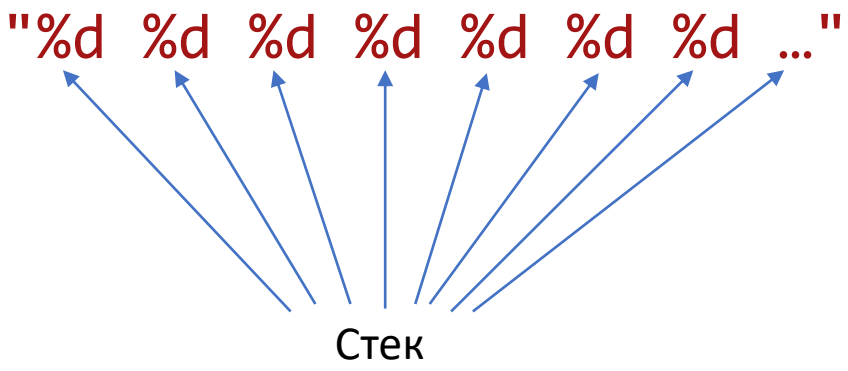


Уязвимости форматной строки

Microsoft x64:



cdecl:

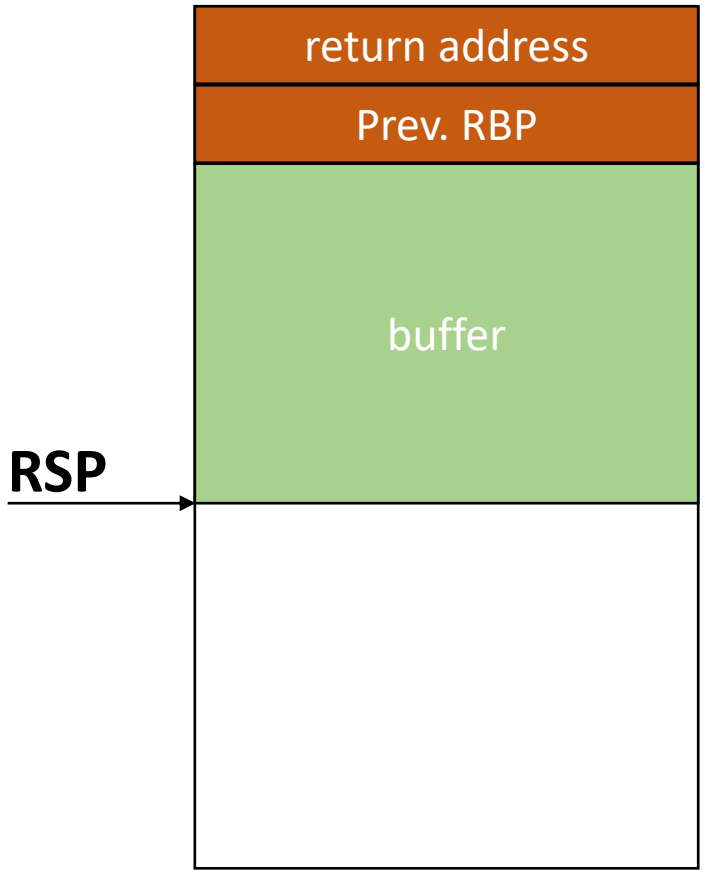



```
void echo() {
    char buffer[100] ;
    scanf("%s", buffer);
    printf(buffer);
}

int main() {
    echo();
}
```

Уязвимость форматной строки дает возможность читать стек, предоставляя информацию атакующему

Уязвимости форматной строки

```
void echo() {
    char buffer[100]{} ; //zeros
    scanf("%s", buffer);
    printf(buffer);
}

int global_variable = 0;

int main() {
    echo();
    if (global_variable)
        printf("Access granted");
}
```



Как можно поменять global_variable?

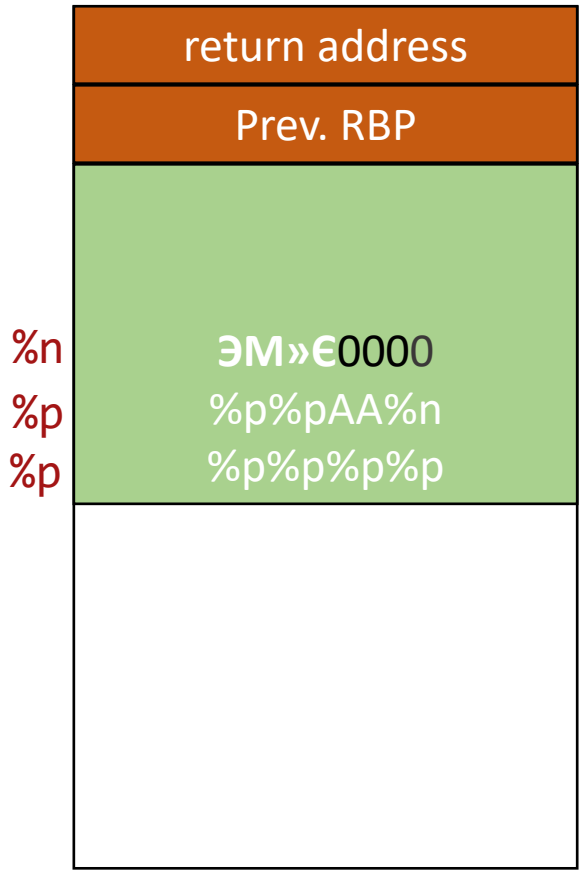
Уязвимости форматной строки

%x	Чтение 4 байт
%llx	Чтение 8 байт
%p	Чтение указателя (8 байт на x86-64)
%n	**Запись числа напечатанных символов по указателю int***
%lln	**Запись числа напечатанных символов по указателю long long***

```
void echo() {
    char buffer[100]{} ; //zeros
    scanf("%s", buffer);
    printf(buffer);
}

int global_variable = 0;

int main() {
    echo();
    if (global_variable)
        printf("Access granted");
}
```

Уязвимости форматной строки

```
void echo() {
    char buffer[100]{} ; //zeros
    scanf("%s", buffer); //%p%p%p%p%p%pAA%nЭМ»Є
    printf(buffer);
}

int global_variable = 0;     //0x00000000AABBCCDD

int main() {
    echo();
    if (global_variable)
        printf("Access granted");
}
```



Защита от уязвимостей форматной строки

```
void echo() {
    char buffer[100] ;
    scanf("%s", buffer);
    printf(buffer);
}

int main() {
    echo();
}
```

Есть уязвимость

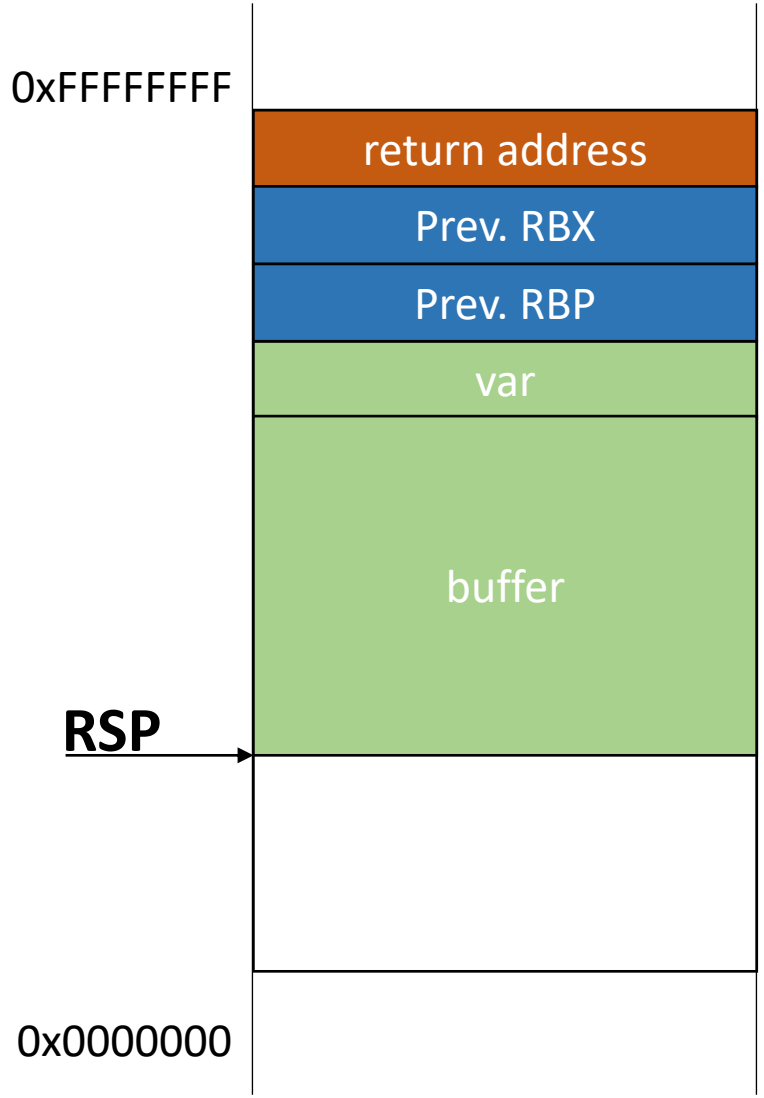
```
void echo() {
    char buffer[100] ;
    scanf("%s", buffer);
    puts(buffer);
}

int main() {
    echo();
}
```

Нет уязвимости

Переполнение буфера

```c
#include <stdio.h>

int main()
{
    int var = 0;
    char buffer[16];
    scanf("%s", buffer);
}
```

Что будет, если ввести строку длиной больше 15?



