

Уязвимости buffer overflow

Переполнения буфера

- Запись чрезмерно большого объёма данных в переменную программы

```
char local_buf[32];  
sprintf(buf, "Message from user: %s", received_data);
```

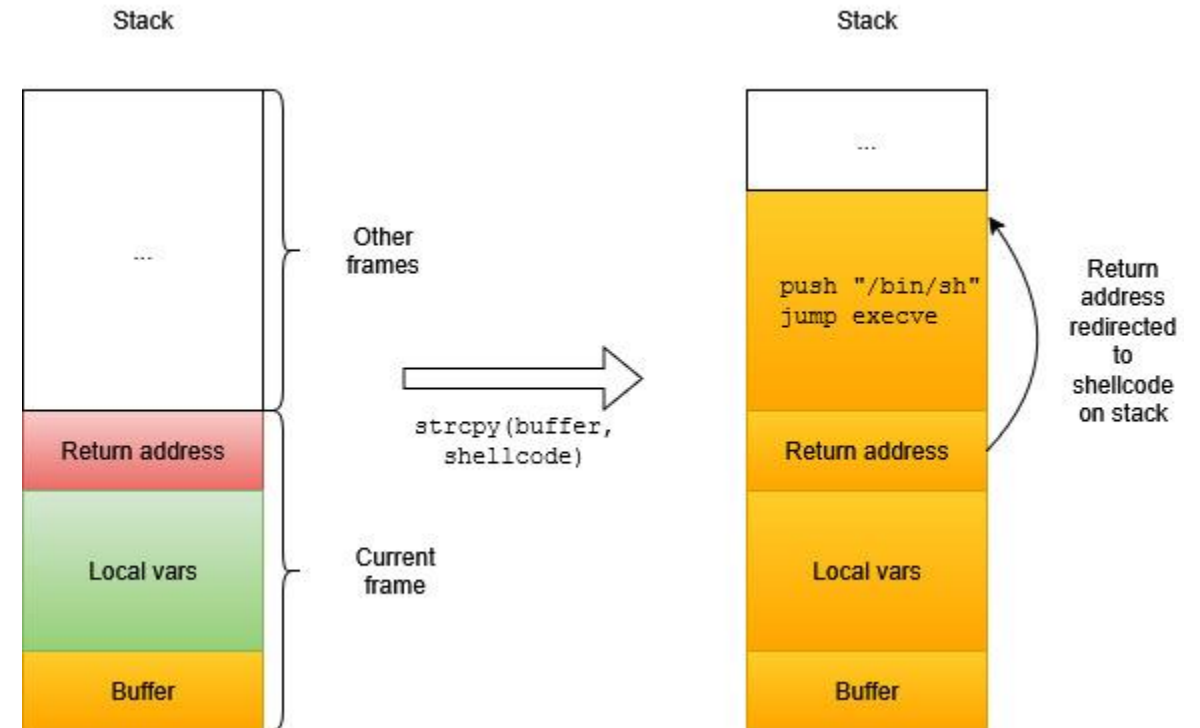
- Переполнение стек (stack overflow)
 - Атаки Stack Smashing, Return-to-libc, Return-Oriented Programming
 - Наиболее стандартизованный и технологичный вид атак
- Переполнение кучи (heap overflow)



<https://www.rawpixel.com/image/5958324/free-public-domain-cc0-photo>

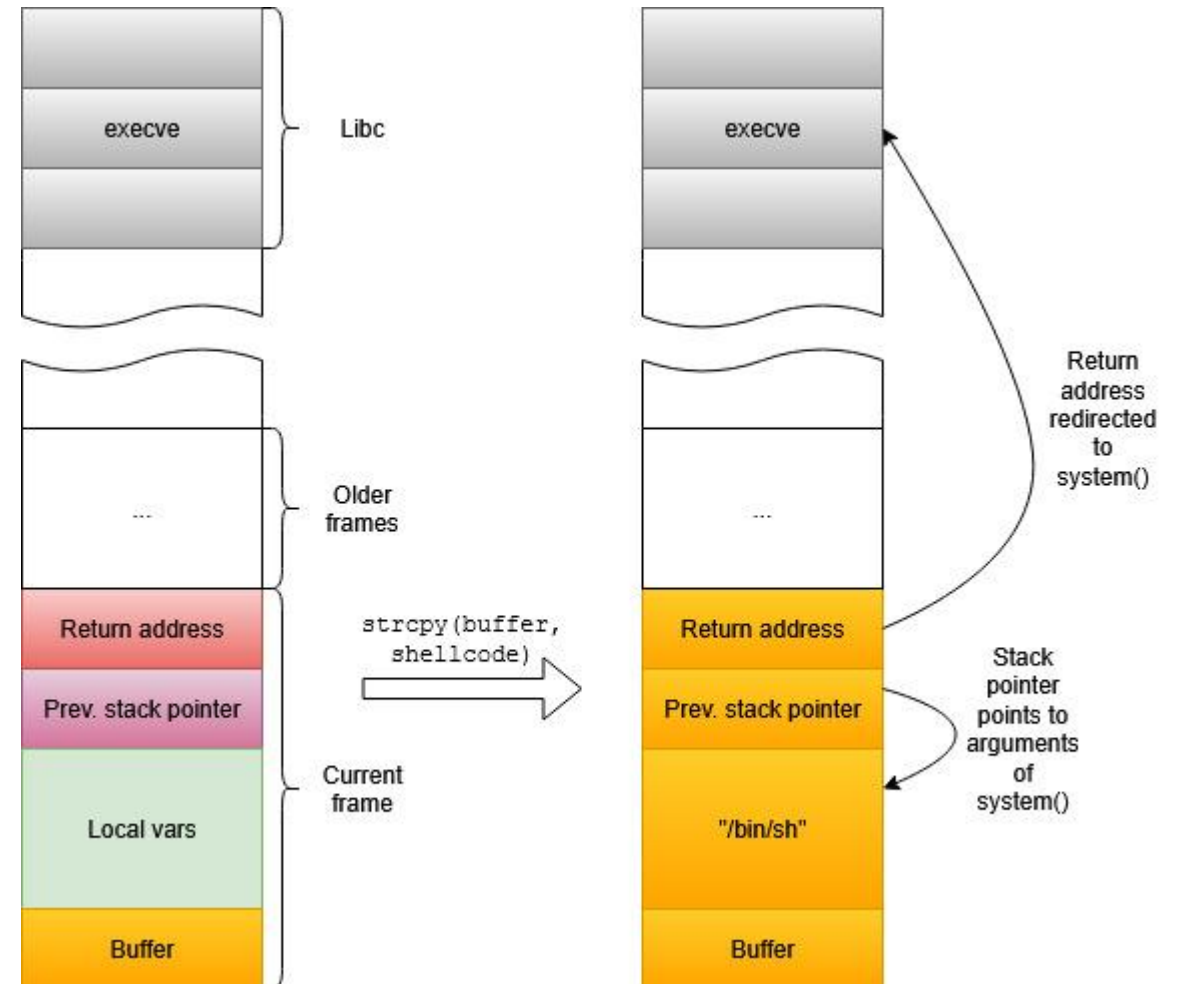
Атаки на стек: Stack Smashing

- Smashing The Stack For Fun And Profit (Aleph One, 1996)
- Запись вредоносного кода на стек и его вызов при возврате функции
- Неактуальна из-за современных защит



