

Низкоуровневое программирование

Лекция 6

Уязвимости форматной строки

Переполнение на стеке

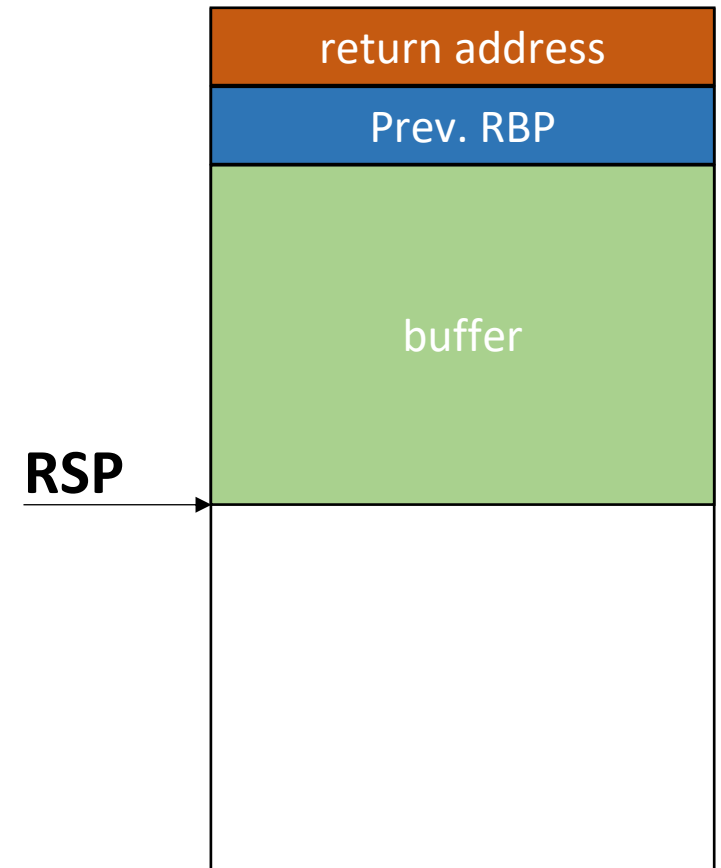
Переполнение в куче

ASLR

Уязвимости форматной строки

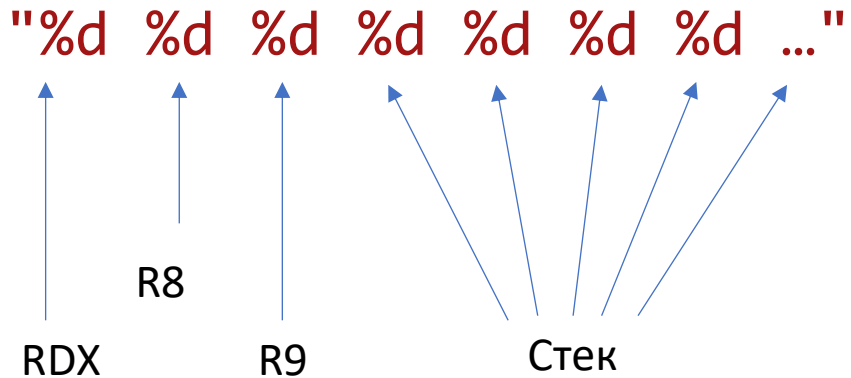
```
void echo() {  
    char buffer[100] ;  
    scanf("%s", buffer);  
    printf(buffer);  
}  
  
int main() {  
    echo();  
}
```

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

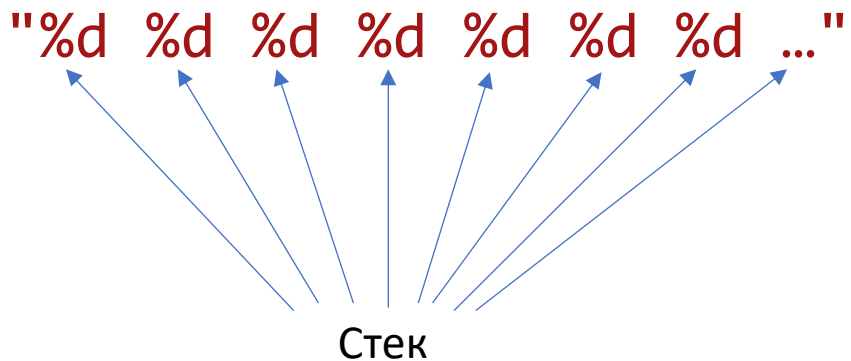


Уязвимости форматной строки

Microsoft x64:



cdecl:



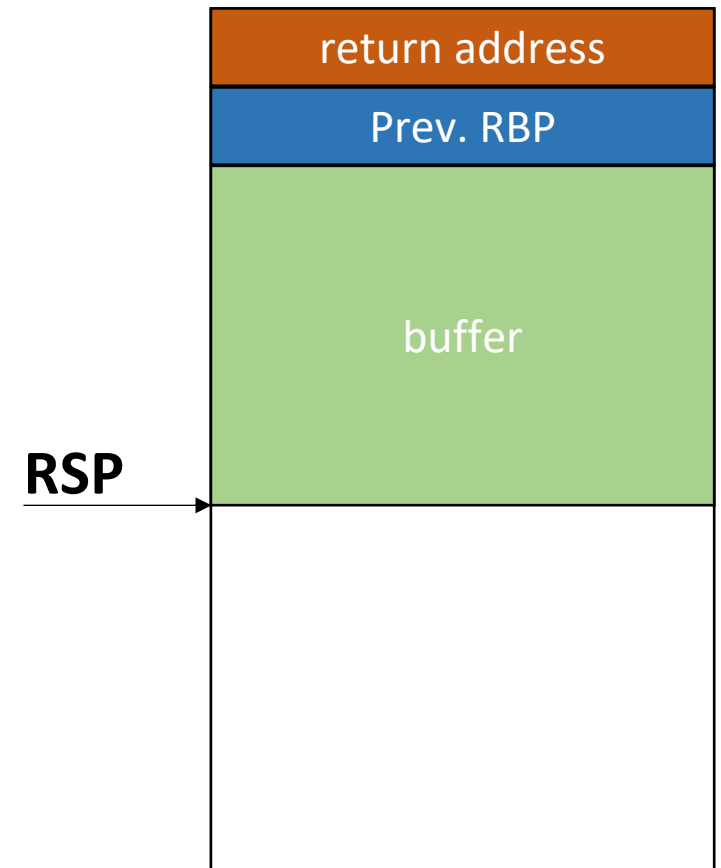
```
void echo() {  
    char buffer[100] ;  
    scanf("%s", buffer);  
    printf(buffer);  
}  
  
int main() {  
    echo();  
}
```

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

%11x

Чтение 8 байт

%p

Чтение указателя (8 байт на x86-64)

%n

Запись числа напечатанных символов
по указателю `int*`

%11n

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

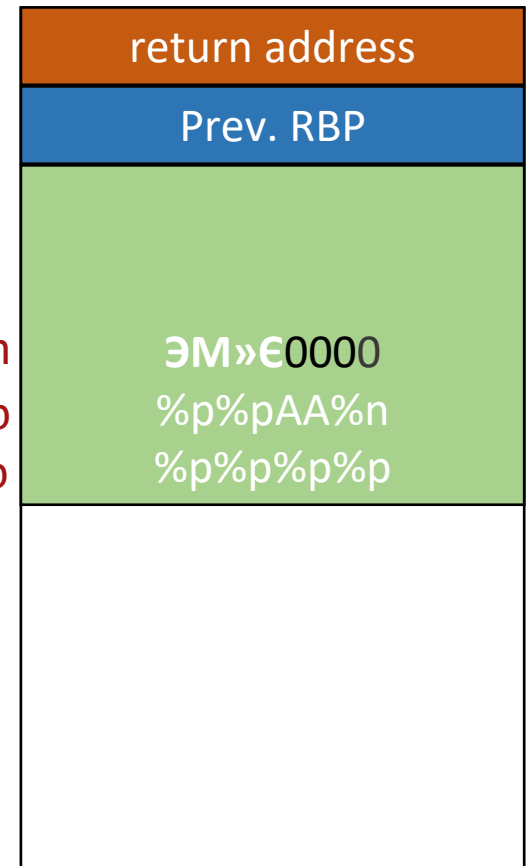
```
void echo() {  
    char buffer[100]{}; //zeros  
    scanf("%s", buffer); //%p%p%p%p%p%pAA%nЭМ»€  
    printf(buffer);  
}
```

```
int global_variable = 0;
```

0x00000000AABBCCDD

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

%n
%p
%p



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

Буферы

Для реализации ввода/вывода 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.

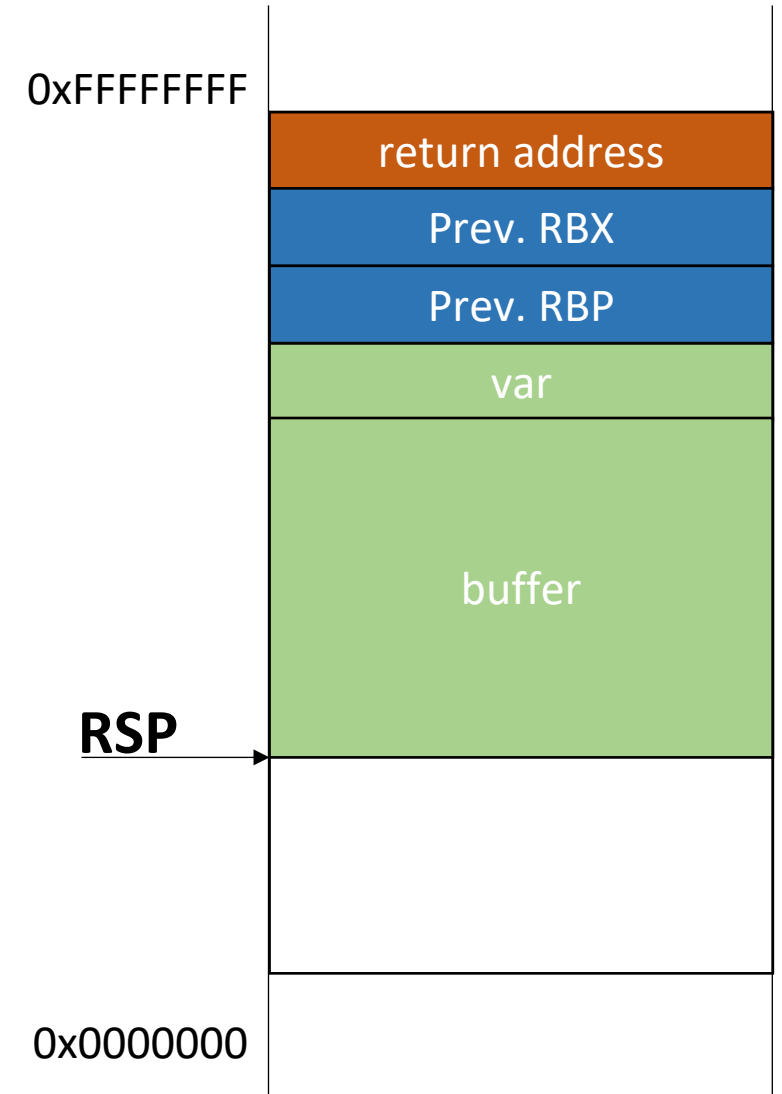
Многие стандартные функции по умолчанию не производят проверку границ буфера.