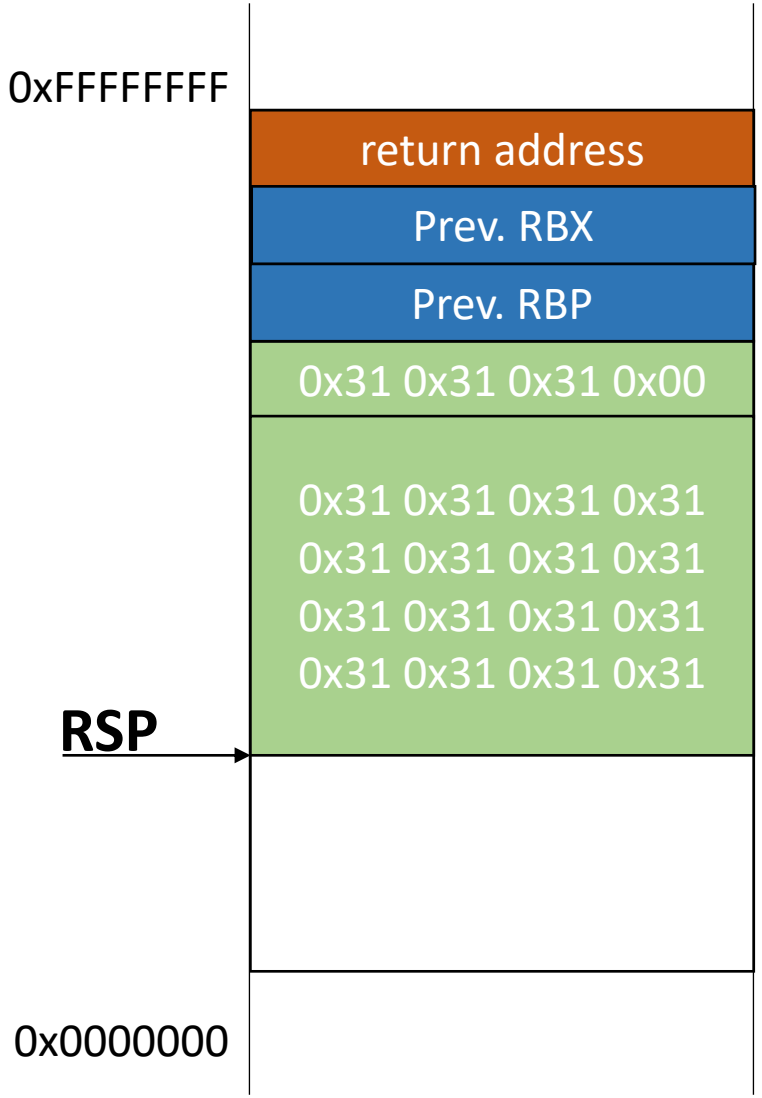
Переполнение буфера

```c
#include <stdio.h>

int main()
{
    int var = 0;
    char buffer[16];
    scanf("%s", buffer);
}
```

Ввод: 1111111111111111

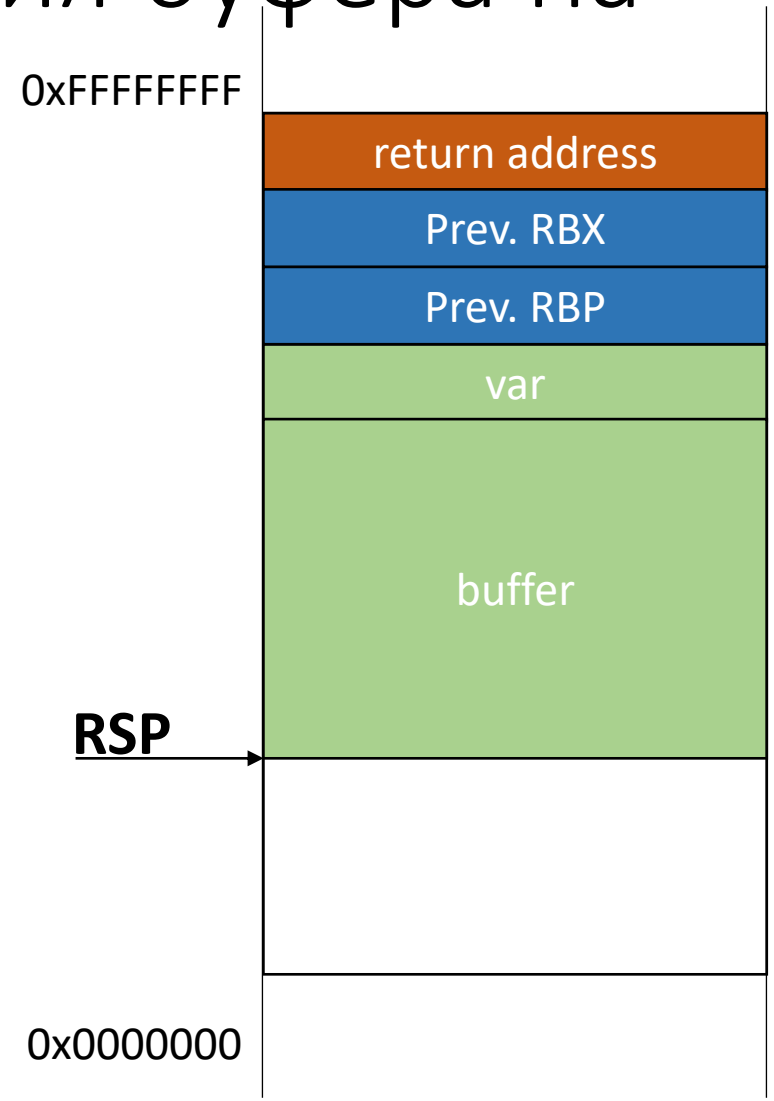
Аналогичные атаки возможны с любой функцией, которая не проверяет размер буфера записи: gets, strcpy, memcpy, memmove,…

Мера защиты: функции с проверкой границ буфера - fgets, scanf_s и пр.



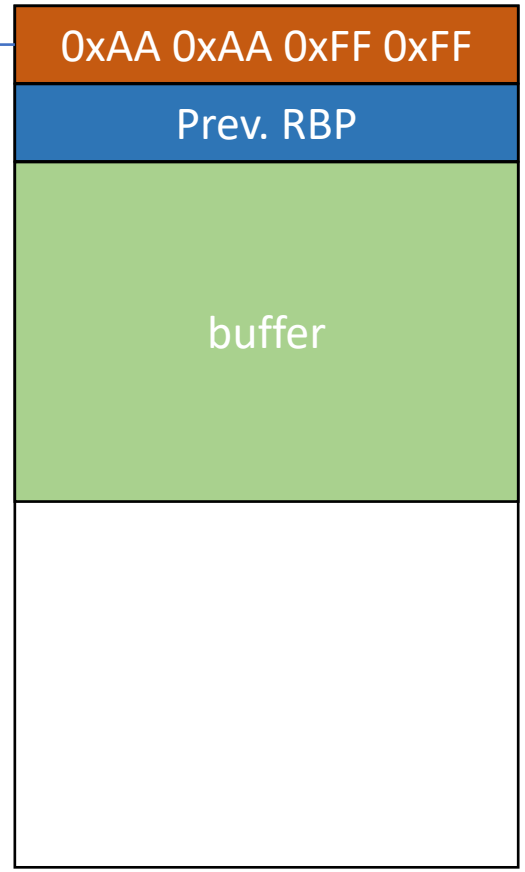
Возможные цели переполнения буфера на стеке

- **Локальные переменные**: перезапись позволяет изменить ход работы текущей функции;
- **Копии регистров на стеке**: чаще всего хранят некоторые переменные из вызывающей функции => перезапись может изменить ход работы вызывающей функции;
- **Адрес возврата.**



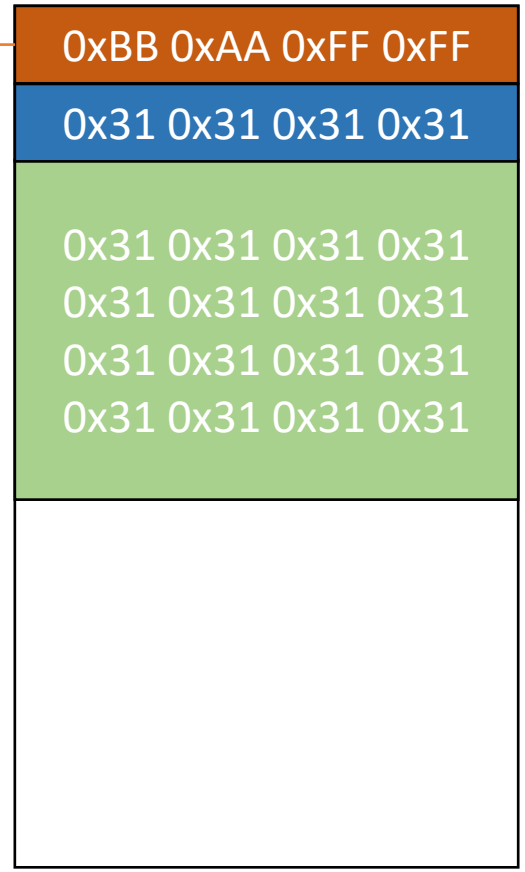
Изменение адреса возврата

```
bool pass_check() {
    char buffer[64];
    scanf("%s", buffer);
    return !strcmp(buffer, "PASSWORD");
}

int main() {
    bool ok = pass_check();
    if (!ok)
        return -1;
   /*ACCESS DATA*/
}
```



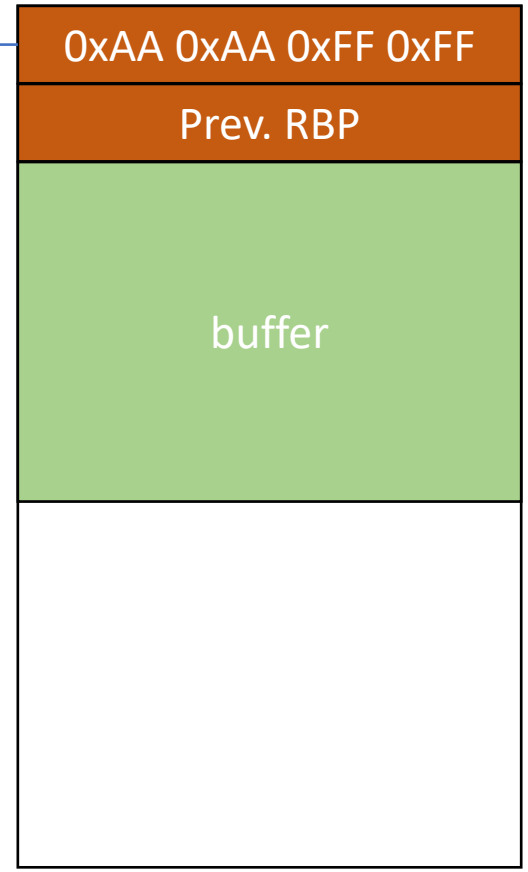
Изменение адреса возврата

```
bool pass_check() {
    char buffer[64];
    scanf("%s", buffer);
    return !strcmp(buffer, "PASSWORD");
}

int main() {
    bool ok = pass_check();
    if (!ok)
        return -1;
   /*ACCESS DATA*/
}
```



Исполнение произвольного кода

```
bool pass_check() {
    char buffer[64];
    scanf("%s", buffer);
    return !strcmp(buffer, "PASSWORD");
}

int main() {
    bool ok = pass_check();
    if (!ok)
        return -1;
   /*ACCESS DATA*/
}
```