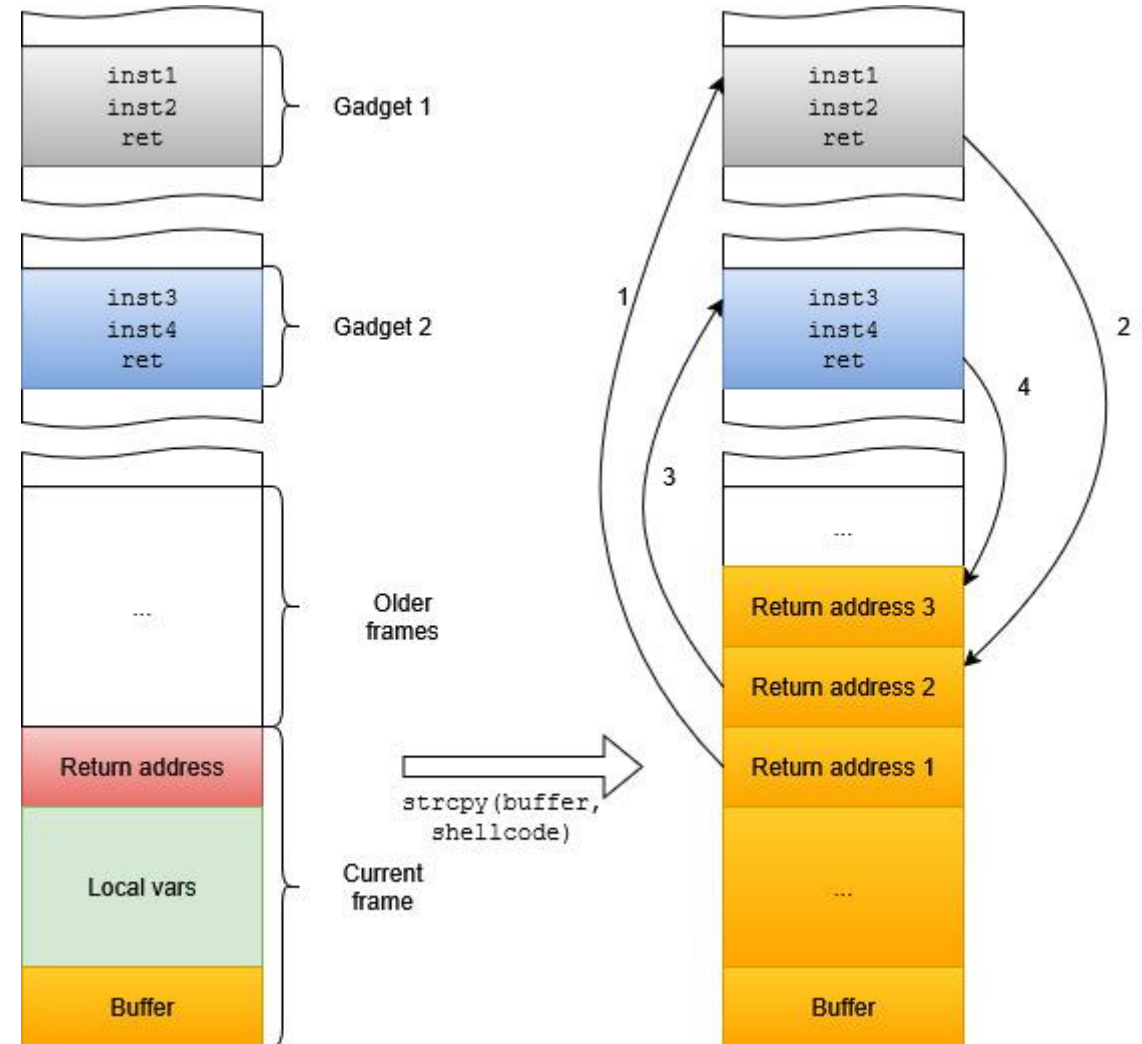
Атаки на стек: Return-to-libc

- Solar Designer, 1997
- Вызов при возврате из функции стандартной библиотечной процедуры
 - Обычно `execve("bin/sh")` или `system("/bin/sh")`
- Вариант атаки: return-to-plt
- Работала на 32-битном x86 (аргументы на стеке)



Атаки на стек: Return-oriented programming

- Nergal, 2001 и Shacham, 2007
- Наиболее актуальная проблема
- Сборка программы из эпилогов различных функций
- Запись на стек множества адресов возврата



Пример: Stack Smashing

```
int main() {
    char buf[32];
    strcpy(buf, code);
}

// Входная строка пользователя
const char code[] =
    "\x31\xc0"    // xorl %eax, %eax
    "\x50"        // pushl %eax
    "\x68"//"sh"   // pushl $0x68732f2f
    "\x68"//"bin"  // pushl $0x6e69622f
    ...
    PAD
    // Return address can be obtained with returns below
    "\x0c\xde\xff\xff"
;
```

```
$ CFLAGS='-Wl,-z,execstack -fno-stack-protector -w'
$ PAD=
$ for i in `seq 1 128`; do
    echo PAD=$i
    gcc -m32 -DPAD="$\"$PAD\"" -march=i686 $CFLAGS repro.c
    setarch -R env -i ./a.out
    PAD="$PAD\\xff"
done

PAD=1
PAD=2
PAD=3
...
PAD=20
Segmentation fault (core dumped)
PAD=21
Segmentation fault (core dumped)
...
PAD=25
$ # Получен доступ к shell
```

Атаки на кучу

- Эксплуатируют ошибки типа Heap Overflow
 - Переполнение буфера в куче
 - Более сложные и разнообразные чем Stack Overflow
- Основные типы:
 - Испортить данные в несвязанном буфере (например указатели на функции или vtables)
 - Испортить метаданные аллокатора
 - Заставить его при вызове несвязанного malloc/free писать по контролируемому адресу
 - Например испортить адрес malloc hook и вызвать его при следующем malloc (атака House of Force)



<https://kingsvilletimes.ca/2022/10/common-sense-health-rake-up-the-leaves-this-fall/>

Распространённость buffer overflow уязвимостей



- Лидирующие позиции в рейтинге наиболее опасных уязвимостей
 - [Mitre CWE Top 25 2024](#) (места 2, 6, 8, 20)
- 70% уязвимостей в продуктах Microsoft вызваны ошибками работы с памятью
 - [MSRC Blog: A proactive approach to more secure code](#)
- 70% high/critical багов в проекте Chromium – ошибки памяти
 - [Chromium Security: Memory Safety](#)
- 40% атак, вызванных ошибками работы с памятью, вызваны buffer overflow
 - [Google Project Zero](#)
- 11% CVE и 6.5% KEV в 2024
 - 20% из них это stack overflow (наиболее опасная уязвимость)
 - Не лучшая метрика (большая часть CVE это уязвимости веб-приложений)

Методы обнаружения на этапе QA

- AddressSanitizer
 - Stack/heap/static overflow, double free, use-after-free/return, etc.
 - State-of-the-art
 - Может ограниченно использоваться в проде для А/В тестирования
 - Особенно варианты с низкими накладными расходами (GWP-Asan, HWASan)
- Отладочные проверки STL
 - Например `-D GLIBCXX_DEBUG` в `libstdc++` или `-D _LIBCPP_ABI_BOUNDED_ITERATORS` в `libc++`
 - Меняют ABI => требуется полная пересборка зависимостей
- Valgrind
 - Только ошибки кучи: heap overflow, double free, use-after-free, etc.
 - Намного медленнее Asan, но может найти доп. ошибки
- Другие инструменты
 - ElectricFence (только heap overflow), [DirtyFrame](#), etc.