Переполнение буфера

```
#include <stdio.h>

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

Что будет, если ввести строку длиной больше 15?



Переполнение буфера

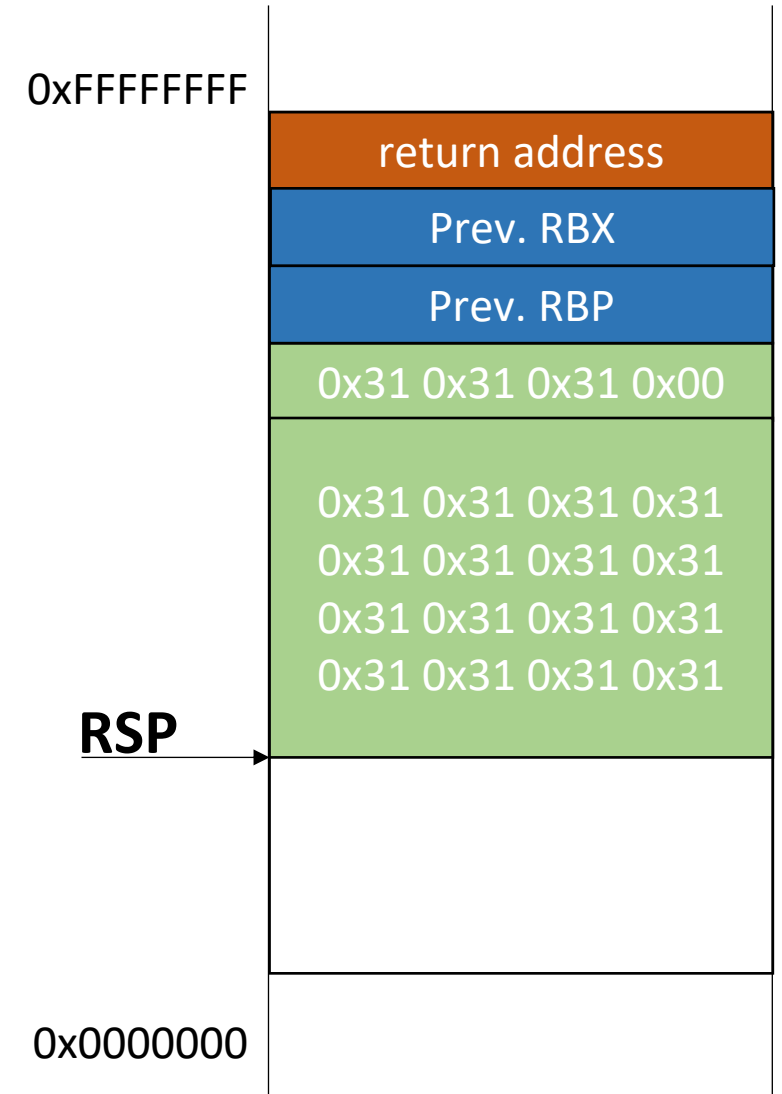
```
#include <stdio.h>

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

Ввод: 11111111111111111111

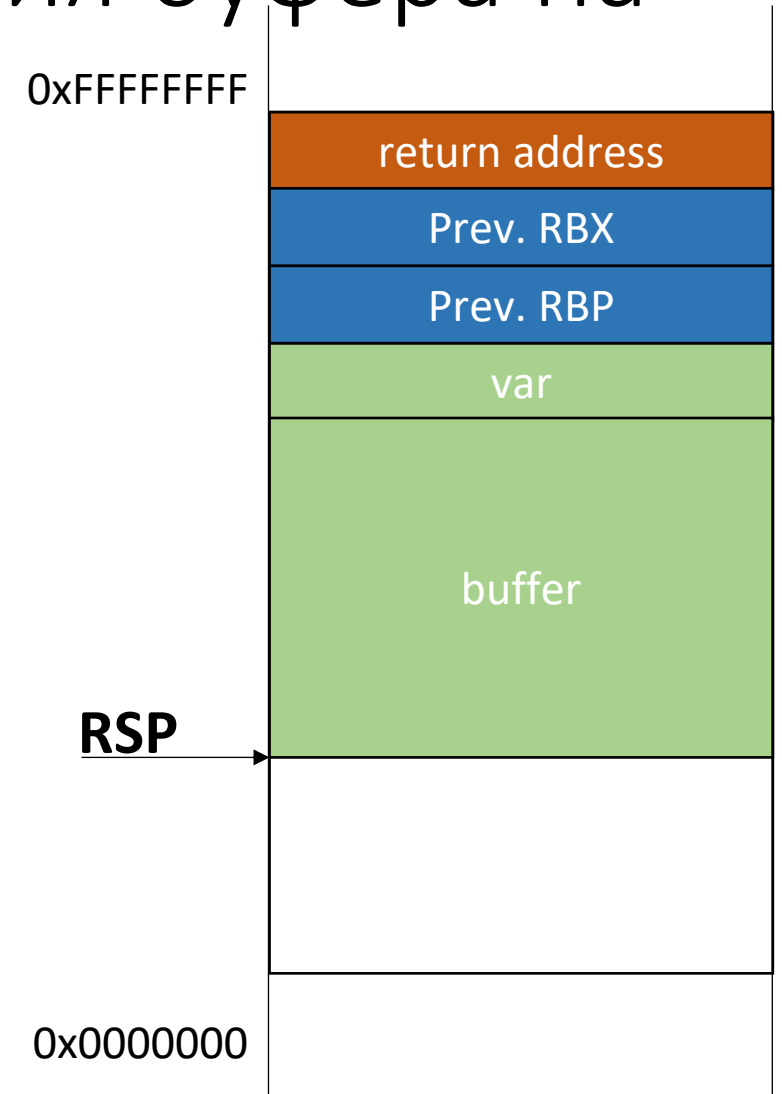
Аналогичные атаки возможны с любой функцией, которая не проверяет размер буфера записи:
gets, strcpy, memcpy, memmove,...

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



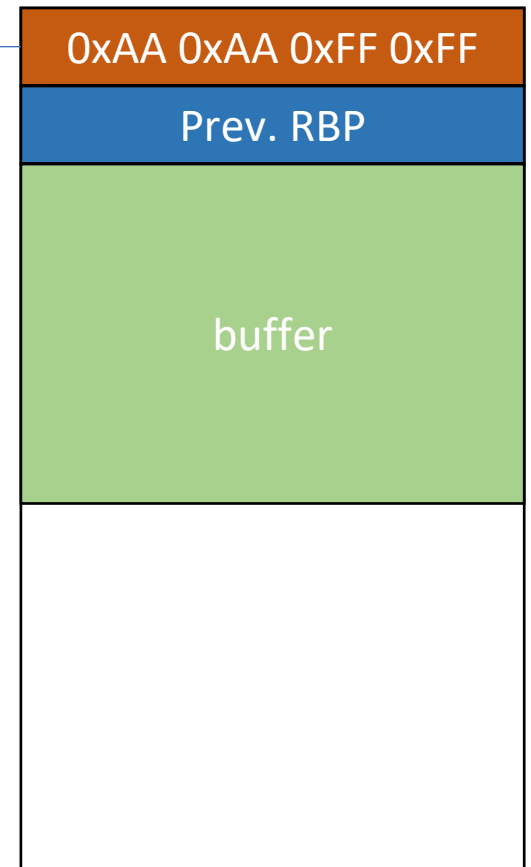
Возможные цели переполнения буфера на стеке

- **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.**



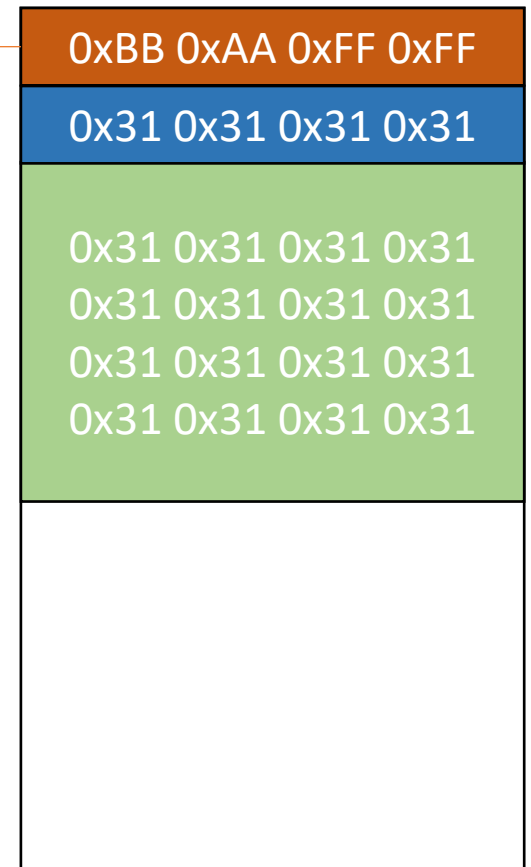
Изменение адреса возврата

```
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*/  
}
```