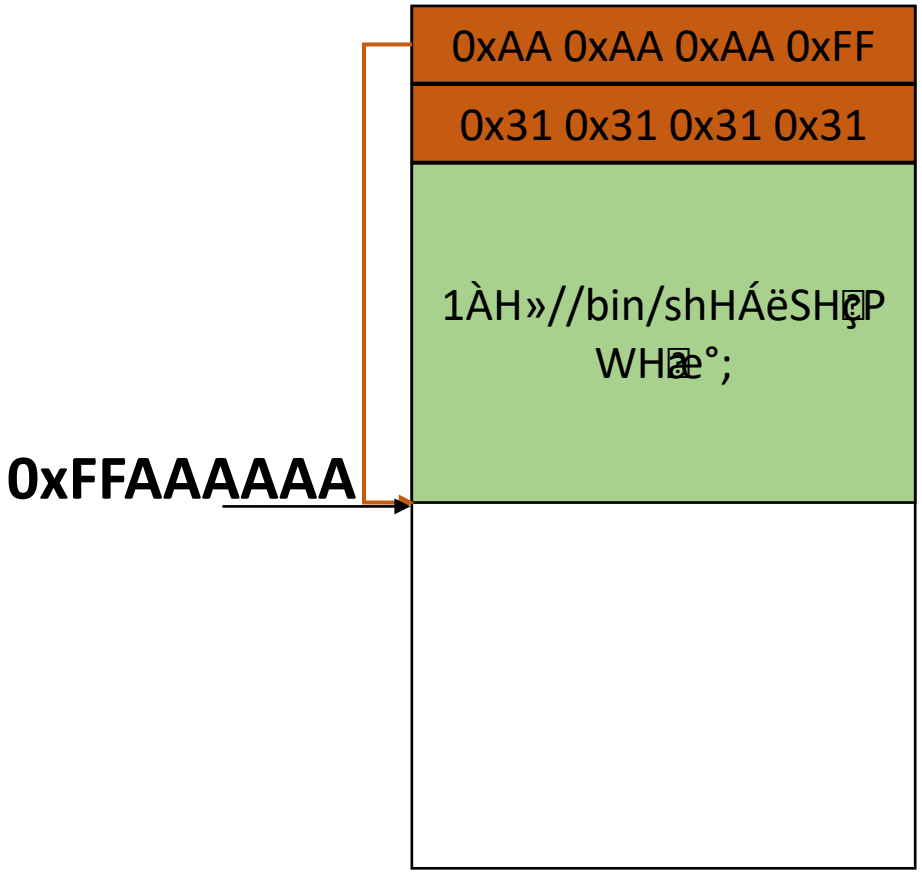
Исполнение произвольного кода

```
xor     eax, eax                    31 c0
mov     rbx, '//bin/sh'             48 bb 2f 2f 62 69 6e 2f 73 68
shr     rbx, 8                      48 c1 eb 08
push    rbx                         53
mov     rdi, rsp                    48 89 e7
push    rax                         50
push    rdi                         57
mov     rsi, rsp                    48 89 e6
mov     al, 0x3b                    b0 3b
syscall                             0f 05
```

1ÀH»//bin/shHÁëSHçPWHæ°;

Исполнение произвольного кода

```
bool pass_check() {
    char buffer[64];
    scanf("%s", buffer);
    return !strcmp(buffer, "PASSWORD");
}

int main() {
    bool ok = pass_check();
    if (!ok)
        return -1;
   /*ACCESS DATA*/
}
```



Об аргументах и переменных среды (Linux)

В ОС на ядре Linux аргументы и переменные среды располагаются в начальной области стека – *даже если программа их не использует*.

Как следствие, аргументы программы и переменные среды могут использоваться, как вспомогательный канал ввода информации, которая используется при успешной эксплуатации других уязвимостей. Например, если буфер слишком мал для шелл-кода, его можно ввести с помощью аргументов программы.

Аргументы удобны тем, что позволяют передавать в программу нулевые байты (пустая строка в качестве аргумента = байт 0). При обычном текстовом вводе-выводе передать в программу последовательность нулевых байтов не всегда возможно, что порождает проблему (например, если вводимый адрес содержит 0, что почти всегда верно для x86-64).

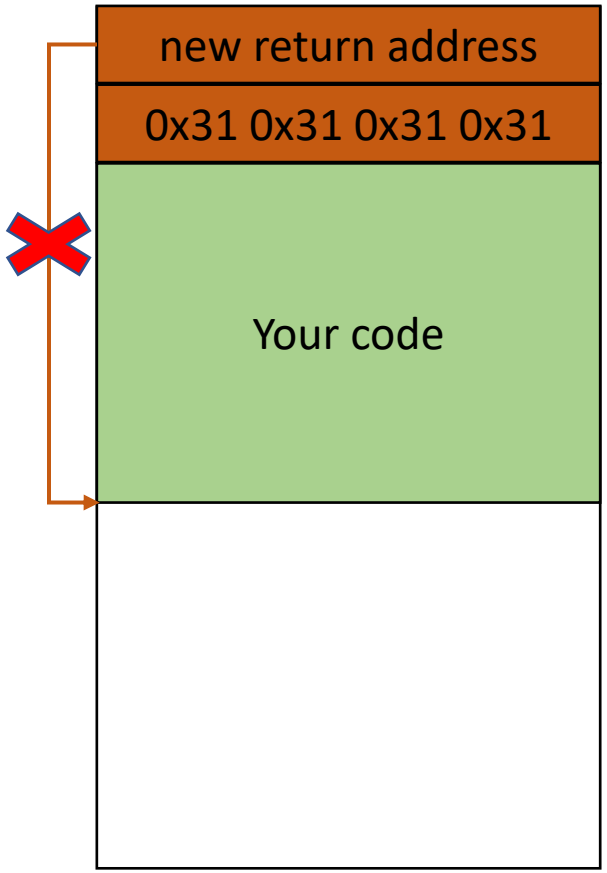
Обратной стороной такого подхода является смещение адресов элементов стека при изменении аргументов/переменных среды, что неудобно для атакующего.

Защита от выполнения

В современных процессорах возможно запрещать исполнение кода в определённых сегментах памяти.

При попытке исполнения кода внутри сегментов .stack, .heap, .data, .rodata будет сгенерировано аппаратное исключение и *программа завершится*.

Можно сделать стек исполняемым – но пользователь должен явно запросить это при сборке программы.



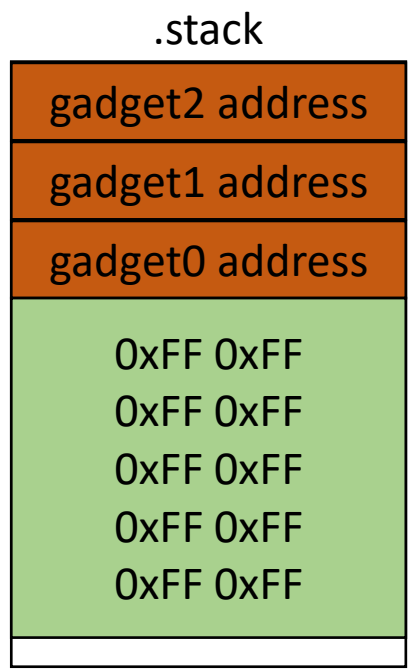
Return Oriented Programming

При включенном запрете исполнения нельзя просто передать управление на записанный код, но адрес возврата изменять все еще можно. Тогда в исполняемых файлах и загружаемых библиотеках можно найти конструкции следующего вида:

```
<часть нужных инструкций>
ret
```

Таки конструкции называются **гаджетами** (gadgets). Если гаджетов достаточно для составления требуемого кода, они могут быть объединены в **ROP-цепочку**.

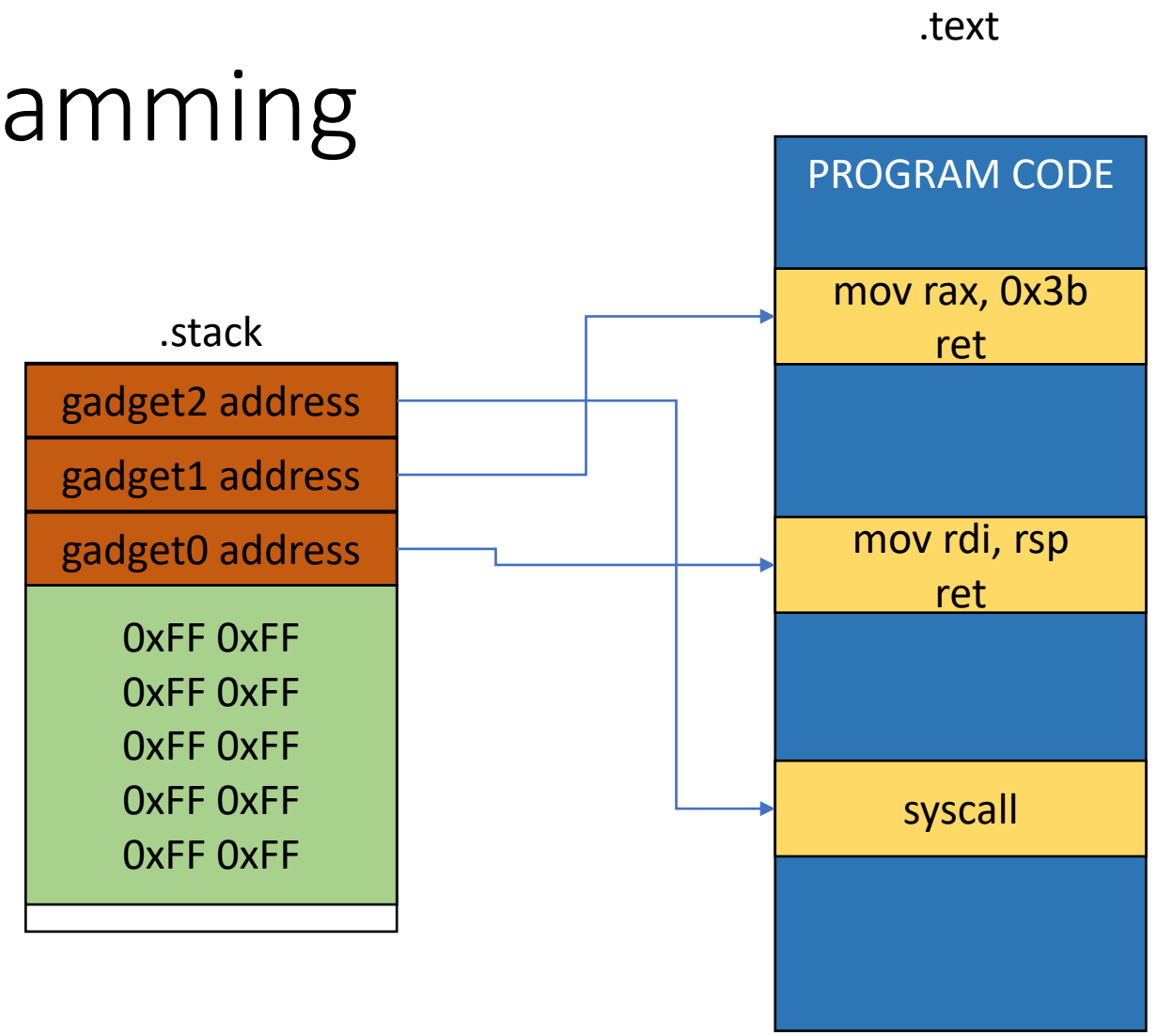
Примечание: помимо `ret` могут использоваться инструкции перехода по вычисляемому аргументу, например `jmp [eax]` (Jump Oriented Programming) или `call [eax]` (Call Oriented Programming) - в этом случае аргумент должен содержать целевой адрес.