Неисполняемый стек

Введение

- W^X / NX bit / Data Execution Prevention
 - Отключение права на исполнения кода в сегменте стека
 - Осуществляется на уровне OS
 - Также применяется ко всем writable-сегментам (куче и глобальным переменным)
- Одна из первых hardening защит
 - Впервые появилась в OpenBSD (2003) и Windows (2004)
 - Полностью исключает (приведённой выше) атаки Stack Smashing
- Включена по умолчанию во всех современных дистрибутивах (GCC, Clang) и Windows

Проблемы

- Требуется чтобы весь код программы был собран в режиме неисполняемого стека
 - В том числе статически связанные динамические библиотеки
 - Но не загруженные динамически с помощью `dlopen` ([BZ #32653](#))
 - Линкер предупредит при сборке
 - Рекомендуется использовать `LDFLAGS += -Wl,--fatal-warnings`
 - Основные причины `execstack` в коде:
 - Забыли проаннотировать ассемблерный код
 - Использование указателей на GNU nested functions
- Накладные расходы отсутствуют

Address Space Layout Randomization (и PIE)

Введение

- Address Space Layout Randomization
 - Рандомизация расположения основных сегментов программы (стека, кучи, библиотек)
 - Осуществляется на уровне ОС (рандомизация mmap)
 - Лишает хакера знания о том какие адреса возврата использовать в Stack Overflow-атаках
- Сильно снижает риски любых buffer overflow атак (return-to-libc, ROP, heap overflow, etc.)
 - Пример Stack Smashing стабильно падает с Segmentation fault
- Одна из первых hardening-защит:
 - PaX патч, 2001
 - Linux, 2005
 - Windows, 2007 (Vista)
 - Оверхед для 32-битных Windows намного выше из-за архитектуры DLL

Position-independent Executable (PIE)

- Необходим для так называемого Full ASLR:
 - Сборка основной программы в специальном режиме PIE
 - Сгенерированный компилятором код не использует абсолютные адреса
 - Это позволяет ОС размещать программу по случайному адресу
- Включена по умолчанию в Ubuntu/Debian (GCC и Clang) и Windows
 - Но не во всех дистрибутивах (например Fedora)
 - Некоторые критические программы в Debian собраны без PIE
 - `/usr/bin/python3`
 - Рекомендуется указывать принудительно флагами `-fPIE -pie`
 - `-fPIE` \sim `fPIC` + `-fno-semantic-interposition` + `-Bsymbolic`

Накладные расходы

- Накладные расходы на современных архитектурах ничтожны
 - Не удалось обнаружить замедления при компиляции CGBuiltin.cpp компилятором Clang
 - На 32-битном x86 замедление до 20%
 - [Too much PIE is bad for performance](#)
- ASLR несовместима с предлинковкой (prelinking) библиотек для ускорения загрузки
 - [C++Russia: Динамические библиотеки и способы ускорения их работы](#)

Недостатки: false negatives (1)

- Уязвимость к info leakage attacks (например [Format string attacks](#)):
 - Рандомизируется только базовый адрес приложения/библиотек
 - Хакер знает относительные смещения кода, глобальных переменных, таблиц GOT/PLT
 - Если становится известен адрес хотя бы одной сущности – защита скомпрометирована
 - В частности использование fork компрометирует ASLR (Zygote-процесс в Android)

Недостатки: false negatives (2)

- Недостаточная рандомизация
 - Рандомизируется только база mmap-адресов (delta_mmap)
 - Относительный библиотек и mmap-регионов фиксирован
 - Android рандомизирует порядок загрузки библиотек и промежутки между ними
 - Небольшое число рандомизируемых битов
 - 16 или даже 8 в 32-битных Windows и ранних Android
 - Не все биты адреса одинаково случайны
 - Windows
 - Рандомизация каждого приложения делается однократно при его первой загрузке (для ускорения)
 - Одна и та же библиотека может грузиться по одному адресу в разных приложениях (для ускорения)
 - Linux
 - Рандомизация делается однократно при старте сервиса
 - Уязвима к brute force (особенно на 32-битных платформах)
- Рекомендуется делать регулярный рестарт сервисов !

Stack Protector

Stack Protector

- Суть Stack Overflow атак – модификация адреса возврата
- Идея:
 - Разместить перед адресом неизвестное хакеру число (stack canary, stack cookie)
 - Перед возвратом из функции проверять что канарейка не поменялась
 - При переполнении нельзя изменить адрес возврата, не поменяв канарейку
- Одна из первых hardening-защит:
 - StackGuard (1997)
 - ProPolice (2001, IBM)
 - StackProtector (2005, RedHat), StackProtectorStrong (2012, Google)
- Сильно снижает риски stack overflow атак (return-to-libc, ROP)
 - Пример Stack Smashing стабильно падает с

```
*** stack smashing detected ***: terminated
Aborted (core dumped)
```



David & Angie,
<https://www.flickr.com/photos/studiomiguel/3946174063>

Дополнительные меры безопасности

- Скалярные переменные кладутся ниже по стеку чем массивы
 - Чтобы при переполнении массива нельзя было модифицировать флаги, адреса функций и т.п.
- Один из байтов канарейки всегда нулевой (чтобы остановить строковый buffer overflow)

Недостатки

- Существенные накладные расходы:
 - Загрузка значения канарейки, сохранение на стек, чтение и проверка перед возвратом
 - 2% при компиляции CGBuiltin.cpp компилятором Clang
 - [The Performance Cost of Shadow Stacks and Stack Canaries](#): 0-9%
- False negatives:
 - Уязвима к info leakage: если канарейка утекла, то защита скомпрометирована
 - Если канарейка хранится в том же сегменте что и стек, хакер может переписать и её
 - Не защищает от переписывания пользовательских указателей на функции на стеке (или библиотечных, например atexit handlers)
 - Не защищает от переписывания адреса возврата без overflow

Разделение стека

Введение

- SafeStack, ShadowStack, backward-edge CFI
 - Основная причина stack overflow – адрес возврата хранится вместе с локальными массивами
 - Можно разделить стек на две несвязные части:
 - адрес возврата (и в случае SafeStack скалярные переменные, адрес которых не берётся)
 - все остальные
- Первое найденное упоминание: StackShield (~2000)
- Сравнение со StackProtector:
 - Дополнительная рандомизация для критических данных
 - Позволяет защитить пользовательские указатели на функции на стеке
 - StackProtector по прежнему может применяться к unsafe stack для обнаружения overflow



<https://picryl.com/media/reaching-shadow-heart-nature-landscapes-d62bda>

Недостатки

- Производительность:
 - 3% при компиляции CGBuiltin.cpp компилятором Clang
 - 0.1% по замерам авторов ([Clang documentation: SafeStack](#))
- False negatives:
 - SafeStack сейчас не поддерживает instrumentation динамических библиотек (по идее этого легко добавить: [OpenSSF #267](#))
 - ShadowStack:
 - Поддерживает только AArch64 и RISC-V
 - Защищает только адреса возврата