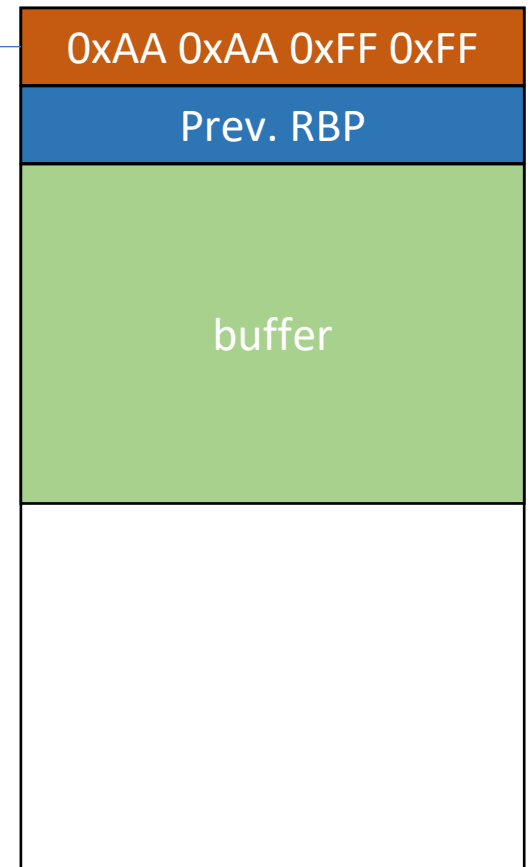
Изменение адреса возврата

```
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*/  
}
```



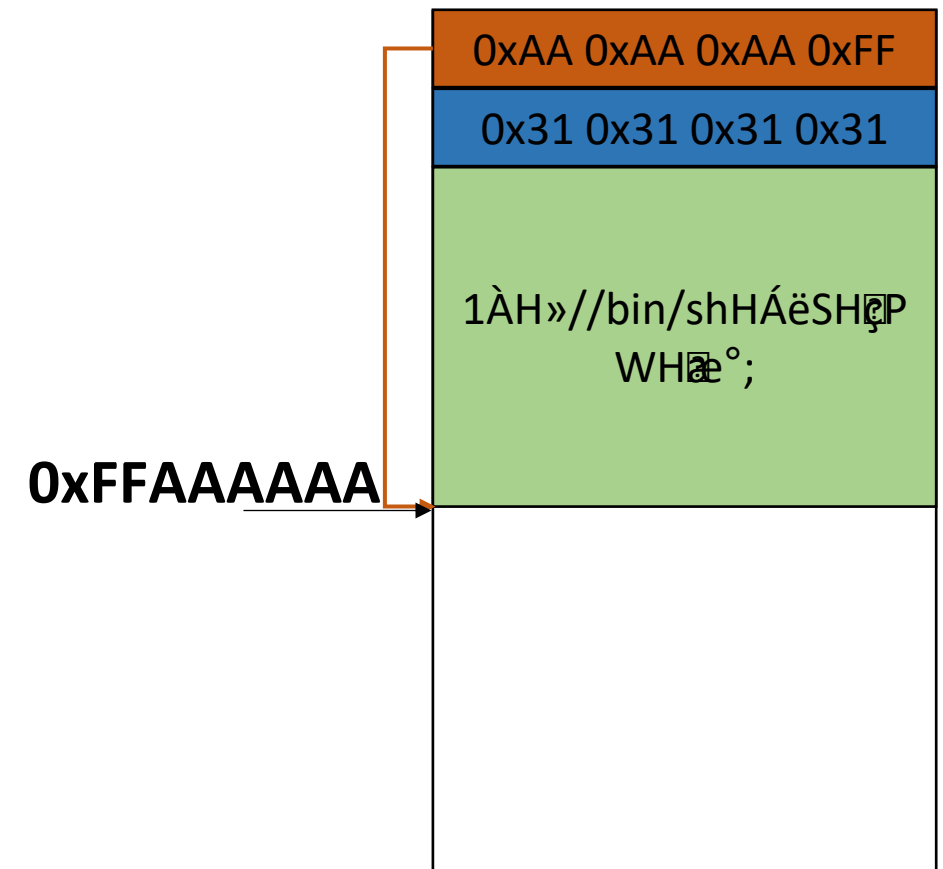
Исполнение произвольного кода

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)

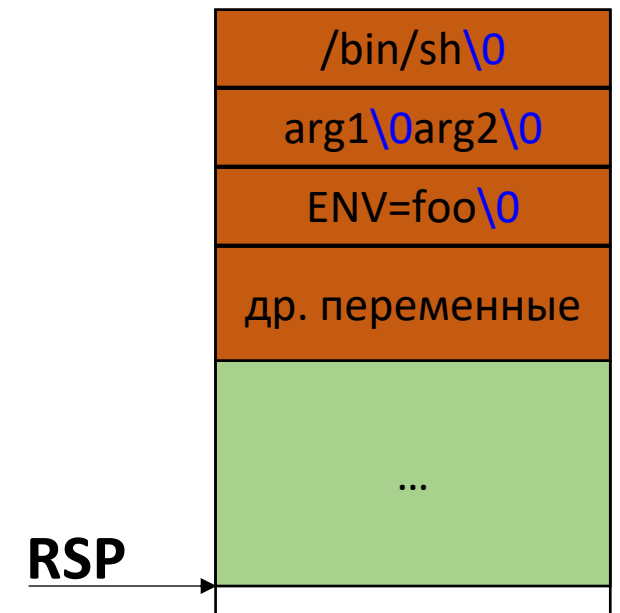
В ОС на ядре Linux аргументы и переменные среды располагаются в начальной области стека – *даже если программа их не использует*.

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.

ENV=foo /bin/sh arg1 arg2

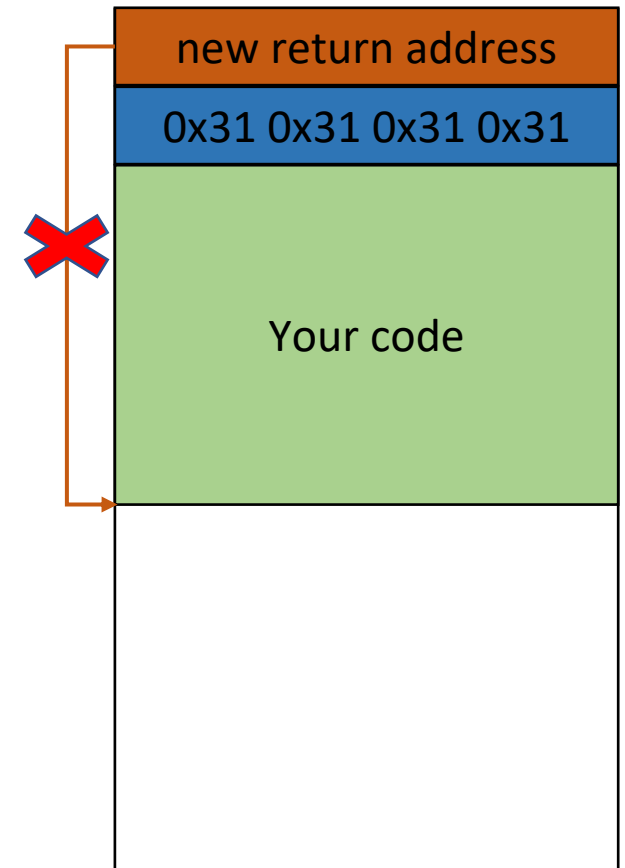


Защита от выполнения

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

При попытке исполнения кода внутри сегментов `.data`, `.rodata`, `.bss`, стека или кучи будет сгенерировано аппаратное исключение и *программа завершится*.

Примечание: можно сделать стек исполняемым – но пользователь должен явно запросить это при сборке программы.



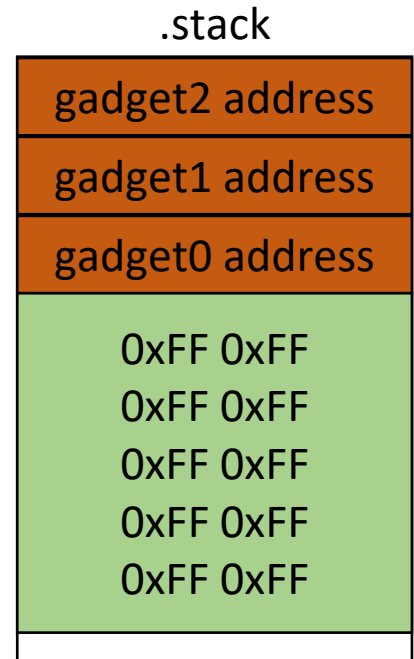
Return Oriented Programming

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**.

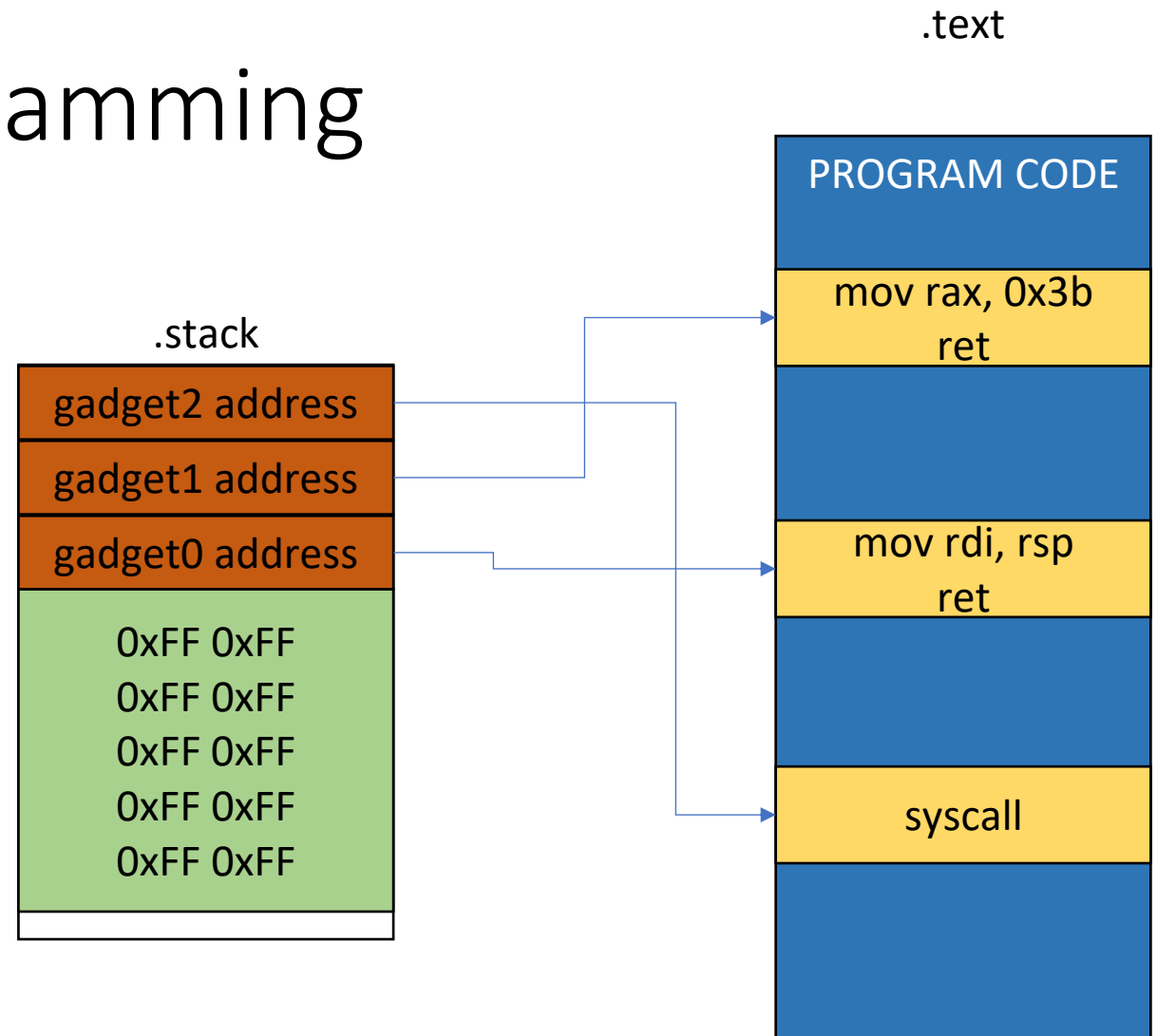
Примечание: помимо `ret` могут использоваться инструкции перехода по вычисляемому аргументу, например `jmp [eax]` (Jump Oriented Programming) или `call [eax]` (Call Oriented Programming) - в этом случае аргумент должен содержать целевой адрес.



