Методы защиты от атаки типа переполнение буфера

Операционные системы

Клименко Алёна Сергеевна

Содержание

Список иллюстраций

Список таблиц

1 Методы защиты от атак типа переполнения буфера

Клименко Алёна Сергеевна

Группа НКАбд-02-2024

Реферат по курсу «Введение в операционные системы»

Российский университет дружбы народов

2025 год

1.1 Введение

Атака переполнения буфера — это один из самых известных и опасных видов атак на программное обеспечение. Она основана на возможности записи данных за пределы выделенной памяти буфера, что может привести к порче данных, сбоям, а в худшем случае — к выполнению произвольного вредоносного кода.

Впервые атака была подробно описана в 1996 году исследователем под псевдонимом Aleph One в статье “Smashing The Stack For Fun And Profit”. Несмотря на прошедшие десятилетия, уязвимость остаётся актуальной. В данном реферате рассматриваются основные методы защиты от атак переполнения буфера с примерами и объяснениями.

1.2 1. Понимание переполнения буфера

Буфер — это участок памяти, предназначенный для временного хранения данных. Если в него записать больше данных, чем он способен вместить, лишняя информация может перезаписать соседние ячейки памяти.

1.2.1 Пример уязвимого кода на C:

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

void vulnerable() {
    char buffer[8];
    gets(buffer); // НЕБЕЗОПАСНО!
    printf("Вы ввели: %s\n", buffer);
}

int main() {
    vulnerable();
    return 0;
}
```

В этом примере использование функции `gets()` допускает ввод данных, превышающих размер `buffer`, что может привести к перезаписи адреса возврата и выполнению произвольного кода.

1.3 2. Безопасное программирование

1.3.1 2.1 Использование безопасных функций

Опасные функции: `gets()`, `strcpy()`, `sprintf()`, `scanf("%s")` без ограничения длины.

Безопасные аналоги:

- `fgets()` — безопасная замена `gets()`
- `strncpy()` — ограничивает количество копируемых символов
- `snprintf()` — безопасная альтернатива `sprintf()`

1.3.2 Пример безопасного кода:

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

void safe() {
    char buffer[8];
    fgets(buffer, sizeof(buffer), stdin); // ограничение по размеру
    printf("Вы ввели: %s\n", buffer);
}

int main() {
    safe();
    return 0;
}
```

1.3.3 2.2 Проверка длины и границ

Перед копированием строк, чтением из ввода или записью данных важно проверять размер буфера, длину входных данных и граничные условия.

1.3.4 2.3 Статический анализ кода

Инструменты для анализа исходного кода (например, `cppcheck`, `Clang Static Analyzer`) позволяют автоматически находить потенциальные ошибки и уязвимости до выполнения программы.

1.4 3. Защита на уровне ОС и компиляции

1.4.1 3.1 Stack Canaries

Между локальными переменными и адресом возврата помещается специальный маркер (“канарейка”). При переполнении он затирается, и программа аварийно завершает выполнение.

Пример включения в GCC:

```
gcc -fstack-protector-all program.c -o program
```

1.4.2 3.2 ASLR (Address Space Layout Randomization)

ASLR случайным образом размещает стек, кучу, сегменты кода и библиотек в виртуальной памяти. Это затрудняет злоумышленнику точное определение адресов, необходимых для атаки.

Проверка в Linux:

```
cat /proc/sys/kernel/randomize_va_space
```

- 0 — отключено
- 1 — частичная рандомизация
- 2 — полная рандомизация (по умолчанию)

1.4.3 3.3 DEP / NX

DEP (Data Execution Prevention) или NX-бит (No-eXecute) запрещает выполнение кода в сегментах памяти, предназначенных только для хранения данных (например, в стеке или куче).

1.4.4 3.4 Флаги компиляции

Современные компиляторы позволяют встраивать защиту на этапе сборки:

- `-fstack-protector`, `-fstack-clash-protection`
- `-D_FORTIFY_SOURCE=2` — автоматическая защита стандартных функций
- `-pie -fPIE` — поддержка ASLR

1.5 4. Аппаратная защита

1.5.1 4.1 NX-бит в архитектуре x86

Процессоры Intel и AMD поддерживают NX-бит, который запрещает выполнение кода в страницах памяти, не помеченных как исполняемые.

1.5.2 4.2 ARM и MTE (Memory Tagging Extension)

Архитектура ARM реализует MTE — тегирование памяти, которое помогает отслеживать неправильное использование указателей и защищает от переполнений и use-after-free.

1.6 5. Языки программирования с встроенной безопасностью

- **Python** — проверка границ массивов, отсутствие прямого доступа к памяти
- **Java** — автоматическое управление памятью, исключения при выходе за границы массива
- **Rust** — строгая система владения памятью, невозможность переполнения без использования `unsafe`

1.7 Заключение

Переполнение буфера — серьёзная уязвимость, эксплуатируемая злоумышленниками для получения контроля над программой. Эффективная защита достигается только комплексным подходом:

- Безопасное программирование
- Использование защитных функций компилятора
- Аппаратная поддержка
- Защита на уровне операционной системы

Объединение этих методов значительно снижает риск эксплуатации уязвимостей и повышает общую безопасность программного обеспечения.

1.8 Список литературы

1. Aleph One. *Smashing The Stack For Fun And Profit*. Phrack Magazine, 1996.

2. Microsoft Security Documentation. *Data Execution Prevention (DEP)*.

3. PaX Team. *Address Space Layout Randomization*. PaX Documentation.

4. Туис раздел “Операционные системы”