Return Oriented Programming

При эксплуатации ROP-цепочки нарушитель должен сформировать на стеке адреса возврата, первый из которых указывает на начальный гаджет.

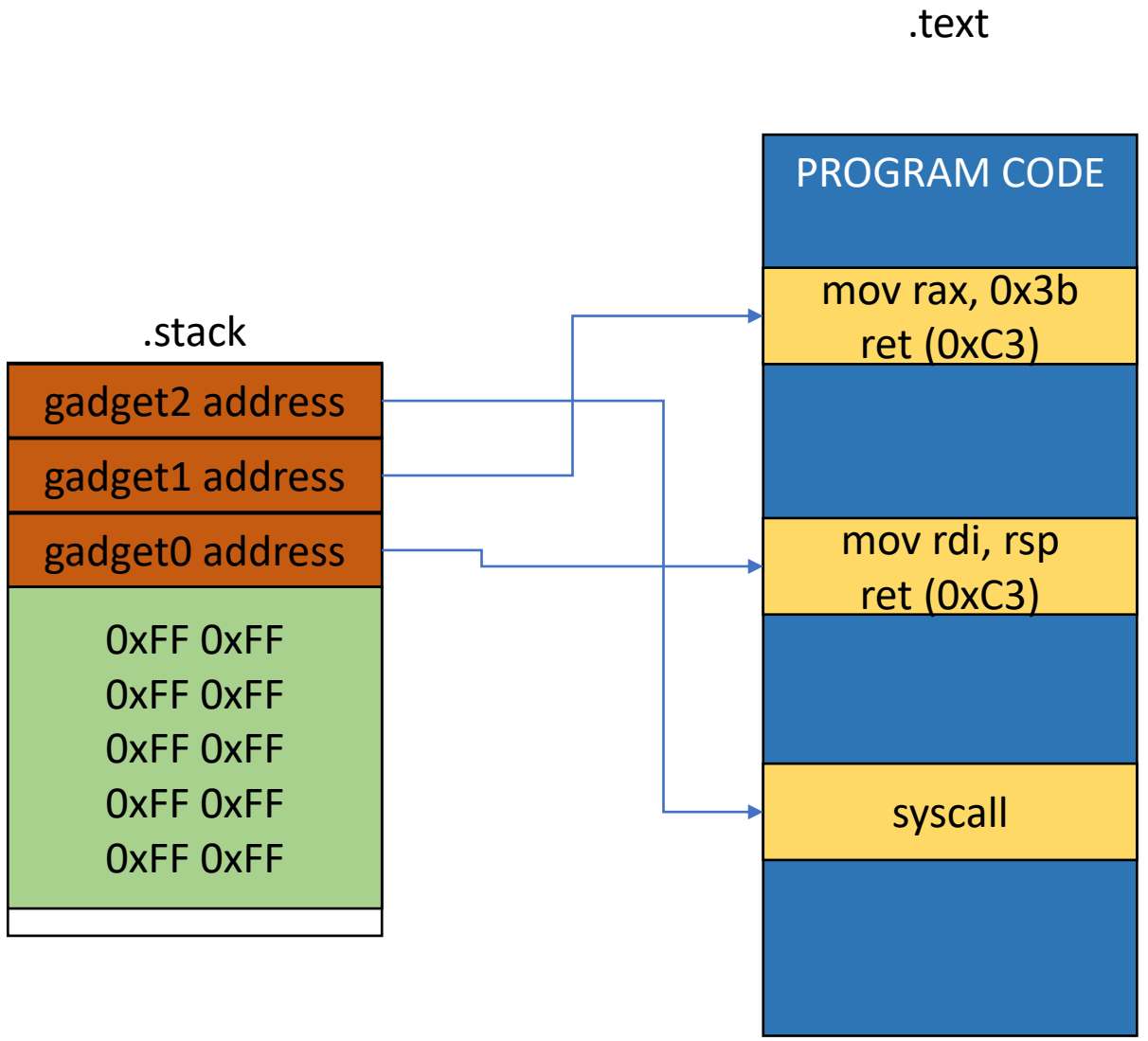
Переход от гаджета к гаджету осуществляется инструкцией ret, которая достает адрес следующего гаджета со стека.



Поиск гаджетов

Гаджеты обычно ищутся специально написанным ПО.

Маркером гаджета является инструкция ret, имеющая код 0xC3. Если байт с данным значением найден, просматриваются байты слева от него. Если они представляют собой нужные инструкции, гаджет запоминается, иначе – поиск идет дальше.



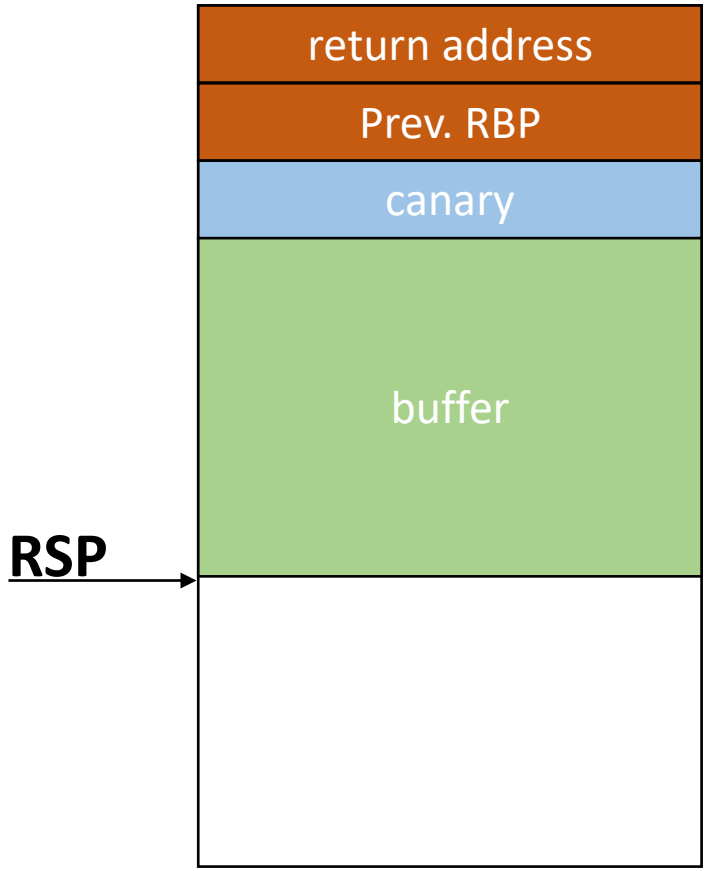
Защита от изменения адреса возврата

Для защиты стека от переполнения буфера и изменения адреса возврата используются т.н. «**канарейки**» (canary, stack protector).

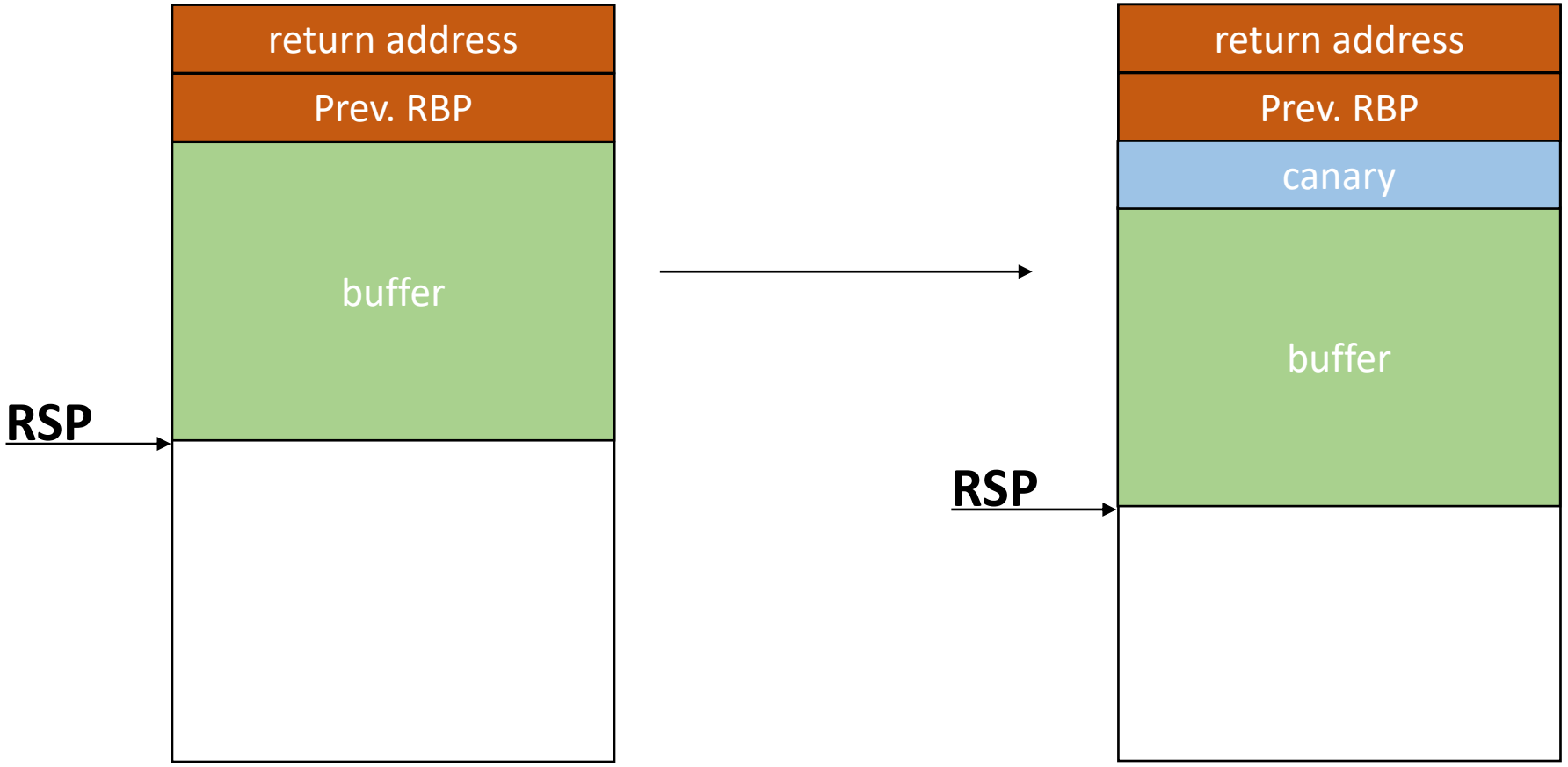
«Канарейка» - это копия значения специальной глобальной переменной, которая записывается на стек при входе в функцию. При выходе из функции значение канарейки сверяется с исходным. Если значения не совпали – возникает исключение.