Return Oriented Programming

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

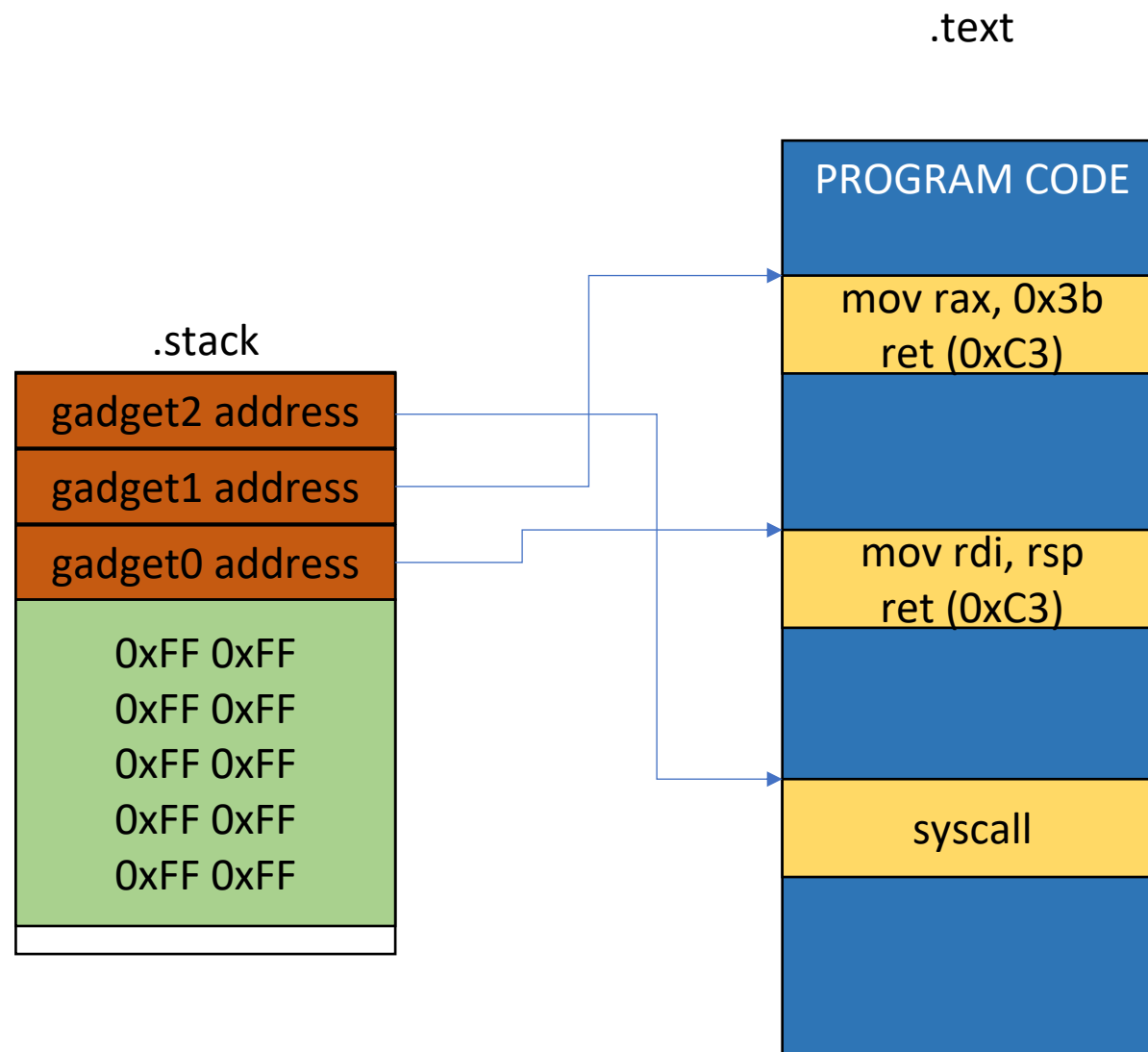
Переход от гаджета к гаджету осуществляется инструкцией `ret`, которая достает адрес следующего гаджета со стека.



Поиск гаджетов

Гаджеты обычно ищутся специально написанным ПО.

В самом простом случае маркером гаджета является инструкция `ret`, имеющая код `0xC3`. Если байт с данным значением найден, просматриваются байты «выше» него. Если они представляют собой нужные инструкции, гаджет запоминается, иначе – поиск идет дальше.



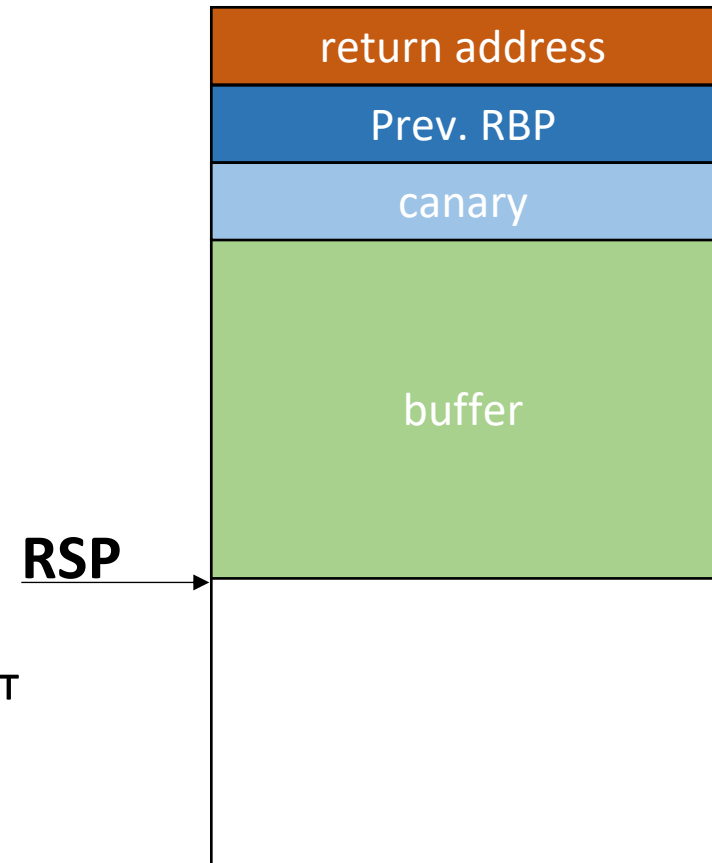
Защита от изменения адреса возврата

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

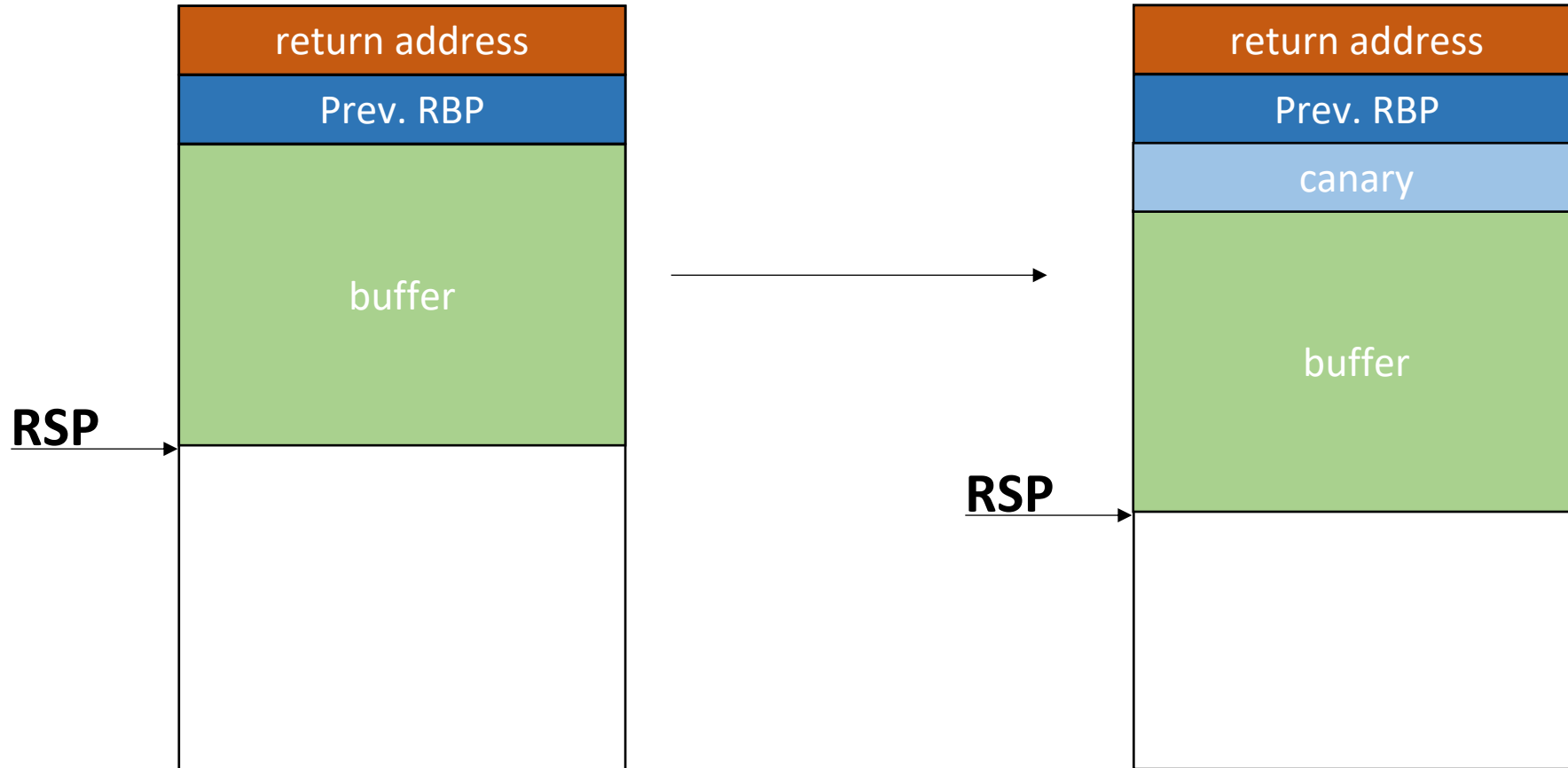
«Канарейка» - это копия значения специальной глобальной переменной, которая записывается на стек при входе в функцию. При выходе из функции значение канарейки сверяется с исходным. Если значения не совпали – возникает исключение.