Stack Clashing (Stack Probes)

Методы hardening: Stack Clashing

- Стек отделён от других сегментов незамапленной страницей (guard page)
 - Обнаруживает исчерпание стека
 - Появилась в Linux в 2010
- Проблема:
 - Не обнаружит переполнение при больших локальных массивах (>4096 байт) или alloca
 - Хакер может перезаписать кучу или стек другого потока
- Уязвимость обнаружена группой Qualys в 2017:
 - [The Stack Clash](#) (10 proof-of-concept атак)
- Идея:
 - Перед выполнением функции пройти по новому фрейму, чтобы спровоцировать SEGV

Недостатки

- Накладные расходы минимальны:
 - Нет замедления при компиляции CGBuiltin.cpp компилятором Clang
 - Не обнаружены регрессии в Firefox
 - [Bringing Stack Clash Protection to Clang / X86](#)

Фортификация
(_FORTIFY_SOURCE)

Пример защиты

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

unsigned n = 4096;

int main() {
    char *a = malloc(1);

    memset(a, 0, n);
    asm("" :: "r"(&a) : "memory");

    a = malloc(200);
    asm("" :: "r"(&a) : "memory");

    return 0;
}
```

```
# No _FORTIFY_SOURCE
$ gcc -U_FORTIFY_SOURCE tmp5.c -O2 && ./a.out

Fatal glibc error: malloc.c:2599
(sysmalloc): assertion failed:
(old_top == initial_top (av) &&
old_size == 0) || ((unsigned long)
(old_size) >= MINSIZE && prev_inuse
(old_top) && ((unsigned long) old_end
& (pagesize - 1)) == 0)

Aborted (core dumped)

# _FORTIFY_SOURCE=3
$ gcc tmp5.c -O2 && ./a.out
*** buffer overflow detected ***:
terminated

Aborted (core dumped)
```

Реализация

- Из Glibc string.h:

```
#if _FORTIFY_SOURCE > 0
__attribute__((always_inline, __nothrow__, leaf))
void *memset (void *__dest, int __ch, size_t __len)
{
    // memset_chk определена в libc.so.6 и содержит проверку диапазона
    return __builtin___memset_chk (__dest, __ch, __len,
                                    __glibc_objsize0 (__dest));
}
#endif
```

- `__glibc_objsize0` вызывает интринсик компилятора `__builtin_object_size` или `__builtin_dynamic_object_size` (в зависимости от уровня защиты)
 - `__builtin_object_size` проверяет указатели на стековые объекты
 - `__builtin_dynamic_object_size` осуществляет dataflow-анализ и применима например к объектам кучи

Введение

- Проверки диапазонов в функции стандартной библиотеки C (там где это возможно)
 - Появились в Glibc 2.3.4 (2004)
- Конкретный список проверяемых функций можно уточнить в Glibc headers (~80 функций)
 - string.h APIs (memcpy, memset, strcpy, strcat, bzero, bcopy, etc.) - проверки диапазона
 - unistd.h APIs (read, pread, readlink, etc.) - проверки диапазона
 - printf and friends - %n допускается только в readonly-строках
- Защищает от stack и heap buffer overflow
- Требуется совместной работы
 - библиотеки (подмена стандартной функции на chk-версию)
 - компилятора (вычисление размера из контекста)

Недостатки

- Накладные расходы:
 - -D_FORTIFY_SOURCE=2: нет изменений при компиляции CGBuiltin.cpp компилятором Clang
 - -D_FORTIFY_SOURCE=3: 2% при компиляции CGBuiltin.cpp компилятором Clang
 - 3% на ffmpeg ([FORTIFY_SOURCE and Its Performance Impact](#))
- Конфликтует с Address- и MemorySanitizer:
 - Asan не умеет анализировать XXX_chk-функции ([sanitizers #247](#))
 - Совмещение Asan с фортификацией приводит к false negatives (пропуску ошибок)
 - GCC (не Clang) вставляет доп. минимальную instrumentation в месте вызова для memcpu_chk, memset_chk, но её недостаточно
 - Из-за того что фортификация включена по умолчанию во многих дистрибутивах лучше явно отключать её в санитарных сборках:
 - -U_FORTIFY_SOURCE или -D_FORTIFY_SOURCE=0
- Поддержана только в Glibc и Bionic (не в musl или Visual Studio)
 - Есть standalone реализация [fortify-headers](#)
- Работает только в -O режиме и только если подключены стандартные .h файлы (нет implicit declarations)
- Компилятор не всегда может вывести допустимый размер указателя из контекста
 - Ограничен рамками функции

-fsanitize=bounds

- Подход фортификации можно расширить на скалярные обращения к массивам известной длины
- Опция `-fsanitize=bounds` в компиляторах GCC и Clang
 - Аналог `-D_FORTIFY_SOURCE=2`: массивы константных размеров или VLA
- Включена в Android для некоторых критичных модулей
 - [Android Developers Blog: System hardening in Android 11](#)
- Нет накладных расходов при компиляции `CGBuiltin.cpp` компилятором Clang
 - Аналогично `-D_FORTIFY_SOURCE=2`

Проверки STL

Пример

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

int main() {
    std::vector<int> v;
    asm("" :: "r" (&v) :
"memory");
    return v[4096];
}
```

```
$ g++ tmp.cc
$ ./a.out
Segmentation fault
```

```
$ g++ -D_GLIBCXX_ASSERTIONS
tmp.cc
$ ./a.out
/usr/include/c++/12/bits/stl_
vector.h:1123: ... :
Assertion 'n < this-
>size()' failed.
Aborted
```

Введение

- Hardened STL
- Конкретные проверки зависят от компилятора и уровня защиты
 - Всегда включены проверки индексов (а также `front`, `back`, etc.) в `std::vector` и `std::string`
 - Защищают от ошибок `buffer overflow`
 - GCC:
 - Проверки на NULL в умных указателях (защита от NULL dereference)
 - Проверки корректности параметров мат. функций и распределений
 - Множество других мелких проверок типа `abs(INT_MIN)` в `std::gcd`
 - LLVM:
 - Например проверки на Strict Weak Ordering компараторов
 - [C++Russia: Как правильно писать компараторы](#)
 - Visual Studio:
 - Аналогичные проверки [имеют слишком большой оверхед](#) и [их планируют переписать](#)

История и будущее

- Хронология:
 - Впервые появились в GCC debug containers (начало 2000-х)
 - Опция `-D_GLIBCXX_ASSERTIONS` для hardening в GCC (2015)
 - Аналогичная проверка в libc++ и Safe Buffers proposal (2022)
- В будущем STL hardening скорее всего станет частью Стандарта C++
 - Через механизм C++ profiles
 - Дефолтным будет профиль, запрещающий работу с raw pointers
 - Инструменты для миграции на `std::span` уже существуют и применяются:
[Safe Buffers](#)