Поскольку исходня глобальная переменная находится не на стеке, атакующий не может на нее повлиять с помощью переполнения буфера на стеке.

Канарейки могут быть *отключены* специальными ключами компиляции (-fno-stack-protector для GCC)

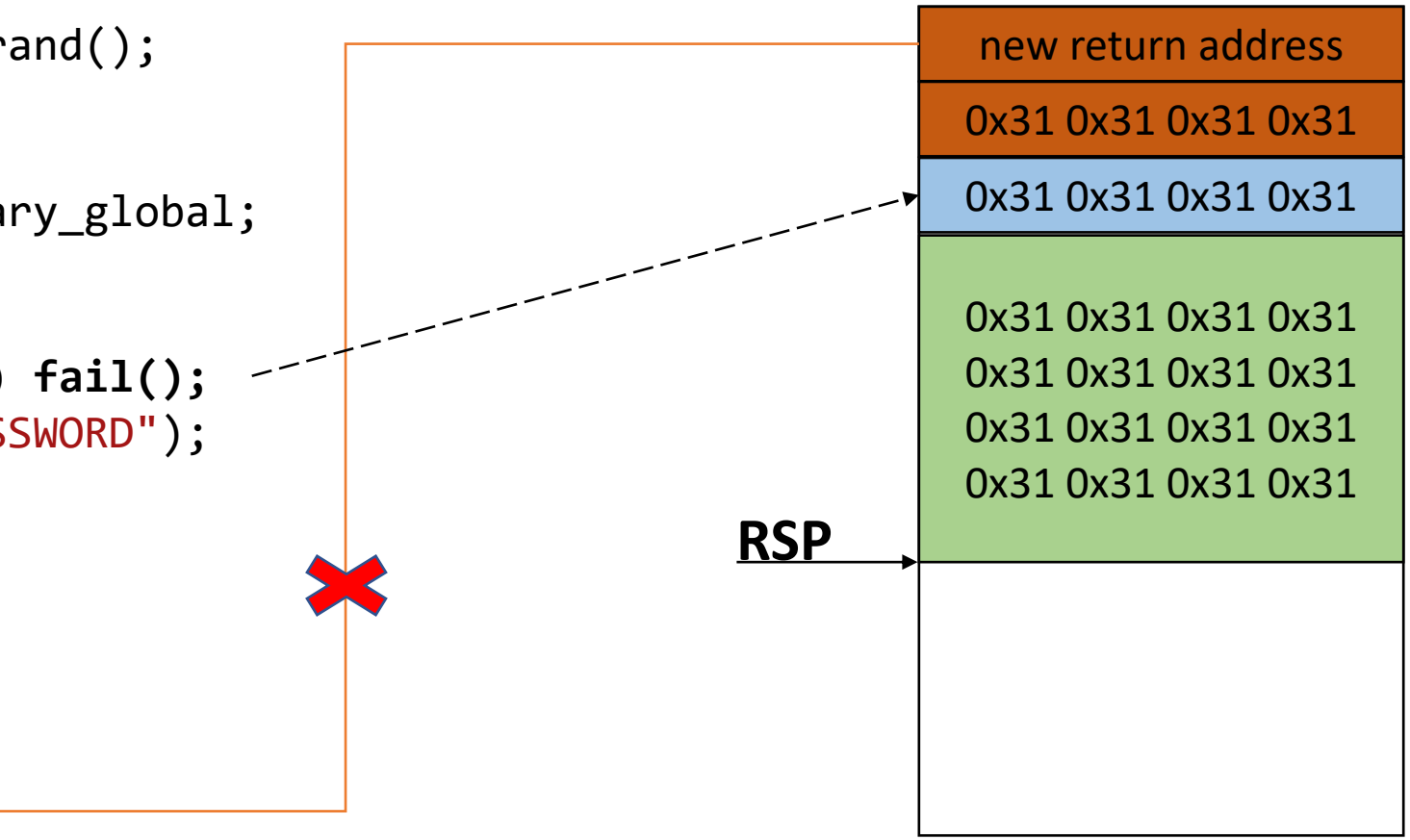


Защита от изменения адреса возврата



Защита от изменения адреса возврата

```
long long int canary_global = rand();

bool pass_check() {
    long long int canary = canary_global;
    char buffer[64];
    scanf("%s", buffer);
    if(canary != canary_global) fail();
    return !strcmp(buffer, "PASSWORD");
}

int main() {
    bool ok = pass_check();
    if (!ok)
        return -1;
   /*ACCESS DATA*/
}
```



Intel Indirect Branch Tracking

Технология Intel IBT позволяет усложнить атакующему эксплуатацию уязвимостей, основанных на изменении адреса возврата.

ROP-атака основана на возможности атакующего выбирать любую точку в программе для последующего перехода.

Intel IBT вводит 2 новые инструкции `endbr32/enbr64` (END Branch) для 32/64-битных программ. Кодировка инструкций подобрана так, что ЦП без поддержки (или с выключенным) IBT воспринимает их как `nop`. Эти инструкции используются для маркировки валидных точек перехода.

Если ЦП выполняет переход (`call/jmp/j*/ret`), и он попадает не на `endbr`, то генерируется аппаратное исключение и программа аварийно завершается => ограничивается выбор для атакующего.

Shadow Stack

Shadow Stack – техника, позволяющая защититься от перезаписи адреса возврата, поддерживаемая ЦП Intel и AMD.

При выполнении инструкции `call` адрес возврата сохраняется в обычный стек, и в специальный теневой стек.

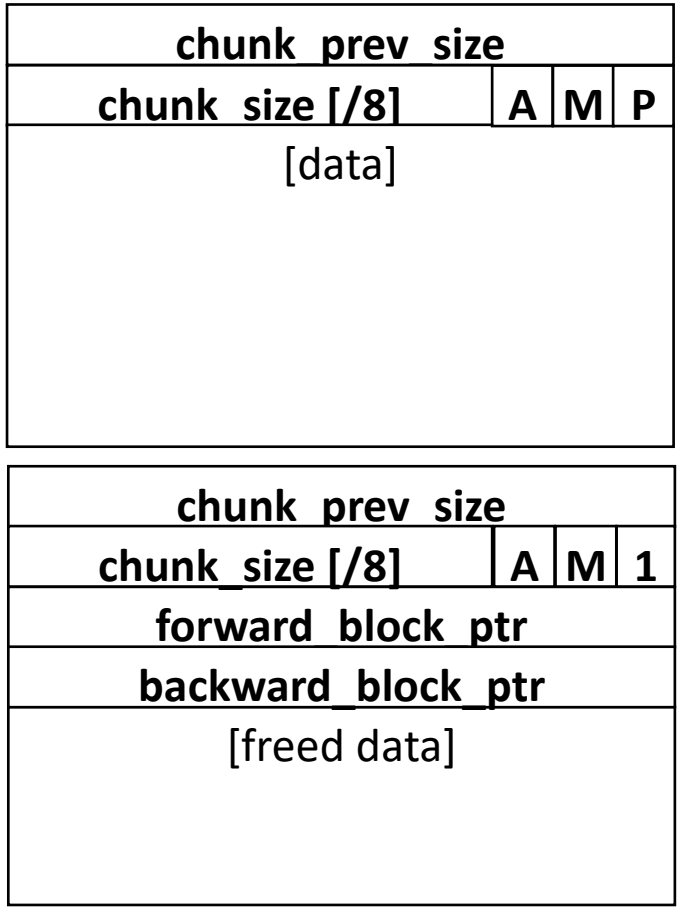
При выполнении инструкции `ret` записи из обычного и из теневого стека сравниваются. Если есть различие – генерируется аппаратное исключение и программа аварийно завершается.

Структура кучи

Структура кучи, вообще говоря, зависит от ОС и от реализации стандартной библиотеки языка.

В языке C за управление кучей отвечают функции malloc/free. В libc исходные реализации этих функций написаны Дугласом Ли.

В общих чертах, структура кучи является двусвязным списком свободных блоков памяти.



Структура кучи

Структура блока памяти выглядит следующим образом:

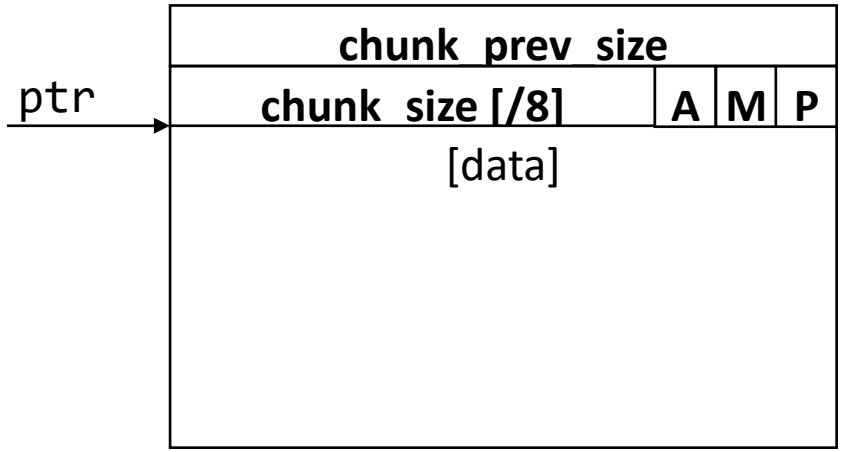
Поле **chunk_prev_size** равно размеру предыдущего блока памяти.

Поле **chunk_size** равно размеру текущего блока. Т.к. размер блока всегда кратен 16 (для x64), последние 3 бита всегда равны 0. Эти 3 бита используются как флаги.

Флаг **P** отвечает за состояние предыдущего блока памяти (0 если свободен, 1 если выделен).