Поскольку исходная глобальная переменная находится не на стеке, атакующий не может на нее повлиять с помощью переполнения буфера на стеке. Значение глобальной переменной выбирается случайным образом при запуске программы, т.е. атакующий не может его предсказать.

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



Защита от изменения адреса возврата

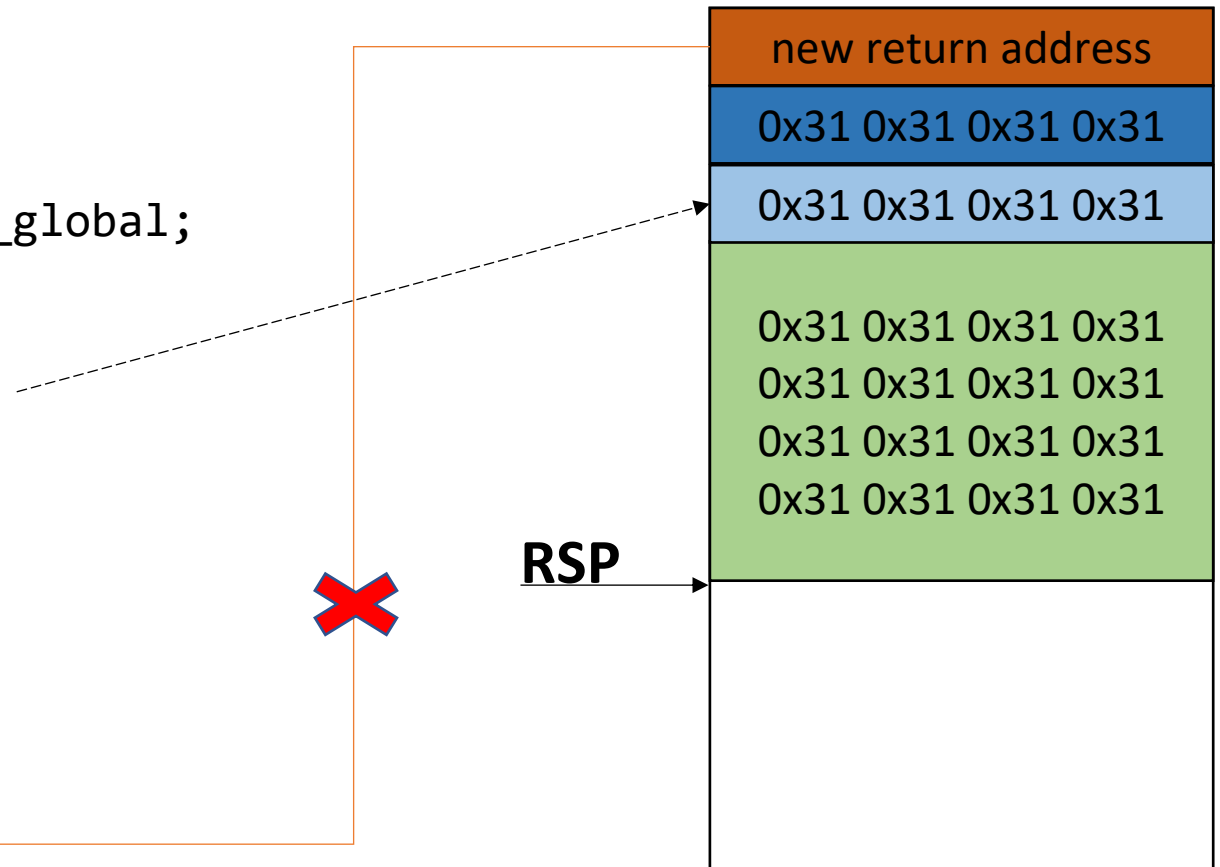


Защита от изменения адреса возврата

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

bool pass_check() {
    volatile 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*/
}
```

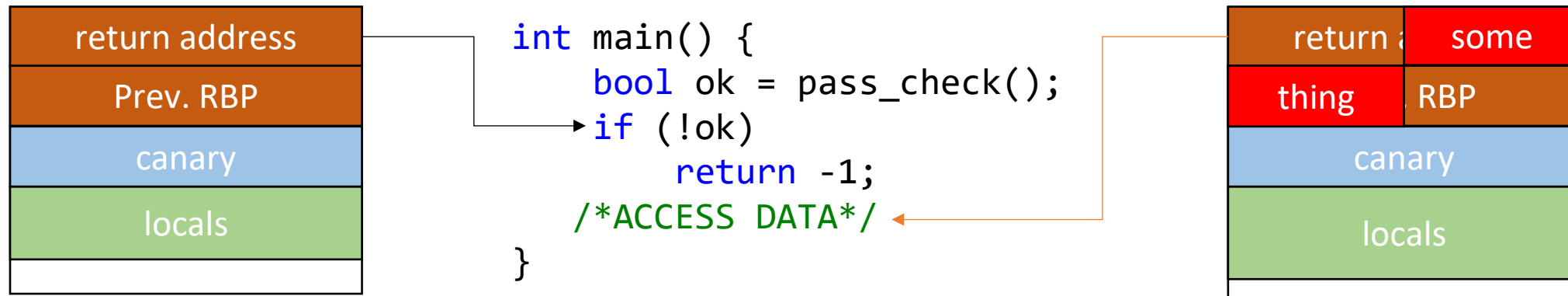


Обход «канареек» 1 (Linux)

При наличии уязвимости, обеспечивающей возможность записи по произвольным адресам, можно обойти «канарейку» и изменить значение на стеке, не затрагивая «канарейку».

К уязвимостям подобного типа относятся в т.ч. уязвимости форматной строки и переполнения в куче (см. далее).

Стоит отметить, что во многих случаях не требуется полной перезаписи переменной или адреса возврата.



Защита на аппаратном уровне: 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.

Защита на аппаратном уровне : Intel Indirect Branch Tracking

Shadow Stack защищает от ROP, но не защищает от JOP и COP, поскольку в этом случае адреса возврата не используются.

Технология Intel IBT позволяет усложнить атакующему эксплуатацию JOP- и COP-цепочек.

Intel IBT вводит 2 новые инструкции `endbr32/endbr64` (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 по нефиксированному адресу (`call/jmp/j* [eax]`), и он попадает не на `endbr`, то генерируется аппаратное исключение и программа аварийно завершается => ограничивается выбор гаджетов для атакующего.