Недостатки

- Накладные расходы:
 - 3.5% при компиляции CGBuiltin.cpp компилятором Clang
 - 0.3% в серверных приложениях Google
 - [Retrofitting spatial safety to hundreds of millions of lines of C++](#)
 - [Только при условии включённых ThinLTO и PGO](#), иначе [1-2%](#)
- False negatives:
 - Покрывает только подмножество ошибок (некорректные индексы, только STL)
 - Некоторые ошибки обнаруживать слишком дорого (например ошибки в итераторах)

Усиленные аллокаторы

Пример ошибки (1)

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

```
void *a, *b;
unsigned n = 4096;
```

```
int main() {
    a = malloc(100);
    memset(a, 0xff, n);
    b = malloc(100);
}
```

```
$ gcc -O2 repro.c
```

```
$ ./a.out
```

```
malloc(): corrupted top
size
```

```
Aborted
```

Пример ошибки (2)

```
#include <stdlib.h>

void *a, *b;

int main() {
    a = malloc(1);
    free(a);
    b = malloc(1);
    // Ошибка copy-paste
    free(a);
    return 0;
}
```

```
$ gcc -O2 repro.c

# Glibc не видит ошибку
$ ./a.out

# Hardened-аллокатор
$
LD PRELOAD=libhardened_malloc.so ./a.out
fatal allocator error:
double free (quarantine)
Aborted
```

Введение

- Дополнительные меры в динамическом аллокаторе для затруднения атак на метаданные аллокатора
- Защита от ошибок кучи (heap overflow, double free, use-after-free, free of invalid address)
 - Метаданные физически отделены от аллоцируемой памяти (нет "хедеров")
 - Рандомизация адресов внутри блоков
 - Чексуммы и/или канарейки для обнаружения перезаписи метаданных
 - Карантин (отложенное переиспользование освобождённой памяти)
 - Зануление данных на free и проверка на malloc

Недостатки

- Накладные расходы:
 - 9% при компиляции CGBuiltin.cpp компилятором Clang (hardened_malloc vs Glibc allocator)

Защита таблиц диспетчеризации (Full RELRO)

Введение

- Вызовы функции из динамических библиотек делаются через специальные трамплины (PLT stubs)
- Функции-трамплины читают и обновляют таблицу GOT, содержащую указатели на функции
 - Т.н. отложенное связывание (lazy binding)
 - Ускоряет запуск приложения
- Таблицу приходится держать в writable-сегменте и у хакеров есть возможность её скомпрометировать
 - Более редкая атака чем buffer overflow (мне неизвестны соответствующие CVE)
- Решение (read-only relocations, RELRO):
 - Инициализировать содержимое таблицы на старте программы и сразу пометить сегмент как readonly

Пример

```
#include <stdio.h>

void shellcode() {
    printf("You have been pwned%s\n", "");
}

extern void *_GLOBAL_OFFSET_TABLE_[];

int main() {
    // Имитируем действия хакера
    _GLOBAL_OFFSET_TABLE_[POS] = shellcode;

    puts("Hello world!\n");
    return 0;
}
```

```
$ for i in `seq 0 16`; do
    gcc -Wl,-z,norelro repro.c -
DPOS=$i
    ./a.out
    i=$((i + 1))
done
Segmentation fault
Segmentation fault
Segmentation fault
You have been pwned
Hello world!
Hello world!
Hello world!
...
```

История

- Подход RELRO уже использовался ранее для инициализации vtables (partial RELRO)
 - [Ian Lance Taylor: Linker relro](#)
- Потребовалась лишь небольшая адаптация для GOT (full RELRO)

Недостатки

- Практически не влияет на производительность
 - Не обнаружили никакого замедления в работе компилятора Clang
 - Но может только замедлить старт программы из-за необходимости разрешения всех символов
 - На X86 имеет смысл совмещать с `-fno-plt` (до 10% прироста производительности)
- False positives:
 - Могут сломаться некоторые программы, если в них были отсутствующие символы (которые не вызывались)
- False negatives:
 - Не защищает пользовательские таблицы функций (и библиотечные, например `atexit handlers`)

Автоинициализация

TODO: понятный
пример

Пример

- Какой код может сгенерировать компилятор для этой программы?

```
int foo() {  
    int x;  
    return x ? 1 : 1;  
}
```

TODO: понятный
пример

Пример

- Какой код может сгенерировать компилятор для этой программы?

```
int foo() {  
    int x;  
    return x ? 1 : 1;  
}
```

```
int foo() {  
    return 1;  
}
```

```
int foo() {  
}
```

```
int foo() {  
    abort();  
}
```

Введение

- Инициализация всех локальных переменных
 - Случайными значениями для debug, нулями для hardening
- История:
 - В коммерческих тулчейнах автоинициализация появилась давно
 - InitAll добавлен в Visual Studio в 2019
 - [CppCon 2019: Killing Uninitialized Memory](#)
 - Решение в GCC в 2021
 - [Первое обсуждение](#) в mailing list в 2014
 - Планируется включить в Стандарт C++26 ([P2795](#), см. ниже)
- Распространённость:
 - 10% CVE root cause в продуктах Microsoft в 2018 (из [Killing Uninitialized Memory](#))
 - 12% exploitable багов в Android (из [P2723](#))

Накладные расходы

- Существенные накладные расходы:
 - Замеры:
 - 4.5% при компиляции CGBuiltin.cpp компилятором Clang
 - 1% на Firefox (из [Trivial Auto Var Init Experiments](#))
 - До 10% в горячем коде ([virtio](#), [Chrome](#))
 - 1-3% в среднем на Postgres, но до 20% на некоторых сценариях ([Ubuntu #1972043](#))
 - <1% в Windows ([Killing Uninitialized Memory](#))
 - Основной проблемный кейс: большой локальный массив (например для IO) на горячем пути

```
while (std::getline(maps, line)) {
    char modulePath[PATH_MAX + 1] =
        "";
    // -ftrivial-auto-var-init
    // вставит здесь memset...
    ret = sscanf(line.c_str(),
                 "%lx-%lx %6s %lx %*s
%*x %" PATH_MAX_STRING(PATH_MAX)
                 "s\n",
                 &start, &end, perm,
                 &offset, modulePath);
}
```

Другие недостатки

- Автоинициализация ломает обнаружение багов в Valgrind и Msan
 - Необходимо обязательно отключать её в соответствующих сборках !
 - По крайней мере флаг сохраняет предупреждения компилятора (-Wuninitialized -Wmaybe-uninitialized)
- В некоторых ситуациях может привести к дополнительным уязвимостям:
 - Инициализация нулями ([обычно рекомендуется](#) для прода): в Linux “0” это например id суперпользователя
 - Инициализация не-нулями: провоцирование buffer overflow
- Применяется только к локальным переменным
 - Глобальные и так инициализируются
 - Для кучи можно использовать hardened allocators

Проверка целочисленных переполнений

Пример ошибки

```
// Из OpenSSH 3.3
nresp = packet_get_int();
if (nresp > 0) {
    // Переполняем целое число до нуля здесь ...
    response = xmalloc(nresp*sizeof(char*));
    // ... и вызываем heap buffer overflow тут
    for (i = 0; i < nresp; i++)
        response[i] = packet_get_string(NULL);
}
```