Malloc() всегда возвращает адрес начала данных.

```
void* ptr = malloc(x);
```

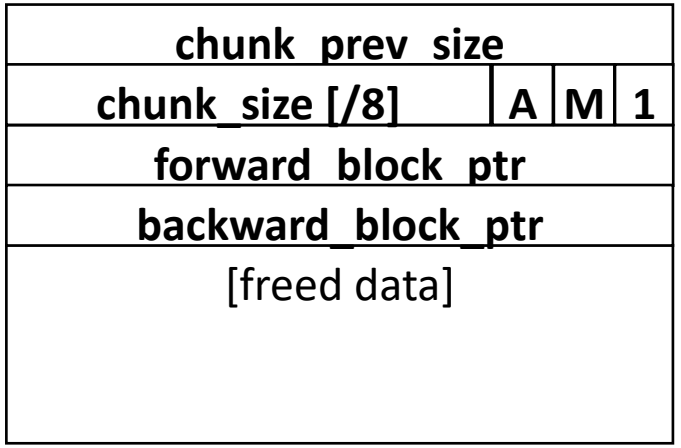


Свободные блоки

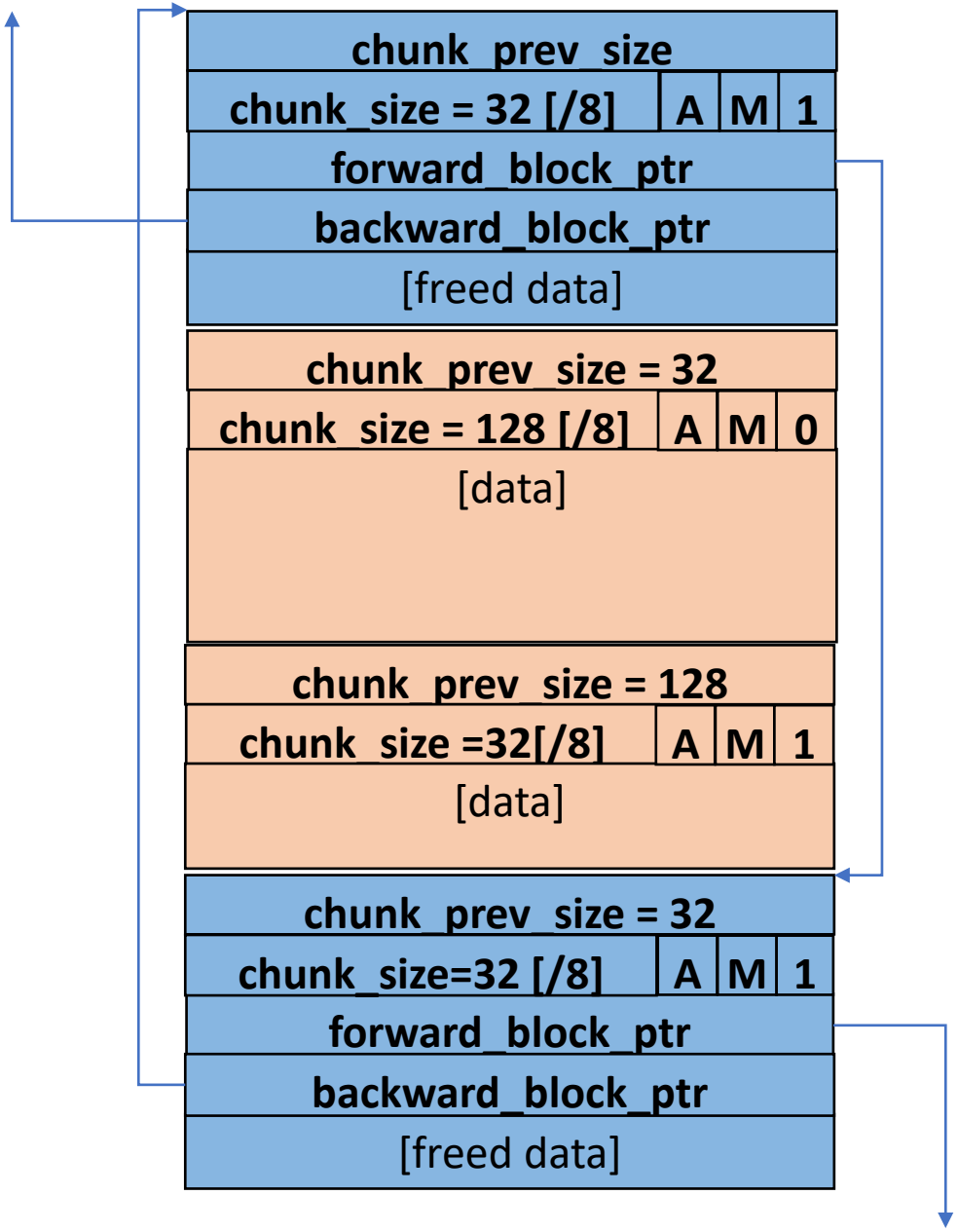
Если блок данных свободен, то в части бывших данных располагаются 2 поля: указатель на предыдущий свободный блок (**forward_block_ptr**) и следующий свободный блок (**backward_block_ptr**) .

Функция free() заполняет эти поля при освобождении блока.

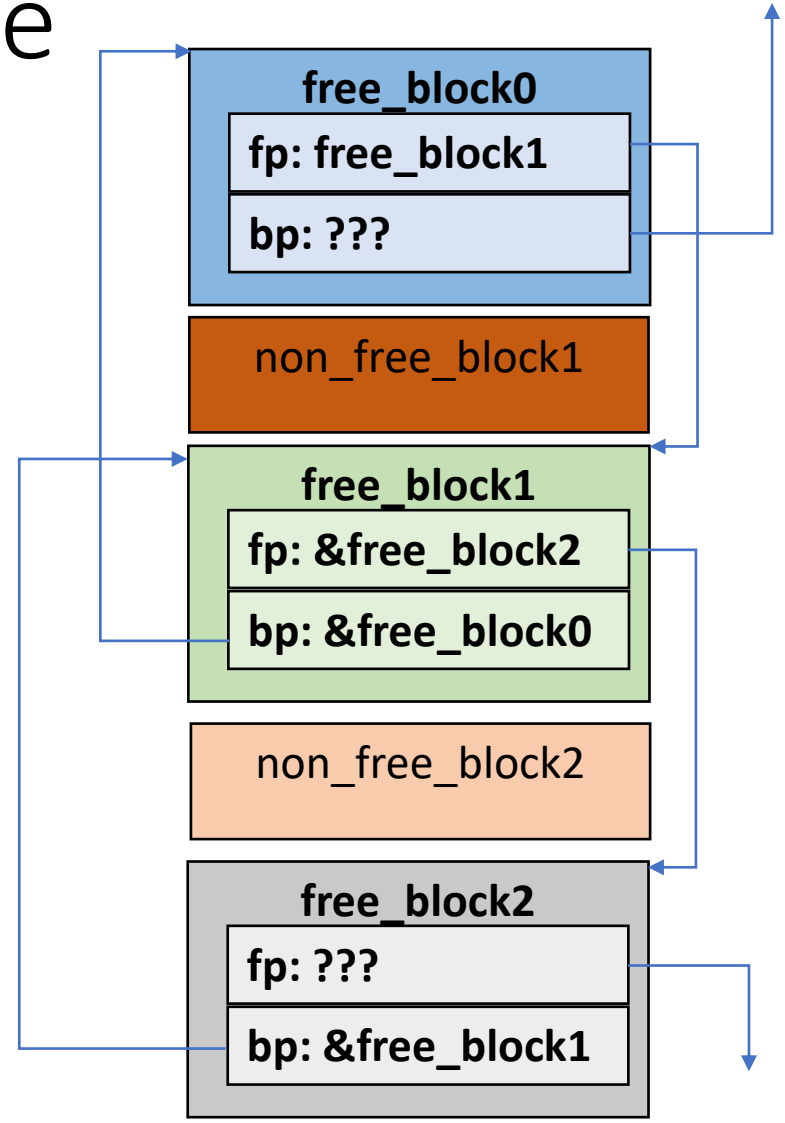
Если освобождается блок данных, перед которым уже есть свободный блок, они объединяются в один => перед свободным блоком и после свободного блока не может находиться другой свободный блок => флаг P=1.