Перезапись GOT (Linux)

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

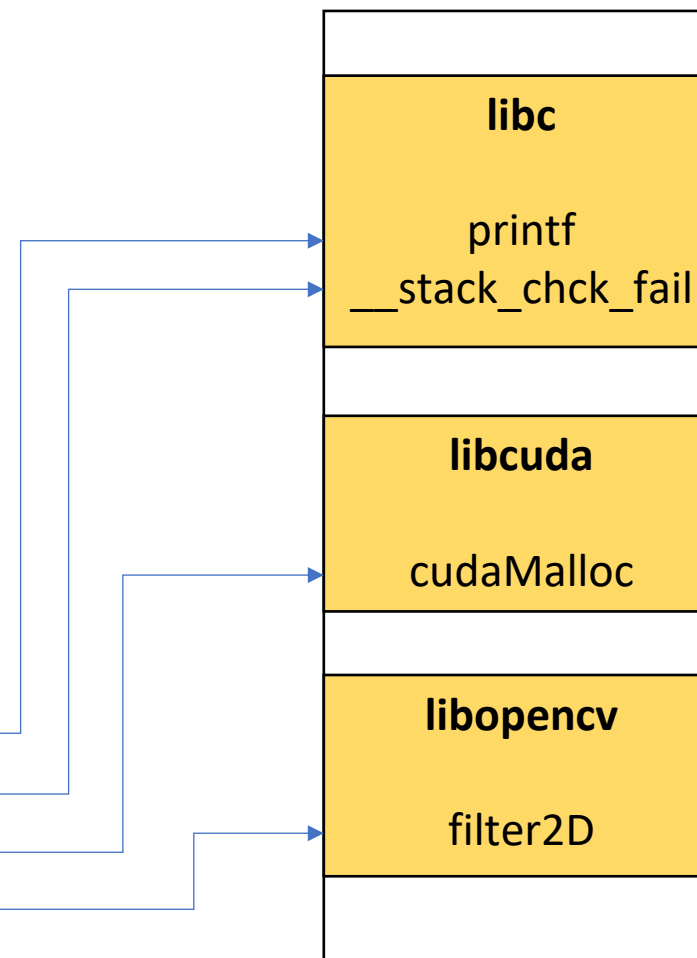
Следовательно, если известно расположение самой GOT, можно

- 1) узнать адреса функций и содержащих их библиотек;
- 2) перезаписать адрес функции в GOT и передать управление в нужное место.

Возможность перезаписи GOT обеспечивается любой уязвимостью, позволяющей выполнять запись по произвольным адресам.

GOT:

printf:	dq ?
__stack_chk_fail:	dq ?
cudaMalloc:	dq ?
filter2D:	dq ?



Обход «канареек» 2 (Linux)

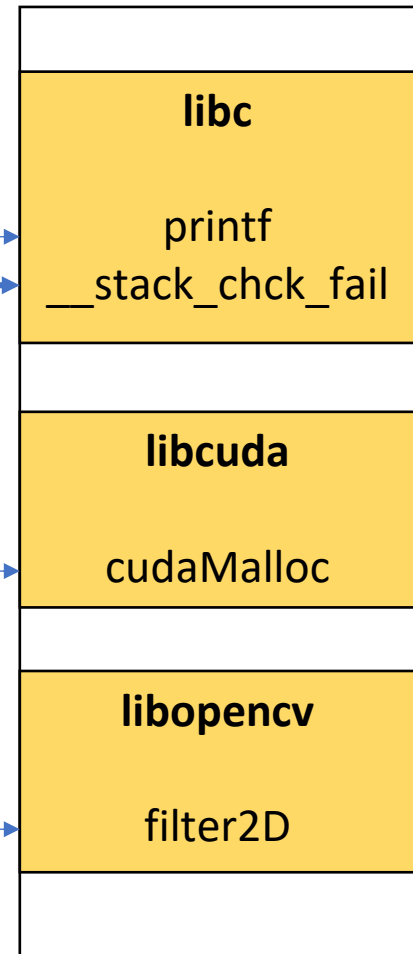
Функция `__stack_chk_fail` или ее аналог (завершение программы при изменении «канарейки») часто является библиотечной функцией, адрес которой хранится в GOT.

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:

<code>printf:</code>	<code>dq ?</code>
<code>__stack_chk_fail:</code>	<code>dq ?</code>
<code>cudaMalloc:</code>	<code>dq ?</code>
<code>filter2D:</code>	<code>dq ?</code>



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.

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, куча является *двусвязным списком свободных блоков памяти*.

chunk prev size/[prev data]			
chunk size [/8]	A	M	P
[data]			

[prev data]			
chunk size [/8]	A	M	1
forward block ptr			
backward_block_ptr			
[freed data]			

Занятые блоки

Структура занятого блока памяти выглядит следующим образом:

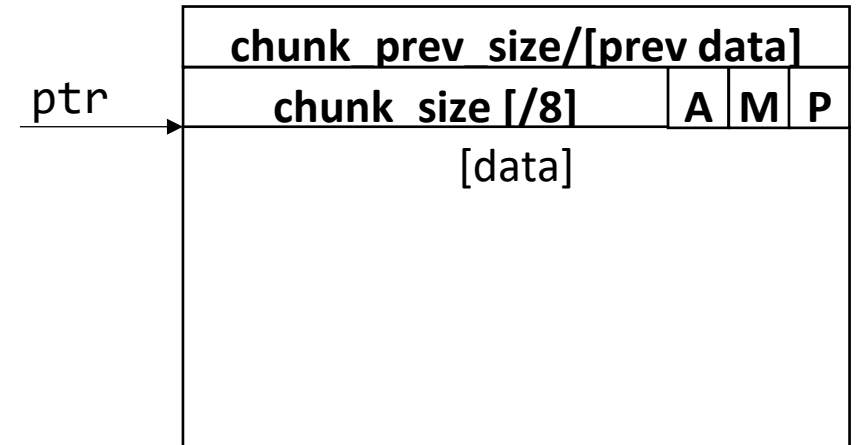
Поле **chunk_size** равно размеру текущего блока. Т.к. размер блока всегда кратен 16 (для x64), последние 3 бита всегда равны 0. Эти 3 бита используются как флаги.

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