Введение

- Проверки целочисленных операций на переполнение
 - Дефолтный рантайм UBsan выдаёт слишком много отладочной информации поэтому не подходит для hardening
 - Выход – использование специального минимального рантайма (с immediate abort)
- Критичность:
 - Наиболее известные примеры:
 - Инцидент с облучателем Therac-25 (1985)
 - Катастрофа ракеты Ariane 5 (1996)
 - ~1% CVE и 1.5% KEV в 2024
 - 23 место в рейтинге [Mitre CWE Top 25 2024](#) (8 в [рейтинге 2019 года](#))
- История:
 - -ftrapv появилась в GCC в 2000 ([patch for -ftrapv option.](#))
 - За фичей не следили и она быстро протухла (например [BZ #35412](#) открыт в 2008)
 - Работы John Regehr в [2010](<https://blog.regehr.org/archives/1559>)
 - Создание UBsan в 2014 (на волне популярности Asan)
 - State-of-the-art

Недостатки

- Накладные расходы:
 - 30% замедление при компиляции CGBuiltin.cpp компилятором Clang
 - До 2х на SPEC (из [статьи про PartiSan](#))
 - Проверка переполнений дефолтно отключена в Rust
- Другие проблемы:
 - UBSan несовместим с `-fno-strict-overflow` и `-fwrapv`
 - False positives:
 - Isan может выдавать ложные срабатывания (например нужен blacklist для STL-кода, полагающегося на переполнение)
 - False negatives:
 - Может не обнаруживать некоторые баги, которые успел "перехватить" оптимизатор (особенно под `-O2`)

Отключение небезопасных оптимизаций

Пример ошибки

```
static void
__devexit agnx_pci_remove (struct pci_dev *pdev)
{
    struct ieee80211_hw *dev = pci_get_drvdata(pdev);
    struct agnx_priv *priv = dev->priv;

    // Компилятор удалил проверку
    if (!dev) return;

    ... do stuff using dev ...
}
```

Введение

- Некоторые компиляторы могут излишне агрессивно реагировать на код, содержащий неочевидные для программиста ошибки, и генерировать небезопасный ассемблер
 - В основном выбрасываются пользовательские проверки
 - Visual Studio менее агрессивен чем GCC/Clang
- Compiler Introduced Security Bugs
 - Термин появился в статье [Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs](#)
 - Соответствующих CVE мало (например [CVE-2009-1897](#))
 - Но в статьях находят сотни CISB в open-source коде
- Для кода с повышенными требованиями безопасности рекомендуется отключать такие оптимизации

Накладные расходы

- 4.5% оверхед при компиляции CGBuiltin.cpp компилятором Clang
- Слабый (до 1%) оверхед для Phoronix Test Suite
 - [Performance Impact of Exploiting Undefined Behavior in C/C++](#)

Как использовать ?

- Обычно для GCC/Clang отключают
 - `-fno-delete-null-pointer-checks`
 - `-fno-strict-overflow` (`== -fwrapv -fwrapv-pointer`)
 - `-fno-strict-aliasing`
- Соответствующие баги можно также обнаруживать с помощью UBSanitizer и TypeSanitizer
- Использование:
 - Флаги по умолчанию выключены во всех компиляторах и дистрибутивах
 - Но многие пакеты в дистрах собираются по крайней мере с `-fno-strict-aliasing`
 - Т.к. правила алиасинга особенно легко нарушить
 - Chrome собирается со всеми тремя флагами
 - build/config/compiler/BUILD.gn
 - Firefox [собирается](#) с `-fno-strict-aliasing`

Control-Flow Integrity

Пример

```
include <stdio.h>

struct A { virtual void foo() {} };
struct B : A { void foo() override {} };

struct Evil { virtual void foo() {
    printf("You have been pwned\n"); }
};

A *tmp = new B;

int main() {
    A *a = new A;
    Evil *e = new Evil;
    asm("mov %1, %0" : "+r"(a) : "r"(e));
    a->foo();
}
```

```
# Подмена объекта
$ clang++ repro.cc -O2
$ ./a.out
You have been pwned
```

```
# CFI обнаруживает подмену
$ clang++ -fsanitize=cfi -flto
-fvisibility=hidden repro.cc -
O2
$ ./a.out
Illegal instruction
```

История

- CFI это generic-термин для любых нарушений исходного control-flow программы
 - Впервые введён Abadi et al. в 2005 ([CFI: Principles, Implementations and Applications](#))
- Два типа:
 - forward-edge (проверка call/jump)
 - backward-edge (проверка return)
- Множество различных методик в статьях
 - В широком смысле Stack Protector и Shadow Stack – тоже CFI
- Но обычно под CFI понимают один из методов:
 - LLVM CFI, 2015 (2015, Clang 3.7)
 - [Microsoft Control Flow Guard](#), 2014
 - [grsecurity RAP](#), 2016
 - Аппаратные методы: Intel CET, 2020 (спецификация 2016) и AArch64 BTI/PAC (2018)

LLVM CFI

- Компиляторная инструментация для forward-edge проверок
- Реализована только в Clang (не поддерживана в GCC)
- Проверяет совпадения статического и динамического прототипа при вызове функции по указателю
 - Поддерживаются vtables и обычные указатели на функции
 - (алгоритмы проверки сильно различаются)
- Также может использоваться для доп. проверок (корректность C++ кастов и пр.)

Аппаратные методы: Intel CET и AArch64 CFI

- Поддержаны в GCC и Clang
- Более грубые проверки чем LLVM CFI
- Проверки адресов возврата:
 - Все места, на которые может быть косвенный переход (бранч/вызов/возврат), помечаются инструкцией-хинтом ENDBR64
- Pointer Authentication (AArch64)
 - Верхние биты адреса возврата используются для вычисления криптостойкой чексуммы
 - Адрес возврата + адрес фрейма + секрет процесса
 - Чексумма проверяется перед возвратом

Накладные расходы

- Оверхед
 - Компиляция CGBuiltin.cpp компилятором Clang: нет изменений при Intel CET, 6% LLVM CFI
 - Менее 1% в Chrome ([Chrome: Control Flow Integrity](#)), но 10% увеличение кода (I\$, BTB)
 - Нет оверхеда на Android при LLVM CFI ([Android: Security: Control flow integrity](#))

Недостатки

- Фрагментация: три несвязанных решения с разными, GCC не поддерживает LLVM CFI
- False positives:
 - Большое количество софта надо дорабатывать для LLVM CFI (всевозможные `reinterpret_cast<void*>`, etc.)
 - Например Clang не проходит проверки без фильтров
- False negatives:
 - LLVM CFI:
 - Только несоответствия на уровне типов (хакер может вызвать неправильную функцию если типы совпадают)
 - Тяжелая интеграция (требует LTO, проблемы с проверкой вызовов между границами DSO)
 - Intel/AArch64: вообще не проверяет типы
 - Не проверяются jump tables, сгенерированные для switch-конструкций (только в CET есть `-mcet-switch`, дефолтно выключен)