Структура кучи

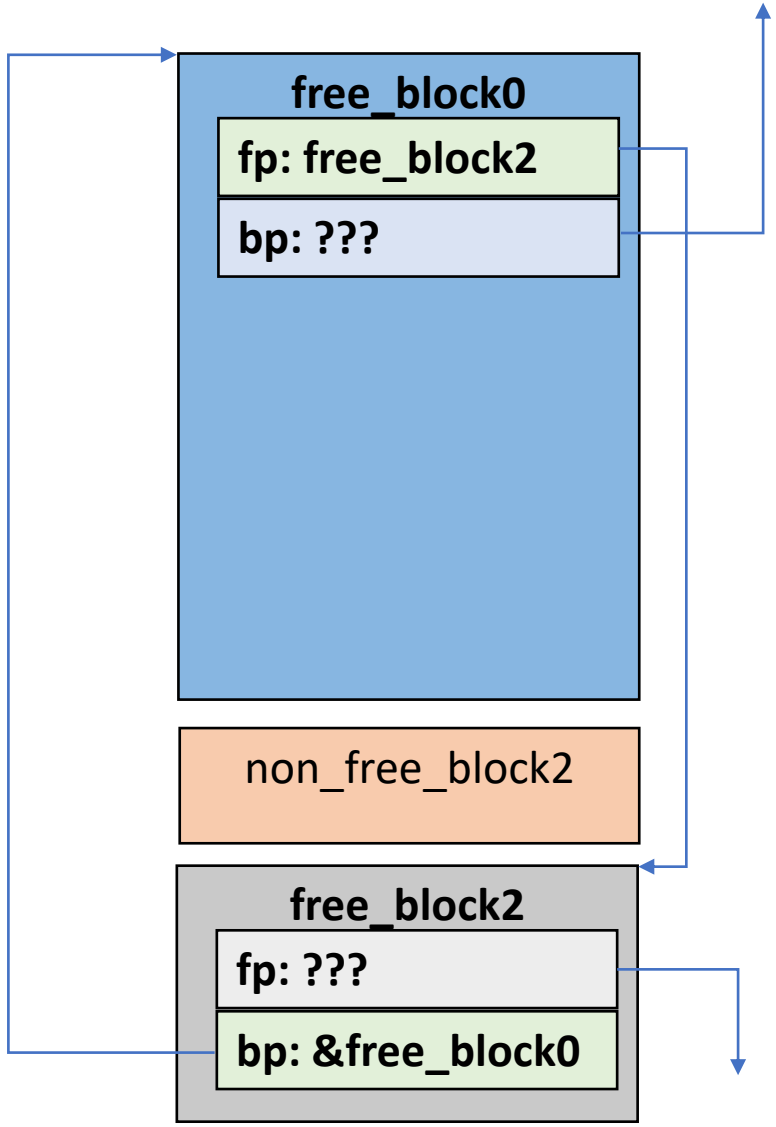


Free



Free
non_free_block1

*(fp + 24) = bp
*(bp + 16) = fp



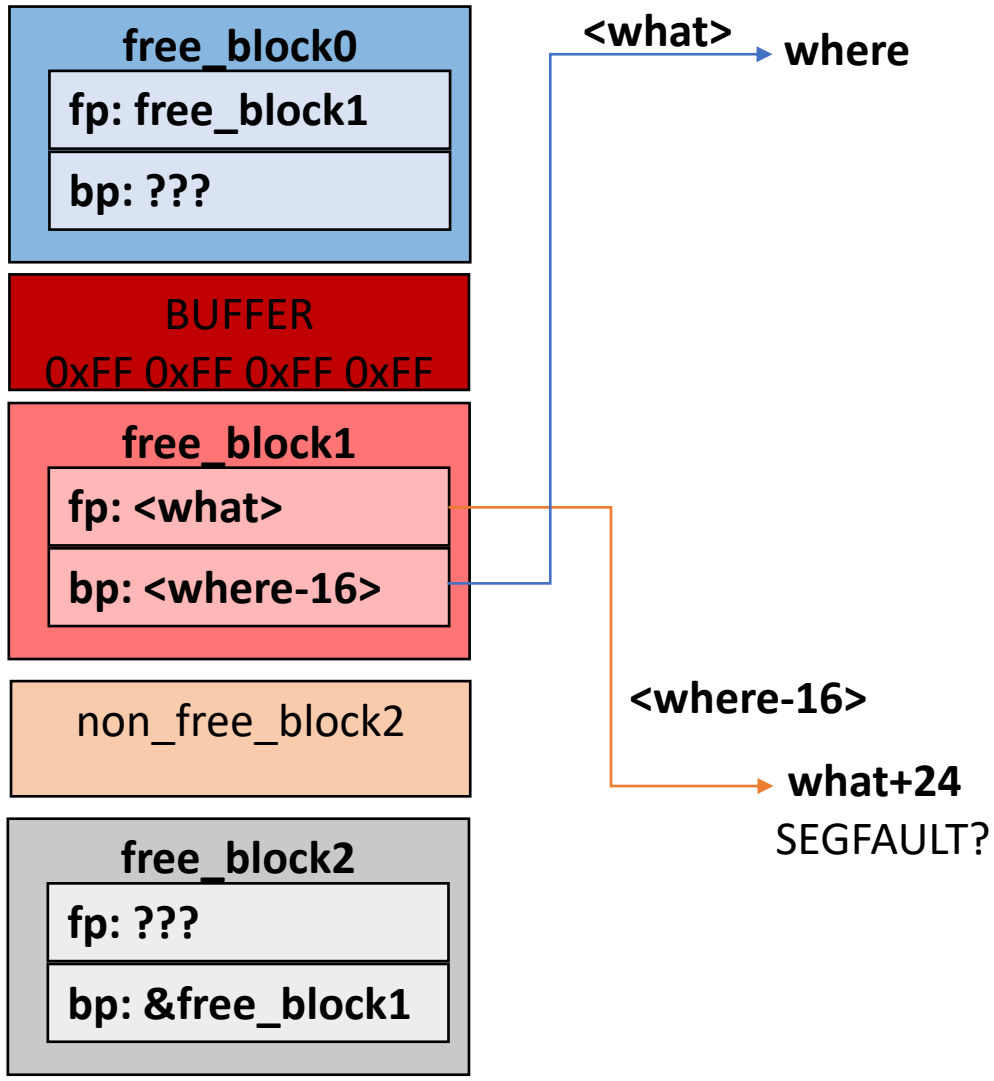
Переполнение в куче

Если буфер данных в куче будет переполнен, то лишние данные будут записаны в управляющие структуры следующего блока.

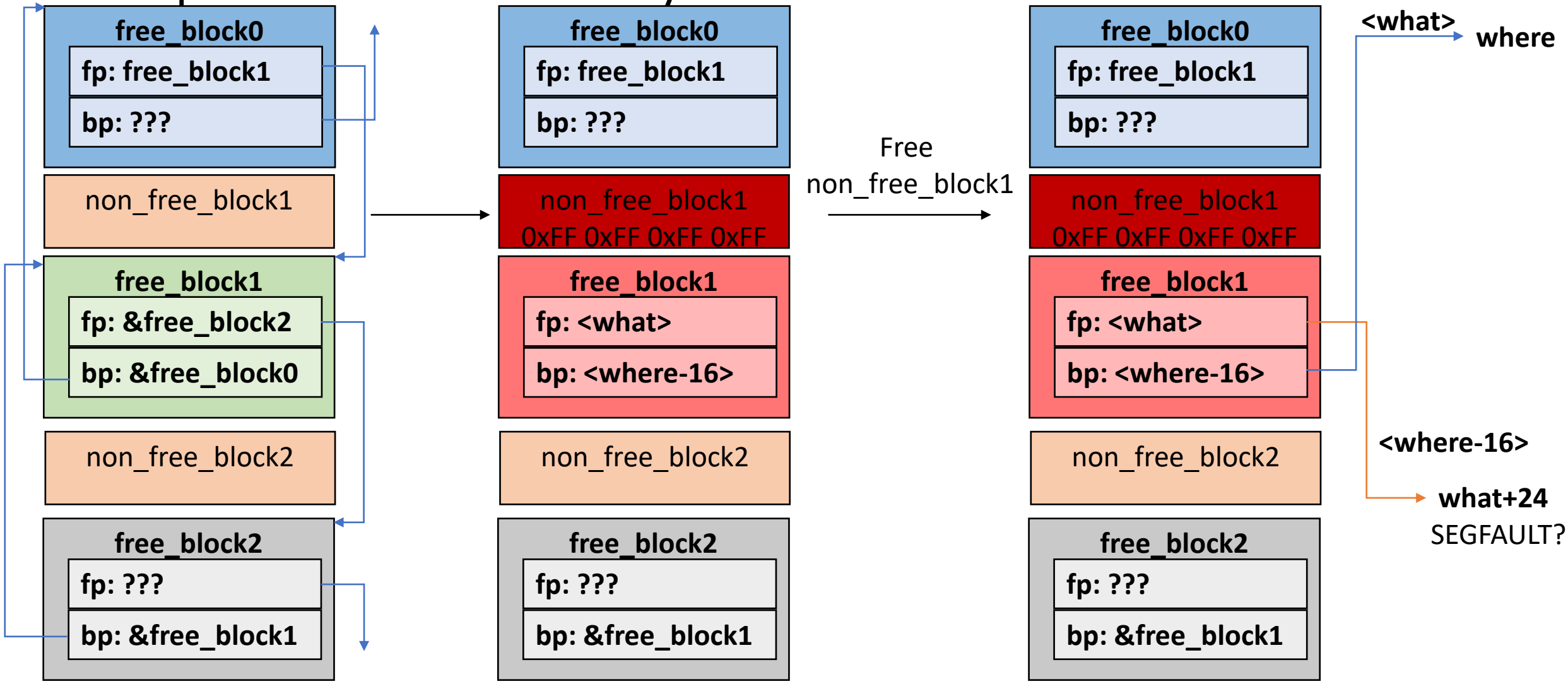
Если следующий блок является свободным, то при слиянии этого блока с предыдущим свободным блоком произойдут 2 записи данных по адресам в соответствующих полях.

В отличие от переполнения буфера на стеке, данный метод позволяет перезаписывать *произвольные* адреса.

Изменение размера предыдущего или текущего блока или флагов также позволяют изменить поведение последующих malloc/free().



Переполнение в куче

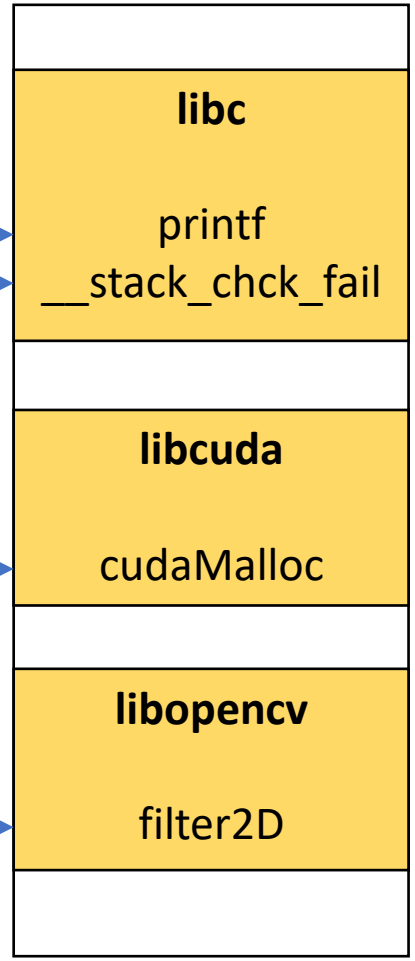


Перезапись GOT (Linux)

Поскольку в GOT происходит запись, она находится в сегменте с разрешением на запись.