Поле **chunk_prev_size** равно размеру предыдущего блока памяти, если предыдущий блок свободен. *Если предыдущий блок занят – в этом поле могут храниться данные предыдущего блока.*

Malloc() всегда возвращает адрес начала данных.

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



Свободные блоки

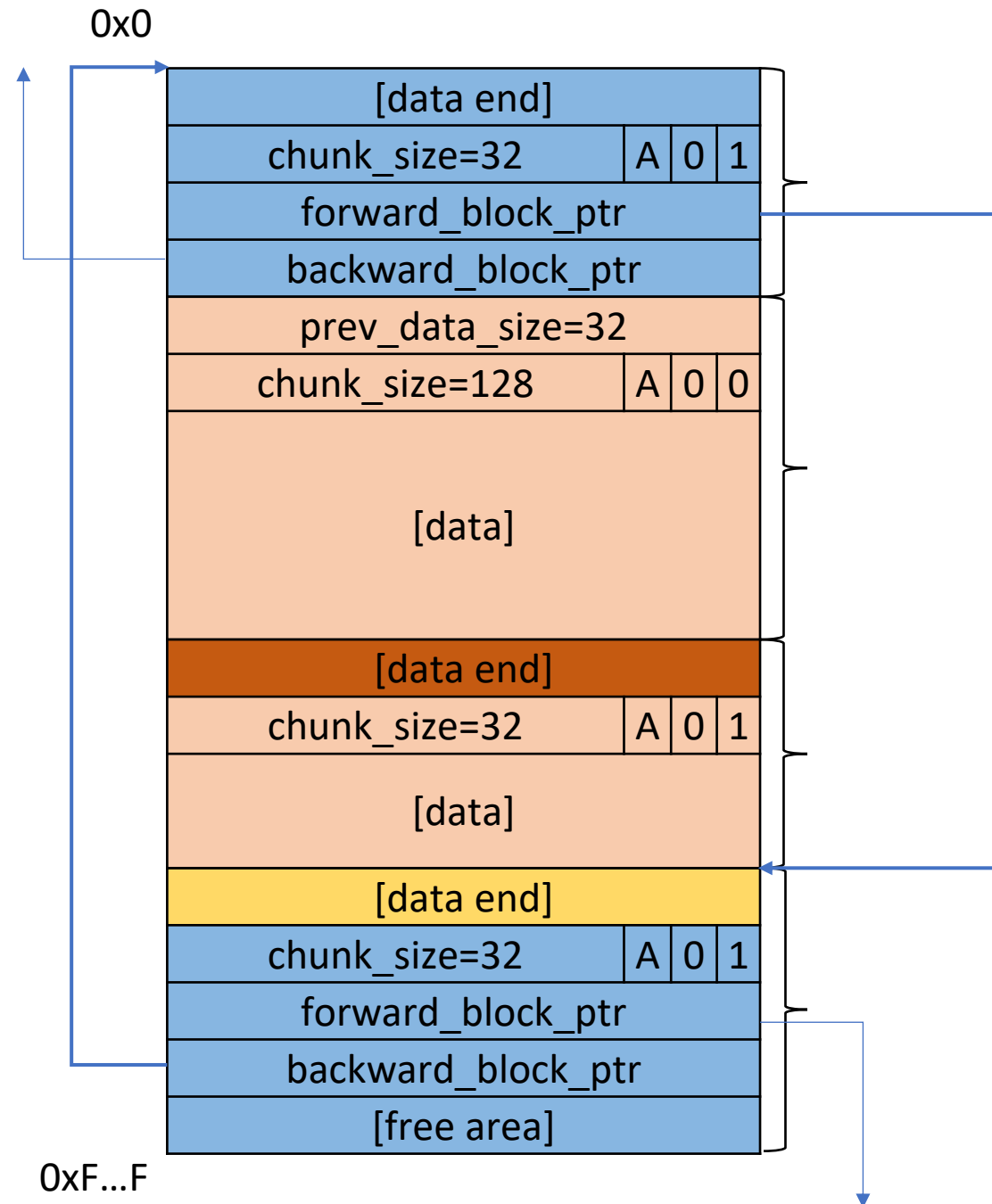
Если блок данных свободен, то в части бывших данных располагаются 2 поля: указатель на предыдущий свободный блок (**forward_block_ptr**) и следующий свободный блок (**backward_block_ptr**) .

Функция free() заполняет эти поля при освобождении блока.

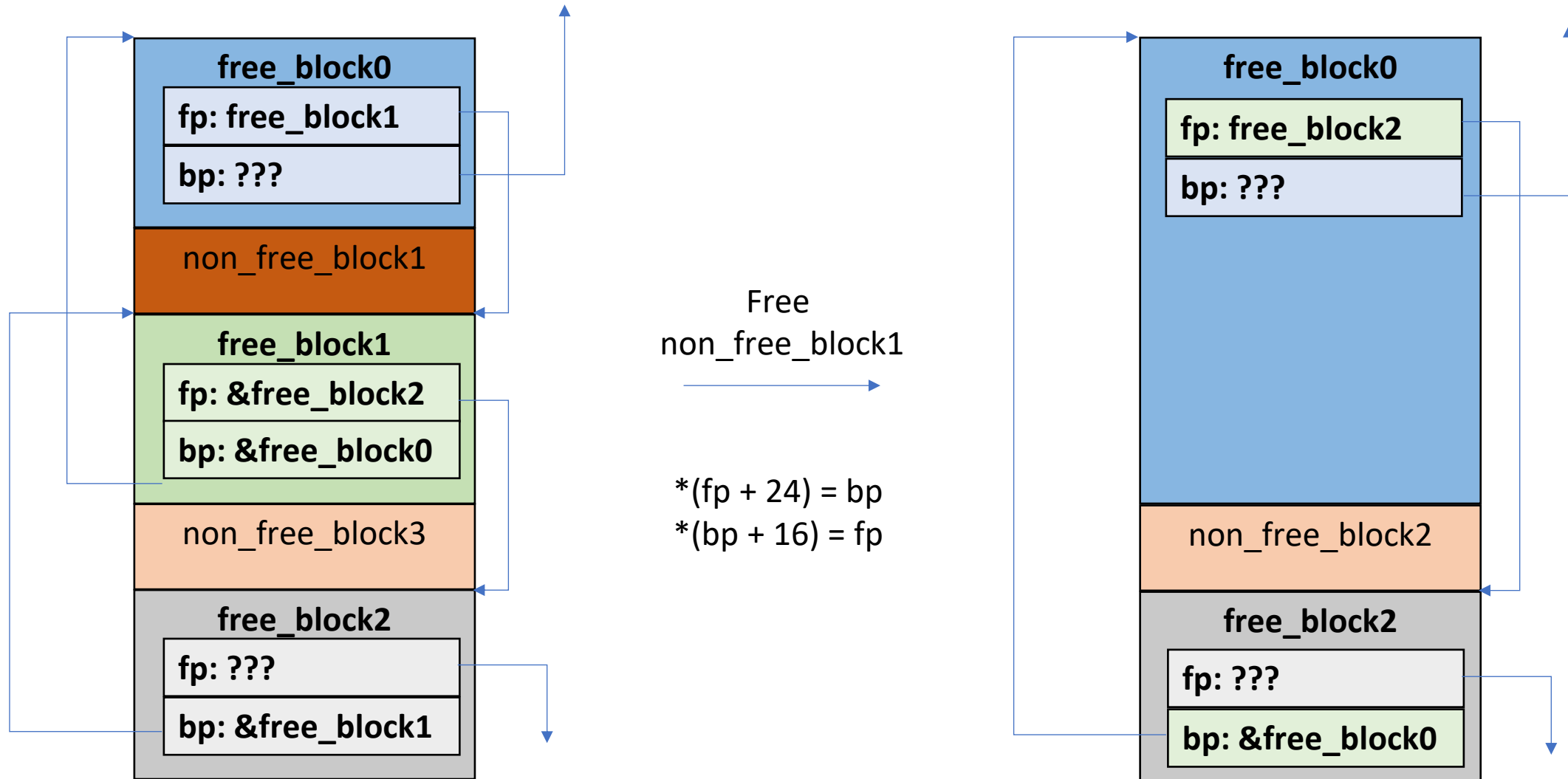
Если освобождается блок данных, перед которым уже есть свободный блок, они объединяются в один => перед свободным блоком и после свободного блока не может находиться другой свободный блок => флаг P=1.

[prev data]			
chunk_size [/8]	A	M	1
forward block ptr			
backward block ptr			
[freed data]			

Структура кучи



Слияние свободных блоков



Переполнение в куче

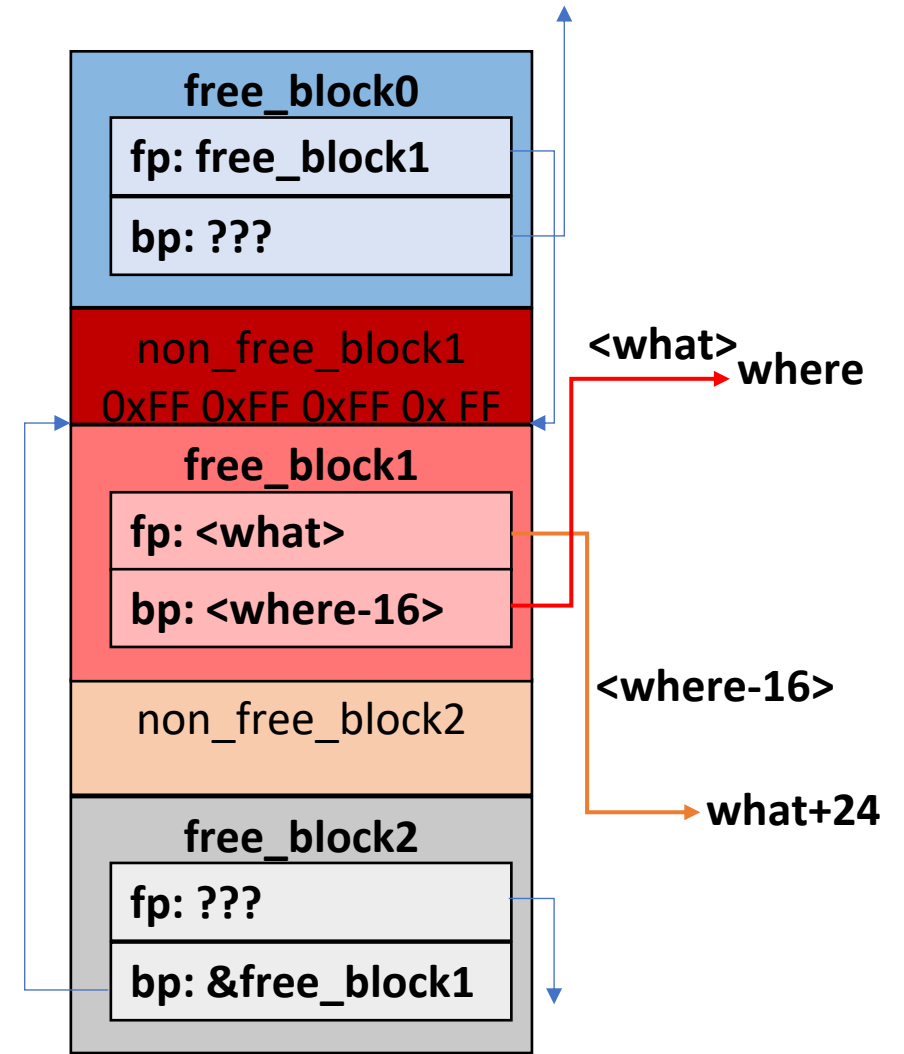
Если буфер данных в куче будет переполнен, то лишние данные будут записаны в управляющие структуры следующего блока.

Если переполненный блок находится между 2 свободными блоками, то при его освобождении произойдет слияние 3 свободных блоков.

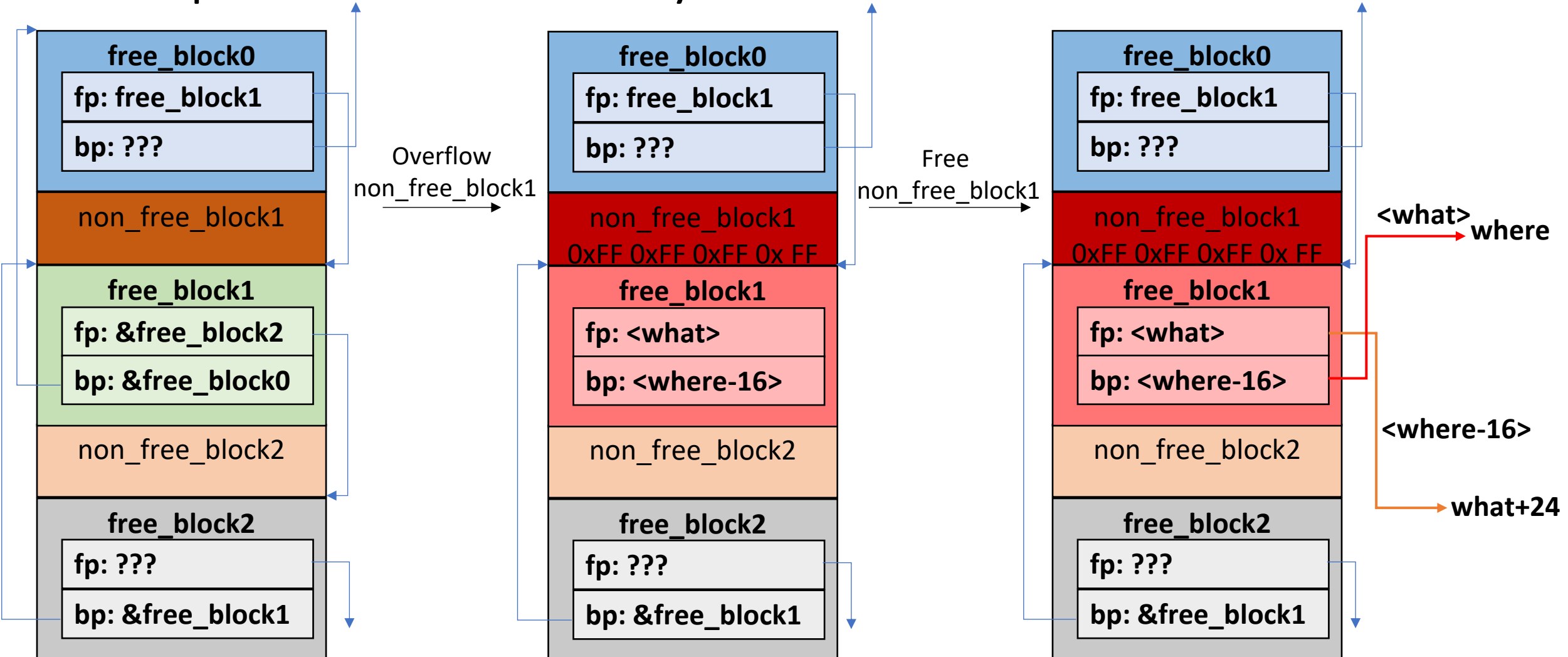
Поскольку указатели в нижнем свободном блоке перезаписаны, при попытке слияния блоков произойдут 2 операции записи по заданным адресам.

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

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



Переполнение в куче

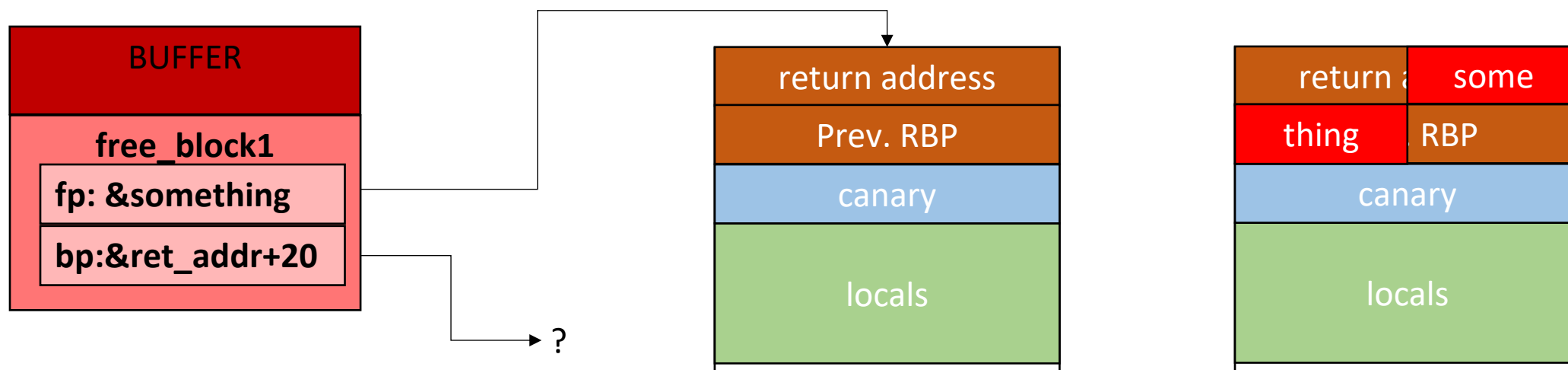


Обход «канареек» через переполнение в куче

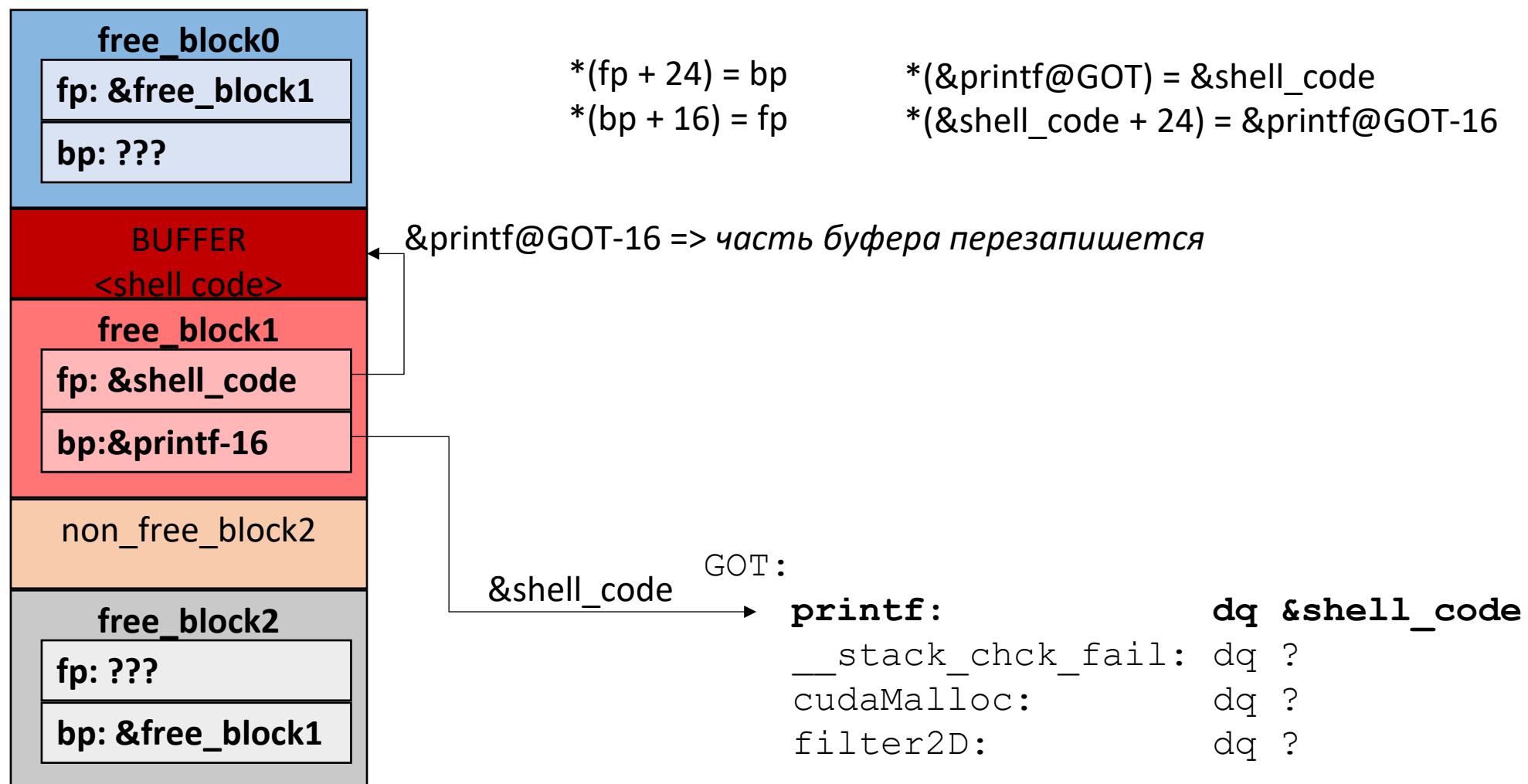