Опции для включения защит

Защита	Флаги
Noexecstack	Включена по умолчанию
Full ASLR (PIE)	-fPIE -pie
Stack Protector	-fstack-protector-strong
Safe Stack	-fsanitize=safe-stack
Stack Clashing	-fstack-clash-protection
_FORTIFY_SOURCE=2	-D_FORTIFY_SOURCE=2 (или 3), -fsanitize=bounds
STL hardening	-D_GLIBCXX_ASSERTIONS (libstdc++), -D_LIBCPP_HARDENING_MODE=... (libc++)
Hardened allocator	LD_PRELOAD=path/to/allocator.so MALLOC_CHECK_=3 или GLIBC_TUNABLES=glibc.malloc.check=3 (Glibc)
Full RELRO	-Wl,-z,relro -Wl,-z,now
Autoinitialization	-ftrivial-auto-var-init=zero
Integer Overflow	GCC: -fsanitize-trap=signed-integer-overflow,pointer-overflow Clang: -fsanitize=signed-integer-overflow,pointer-overflow -fsanitize-minimal-runtime (рекомендуется также integer)
Отключение оптимизаций	-fno-delete-null-pointer-checks -fno-strict-overflow(== -fwrapv -fwrapv-pointer) -fno-strict-aliasing
Control-flow integrity	LLVM: -fsanitize=cfi -flto=thin -fvisibility=hidden -fsanitize=cfi-cross-dso Intel CET: -fcf-protection AArch64: -mbranch-protection=standard (непонятно почему не под -fcf-protection)

Использование в реальном
коде

Дистрибутивы Linux

TODO: Android

Защита	Ubuntu 24.04			Debian 12			Fedora 42		
	GCC	Clang	Пакеты	GCC	Clang	Пакеты	GCC	Clang	Пакеты
Noexecstack	Y	Y	Y	Y	Y	Y	Y	Y	Y
Full ASLR (PIE)	Y	Y	Y	Y	Y	Y	N	N	Y
Stack Protector	Y	N	Y	N	N	Y	N	N	Y
Safe Stack	N	N	N	N	N	N	N	N	N
Stack Clashing	Y	N	Y	N	N	N*	N	N	Y
_FORTIFY_SOURCE	Y (2)	N	Y (2)	N	N	Y (2)	N	N	Y (3)
STL hardening	N	N	N	N	N	N	N	N	Y (libstdc++)
Hardened allocator	N	N	N	N	N	N	N	N	N
Full RELRO	Y	N	Y	N	N	Partial	N	N	Y
Autoinitialization	N	N	N	N	N	N	N	N	N
Integer Overflow	N	N	N	N	N	N	N	N	N
Отключение оптимизаций	N	N	N	N	N	N	N	N	N
Hardware CFI (Intel CET, AArch64 BP)	Y	N	Y	N	N	N*	N	N	Y

- Наборы защит в дистрибутивах отличаются
- В Clang включено намного меньше защит
- Многие новые защиты по умолчанию не включены
- Пакеты системы защищены лучше пользовательских программы
- Важно: дефолтные защиты могут быть отключены в конкретных пакетах (например [python3 в Debian 12](#))

* - будет включён в следующей версии Debian

Браузеры

Защита	Chrome (d0273f3d)	Firefox (b0ca903b)
Noexecstack	Y	Y
Full ASLR (PIE)	Y	Y
Stack Protector	Y (weak)	Y
Safe Stack	N	N
Stack Clashing	N	Y
_FORTIFY_SOURCE=2	Y	Y
STL hardening	Y (libstdc++)	N
Hardened allocator	Y	N
Full RELRO	Y	Y
Autoinitialization	Y	N
Integer Overflow	N	N
Отключение оптимизаций	Y	Y
CFI	Y (LLVM на X86, AArch64 CFI)	N

- Рассматривались дефолтные флаги Linux-версии

Использование в безопасных языках

Hardening в Rust

Защита	Актуальность	Статус в Rust	Замечания
Noexecstack	Только unsafe/внешний код	Включена	Prooflink
ASLR	Только unsafe/внешний код	Включена	Prooflink
Stack Protector	Только unsafe/внешний код	Nightly-опция	Prooflink
Safe Stack	Только unsafe/внешний код	Nightly-опция	Prooflink
Stack Clashing	Актуальна	Включена	Prooflink
_FORTIFY_SOURCE, STL hardening	Нет	Нет	Уже есть в языке
Hardened allocator	Только unsafe/внешний код	Есть биндинги к hardened_malloc	Prooflink
Full RELRO	Только unsafe/внешний код	Включена	Prooflink
Autoinitialization	Нет	N/A	Уже есть в языке
Integer Overflow	Актуальна	Есть опция (дефолтно отключена)	
CFI	Только unsafe/внешний код	Nightly-опция	Prooflink

О чём мы не рассказали

- Опции для очистки секретов (паролей, ключей, etc.):
 - Stack scrubbing – очистка стека при выходе из функции (`-fstруб`)
 - Очистка регистров при выходе из функции (`-fzero-call-used-regs`)
- Опции для защиты от аппаратных атак (Spectre, etc.)
- `-fhardened` – зонтичная опция для наиболее важных защит
 - Хороший дефолтный флаг, но пока реализован только в GCC ([LLVM #122687](#))
 - Включает все опции, рекомендованные OpenSSF
 - Конкретный набор зависит от версии компилятора (для GCC есть `--help=hardened`)
 - Сейчас: `-D FORTIFY_SOURCE=3 -D GLIBCXX_ASSERTIONS -ftrivial-auto-var-init=zero -fPIE -Wl,-z,nōw -Wl,-z,relro -fstack-protector-strong -fstack-clash-protection -fcf-protection=full`
- Hardening-защиты в критическом ПО
 - Чаты, почтовые клиенты, мультимедиа, интерпретаторы, БД, офисное ПО, ридеры

Заключение

Что стоит сделать ?

- Проверить дефолтные опции при сборке продуктового кода и дистрибутива
 - Решить с Security Team какие hardening-методы включить
- Поиск проблемных программ (no-pie, etc.) можно автоматизировать с помощью [утилиты checksec](#)
 - Может проверить наличие noexecstack, PIE, _FORTIFY_SOURCE, RELRO, etc.
 - Пока (?) не поддерживает более новые защиты: CFI (все виды), Stack Clashing, Safe Stack, etc.

Что почитать ?

- Примеры атак:
 - [Nightmare](#)
 - [Overview of GLIBC heap exploitation techniques](#)
- Руководства по hardening
 - [OpenSSF Compiler Options Hardening Guide for C and C++](#)
 - [Linux Hardening Guide](#)
- Статьи John Regehr про UB
 - [A Guide to Undefined Behavior in C and C++](#)
 - [UB in 2017](#)

Спасибо за внимание !

- Полная версия слайдов доступна по адресу
 - <https://github.com/yugr/slides/blob/main/CppZeroCost/2025/RU.pptx>
- Вопросы ?



Приложения

Воспроизведение результатов

- Версии дистрибутивов:
 - Проверялись последние *стабильные* версии
 - Debian 12 (bookworm), Fedora 42, Ubuntu 24.04 (noble)
- Версии браузеров:
 - Firefox: <https://github.com/mozilla-firefox/firefox> (коммит b0ca903b)
 - Chrome: <https://chromium.googlesource.com/chromium/src> (коммит d0273f3d)
- Замеры производительности Clang:
 - <https://github.com/yugr/slides/tree/main/CppZeroCost/2025/bench>
- Подсчёт CVE/KEV-метрик:
 - <https://github.com/yugr/slides/tree/main/CppZeroCost/2025/scripts>
- Пруфлинки и дополнительная информация доступны в файле
 - <https://github.com/yugr/slides/blob/main/CppZeroCost/2025/plan.md>

Stack Protector: Как включить ?