Следовательно, если известно расположение самой GOT, можно
1)узнать адреса функций и содержащих их библиотек;
2) перезаписать адрес функции в GOT и передать управление в нужное место.

```
GOT:
    printf:             dq ?
    __stack_chck_fail:  dq ?
    cudaMalloc:         dq ?
    filter2D:           dq ?
```

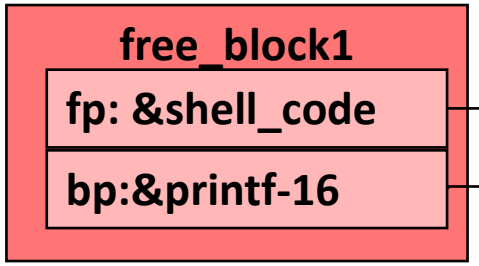


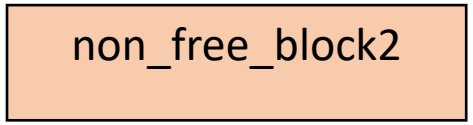
Переполнение в куче (Linux)

*(fp + 24) = bp
*(bp + 16) = fp

*(&shell_code + 24) = &printf-16
*(&printf) = &shell_code

&_chck_fail-16 => *часть буфера перезапишется*





&shell_code

```
GOT:
printf:              dq &shell_code
__stack_chck_fail:   dq ?
cudaMalloc:          dq ?
filter2D:            dq ?
```

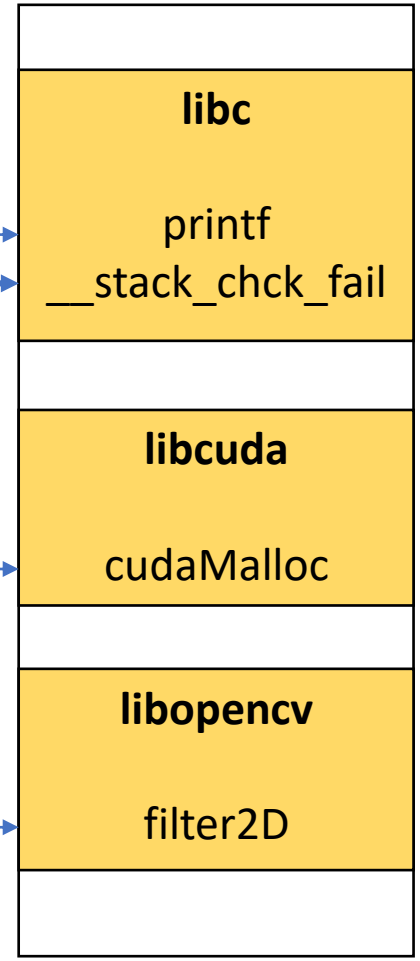
Обход «канареек» 1 (Linux)

Функция __stack_chck_fail или ее аналог (завершение программы при изменении «канарейки») часто является библиотечной функцией, адрес которой находится в GOT.

Если удастся перезаписать адрес функции на адрес целевого кода, то код выполнится автоматически, когда проверка «канарейки» провалится.

В качестве такого кода может выступать как шелл-код, так и простая инструкция RET/JMP.

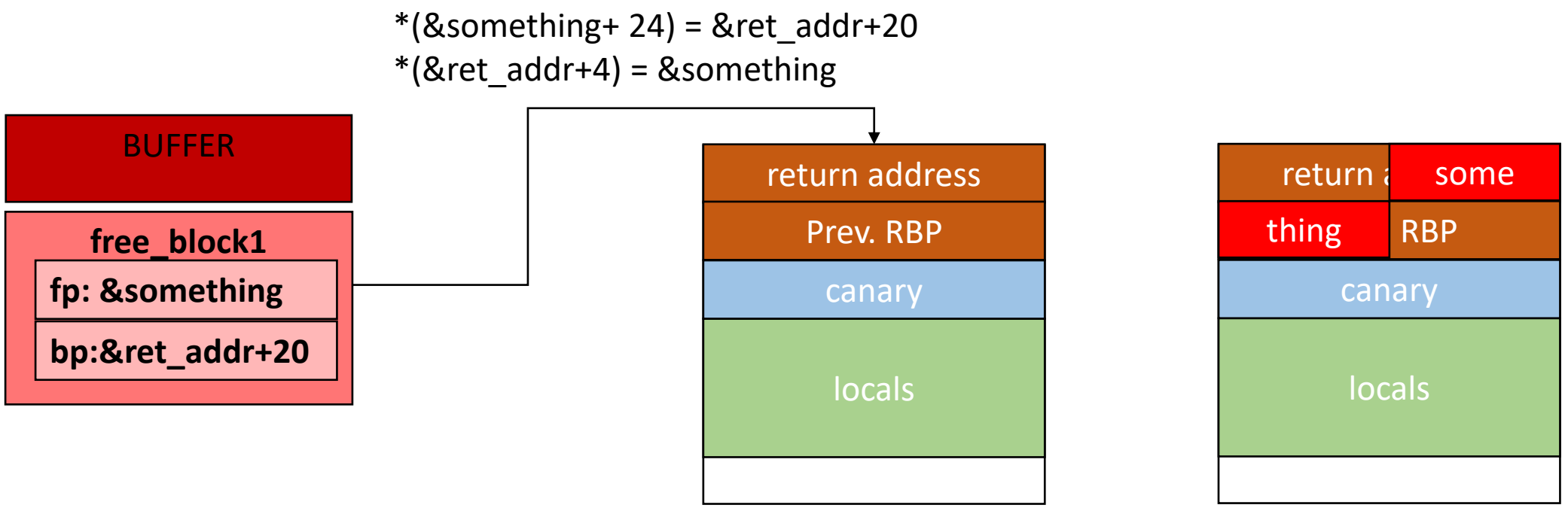
```
GOT:
    printf:              dq ?
    __stack_chck_fail:   dq ?
    cudaMalloc:          dq ?
    filter2D:            dq ?
```



Обход «канареек» 2 (Linux)

Поскольку стек является гарантированно перезаписываемым, можно полностью или частично изменить адрес возврата из функции, не затрагивая канарейку.

Т.к. вторая запись мешает записать произвольное значение (т.к. это значение при второй записи интерпретируется, как адрес), может быть удобным подобрать такое значение, что его первые (или последние) байты при частичной перезаписи адреса возврата позволят сформировать целевой адрес.



Защита от перезаписи GOT

Эффективной защитой от перезаписи GOT является т.н. RELRO (relocation read-only). Обычно RELRO включается специальным флагом компоновщика.

При включенном RELRO:

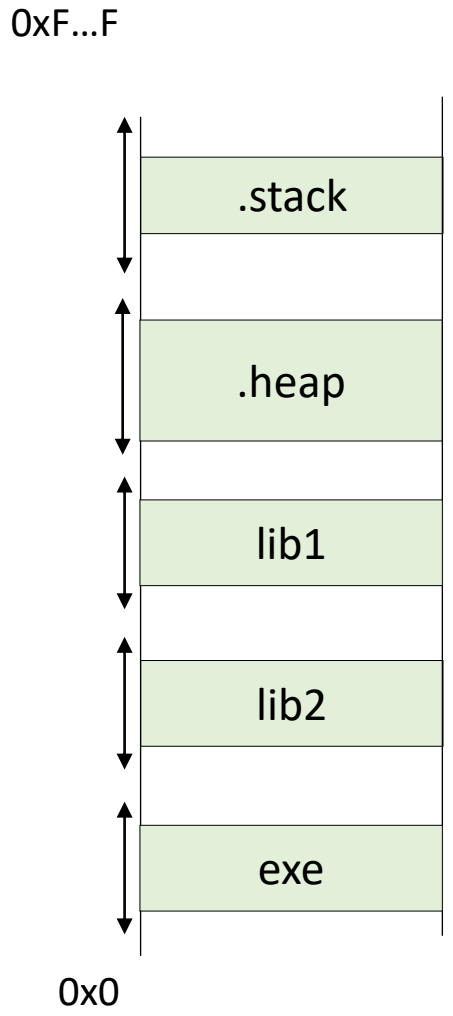
- Таблица GOT заполняется компоновщиком полностью при старте программы.
- Сразу после заполнения секция .got помечается, как read-only. Любая попытка записи приведет к аппаратному исключению.

Address Space Layout Randomization

Для эксплуатации многих уязвимостей необходимо точно знать адреса, по которым происходит запись/передача управления. Если адрес неизвестен, вероятность успешной эксплуатации крайне мала. На этом принципе основан защитный механизм ASLR.

При включенном ASLR адреса стека и кучи, а также адреса загрузки библиотек изменяются при каждом запуске программы.

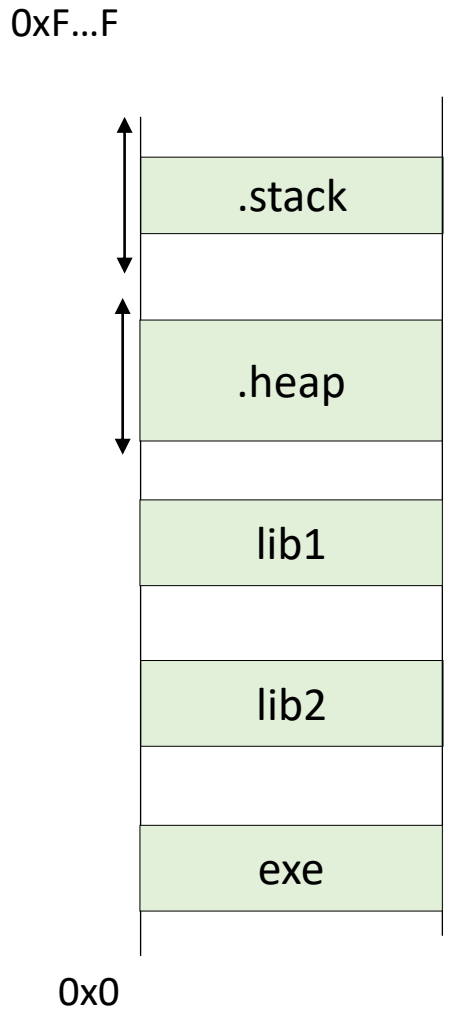
Если без ASLR атакующий может позволить себе «уронить» программу, чтобы узнать адрес и провести атаку позже, то с ASLR такой подход становится невозможным.



ASLR и Windows

В Windows из-за особенностей динамической загрузки библиотеки всегда загружаются по одному и тому же адресу *до следующей перезагрузки* в силу необходимости релокации при загрузки по новому адресу.

Стек и куча имеют каждый раз случайный адрес.



Меры защиты на уровне компилятора

Многие уязвимости являются следствием ошибок, на которые компилятор реагирует предупреждениями => **читайте предупреждения компилятора**.

Многие дистрибутивы Linux устанавливают флаги системного компилятора, посылаемые при компиляции. Обычно эти флаги включают в себя вышеупомянутые «канарейки» и Intel IBT. Как следствие, собираемые программы будут защищены автоматически.

Кроме того, при включенной оптимизации (а иногда и без нее) компиляторы могут располагать переменные после буферов, чтобы защитить их от перезаписи.

GCC/clang поддерживают макрос _FORTIFY_SOURCE. Если этот макрос определен и его значение не равно 0, то компилятор может вставлять дополнительные проверки границ буферов и автоматически заменять небезопасные функции ввода/вывода на безопасные.

Полезные ссылки

Простые задачи на эксплуатацию уязвимостей:
https://exploit.education/protostar/

Канал с неплохими обучающими видео
https://www.youtube.com/channel/UClcE-kVhqyiHCcjYwcpfj9w

Реализация malloc:
https://code.woboq.org/userspace/glibc/malloc/malloc.c.html

Эксплуатация heap3 (переполнение в куче для x86):
https://airman604.medium.com/protostar-heap-3-walkthrough-56d9334bcd13