Поскольку переполнение в куче предоставляет возможность произвольной записи, эту уязвимость можно использовать для обхода канареек.

В таком случае значение в `backward_block_ptr` в этом случае нацеливается на адрес возврата или его часть.

Значение в `forward_block_ptr` подбирается так, чтобы вторая операция записи не вызвала SIGSEGV и при этом адрес возврата изменился на нужный атакующему.



Перезапись GOT через переполнение в куче(Linux)



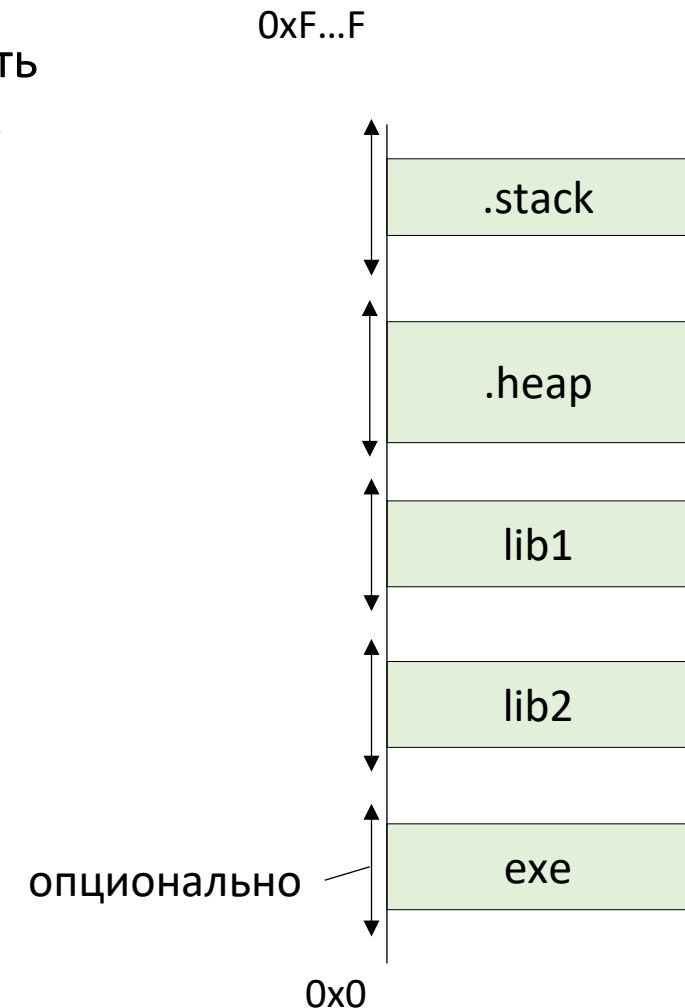
Address Space Layout Randomization

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.

Сама программа загружается по случайному адресу, если она была собрана с поддержкой Position Independent Code (см. лекцию 5). При этом как в Windows, так и в Linux можно включить принудительную рандомизацию адресов ценой усложнения процесса загрузки.

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

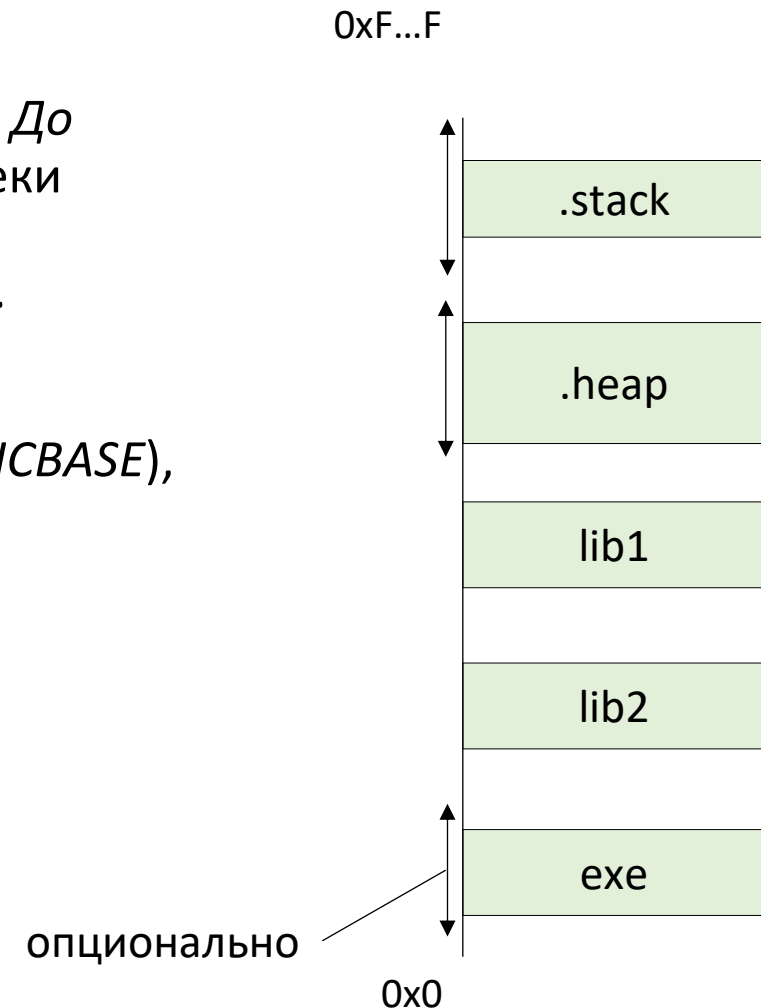


ASLR и Windows

В Windows рандомизация адресов загрузки разделяемых библиотек производится при первой загрузке библиотеки. *До следующей перезагрузки системы* разделяемые библиотеки будут загружаться по одним и тем же адресам в силу необходимости релокации при загрузке по новому адресу.

Стек и куча имеют каждый раз случайный адрес.

Программы, собранные с поддержкой ASLR (флаг `/DYNAMICBASE`), загружаются по случайному адресу при каждом запуске.



Меры защиты на уровне компилятора

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

Многие дистрибутивы Linux устанавливают флаги, неявно посылаемые системному компилятору при сборке программ. 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/UCIcE-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>