- Включен по умолчанию только в компиляторе Ubuntu GCC (нет в Fedora и Debian, нет в Clang) и Windows
 - Рекомендуется явно указывать флаг `-fstack-protector-strong`
 - Пакеты в Debian, Fedora, Ubuntu собираются с этим флагом
- Включён в релизной сборке Firefox
- В Chrome включён слабый вариант Stack Protector

Safe Stack: Как включить ?

- Несколько реализаций:
 - SafeStack: `-fsanitize=safe-stack` (наиболее распространённый флаг)
 - Intel CET Shadow Stack: не реализован (нельзя включить ни по `-fcf-protection`, ни по `-mshstk`)
 - ShadowCallStack: `-fsanitize=shadow-call-stack`
- Защита не включена по умолчанию в дистрибутивах и браузерах Chrome/Firefox

Stack Clashing: Как использовать ?

- Включен по умолчанию только в компиляторе Ubuntu GCC (нет в Fedora и Debian, нет в Clang)
 - Рекомендуется явно указывать флаг `-fstack-clash-protection`
 - Пакеты в Debian, Fedora, Ubuntu собираются с этим флагом
- Использование в дистрибутивах:
 - Пакеты Fedora и Ubuntu дефолтно собираются с Stack Clash
 - Статус на Debian неясен ([compiler-flags-distro #12](#))
 - На Debian 12 (stable) не защищены даже уязвимые программы: bash, bzip2, curl, ffmpeg, perl, python, etc.
- Firefox использует защиту от Stack Clash ([BZ #1852202](#)), а Chrome нет

`_FORTIFY_SOURCE`: Как включить ?

- Для явного включения используются макросы –
`_FORTIFY_SOURCE=2` или `-D_FORTIFY_SOURCE=3`
 - Пока не появится `-D_FORTIFY_SOURCE=4` 😊
- Включена по умолчанию в компиляторе Ubuntu GCC (`-D_FORTIFY_SOURCE=3`)
 - Не включена в Debian и Fedora
 - Для Clang не включена по умолчанию нигде
- Использование в реальных проектах
 - В Debian пакеты дефолтно собираются с `-D_FORTIFY_SOURCE=2`
 - В Ubuntu с `-D_FORTIFY_SOURCE=3`
 - В Fedora: пакеты дефолтно собираются с `-D_FORTIFY_SOURCE=3` (с 2023)
 - Chrome и Firefox собираются с `-D_FORTIFY_SOURCE=2`

STL hardening: Как включить ?

- Libstdc++: `-D_GLIBCXX_ASSERTIONS`
 - (дефолтная STL в GCC и Clang)
- Libc++: `-D_LIBCPP_HARDENING_MODE=...`
 - (включается в Clang по флагу `-stdlib=libc++`)
- Visual Studio: `-D_ITERATOR_DEBUG_LEVEL=1`
- По умолчанию не включена в компиляторах в дистрибутивах Debian, Ubuntu и Fedora
- Использование в реальных проектах:
 - Включена по умолчанию для пакетов Fedora, но не для Debian и Ubuntu
 - Google: Chrome and server systems
 - [Retrofitting spatial safety to hundreds of millions of lines of C++](#)

Hardened allocators: Как включить ?

- Обычно просто `LD_PRELOAD=path/to/allocator.so`
- Проверки в Glibc включаются по `MALLOC_CHECK_=3` или `GLIBC_TUNABLES=glibc.malloc.check=3`
- Использование в реальных проектах:
 - Большинство дистрибутивов Linux используют Glibc
 - Android использует Scudo по умолчанию
 - Chrome использует hardened-аллокатор PartitionAlloc
 - [Efficient And Safe Allocations Everywhere!](#)
 - Firefox использует не-hardened аллокатор :(
 - [Firefox and Chromium: Memory Allocator Hardening](#)

Full RELRO: Как включить ?

- Опции линкера для включения Full RELRO: `-Wl, -z, now -Wl, -z, relro`
 - В Ubuntu включены по умолчанию в GCC, но не в Clang (в Clang только partial RELRO)
 - В Debian и Fedora не включены по умолчанию ни в GCC, ни в Clang
- Использование в реальных проектах
 - В Ubuntu и Fedora пакеты дефолтно собираются с Full RELRO
 - В пакетах Debian Full RELRO дефолтно не включён
 - Включён по дефолту в Chrome (BUILDD.gn) и Firefox (flags.configure)

Автоинициализация: Как включить ?

- Флаг `-ftrivial-auto-var-init=zero` в GCC и Clang
 - Не включён по умолчанию в компиляторе в Ubuntu, Debian, Fedora
- Скрытый флаг `-initiall` в Visual Studio
- Использование в реальных проектах:
 - Не включён по умолчанию в пакетах Ubuntu, Debian, Fedora
 - Дискуссия в треке Ubuntu ([#1972043](#))
- Включён в Chrome ([Chromium #40633061](#))
 - Исправление и отключение hot paths заняло ~4 месяца
- Пока не включён в Firefox ([Trivial Auto Var Init Experiments](#))
- Включён в Android user- и kernelspace ([System hardening in Android 11](#))

Целочисленные переполнения: Как включить ?

- **GCC:** `-fsanitize-trap=signed-integer-overflow,pointer-overflow`
 - GCC не поддерживает `integer`
 - Ещё раз отметим что `-ftrapv` *неработоспособна*
- **Clang:** `-fsanitize=signed-integer-overflow,pointer-overflow -fsanitize-minimal-runtime`
 - Рекомендую также добавлять `-fsanitize=integer` (может потребоваться добавить некоторые STL хедеры в `blacklist`)
- **Использование в реальных проектах:**
 - Защита не используется в Ubuntu, Debian, Fedora, а также в браузерах Chrome и Firefox
 - Включена в Android media stack:
 - [Android Developers Blog: Hardening media stack](#)
 - [Android Developers Blog: Compiler-based security mitigations in Android P](#)

CFI: Использование

- Включена по умолчанию на Android
- LLVM CFI не включена по умолчанию для пакетов в Ubuntu, Debian, Fedora
 - LTO + отсутствие поддержки в GCC
- Intel CET и AArch64 CFI по умолчанию включён для пакетов в [Ubuntu](#), [Fedora](#) и [Debian](#)
- Chrome использует LLVM CFI для X86 и AArch64 CFI для ARM
 - Firefox не использует никакой вариант CFI

CFI: Как включить ?

- LLVM CFI:
 - Не включена по умолчанию ни в GCC, ни в Clang в Ubuntu, Debian, Fedora
 - Включается по `-fsanitize=cfi` (также требует `-flto=thin` - `fvisibility=hidden`)
 - (LTO нужна для построения полного call graph программы, visibility для сокращения внешних вызовов)
 - Для межбиблиотечных вызовов нужна `-fsanitize-cfi-cross-dso` (замедляет выполнение)
- Intel CET:
 - Включается по `-fcf-protection`
 - Раньше ещё нужно было указывать флаги `-mcet`, `-mshstk` и `-mibt`, но теперь нет
 - Включена по умолчанию в GCC в Ubuntu ([Toolchain: Compiler flags](#))
- AArch64 CFI:
 - Включается по `-mbranch-protection=standard`
 - Никто не знает почему не использовали `-fcf-protection :`