

Методы защиты от атаки типа переполнение буфера

Операционные системы

Клименко Алёна Сергеевна

Российский университет дружбы народов, Москва, Россия

Информация

- Клименко Алёна Сергеевна
- НКАбд-02-2024 № Студенческого билета: 1132246741
- Российский университет дружбы народов
- https://github.com/Alstrr/study_2024-2025_os-intro

Введение

Атака переполнения буфера — это одна из наиболее распространённых уязвимостей в программном обеспечении, которая позволяет злоумышленнику записывать данные за пределы выделенной памяти, что может привести к исполнению вредоносного кода.

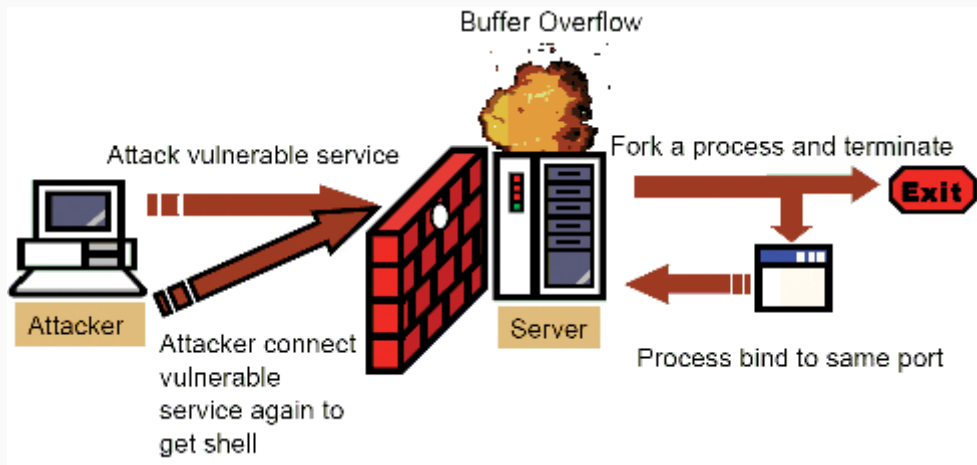


Рис. 1: Пример переполнения буфера

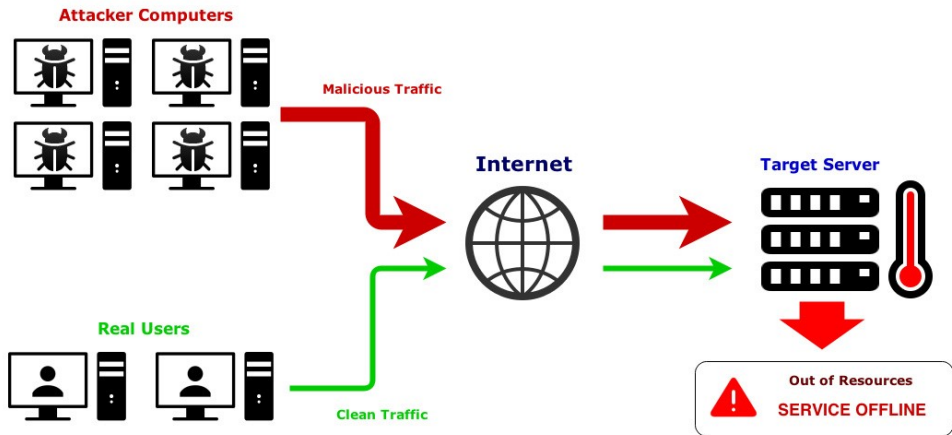
Основные методы защиты

Безопасное программирование включает:

- Использование безопасных функций
- Проверку границ буфера
- Анализ кода и статическую проверку

Например: заменить `gets()` на `fgets()`, использовать `strncpy()` вместо `strcpy()`, применять `snprintf()` вместо `sprintf()`.

Operation of a DDoS attack



Scudlayer

Рис. 2: Безопасные и небезопасные функции

- ASLR (Address Space Layout Randomization) — случайное размещение областей памяти процесса
- DEP (Data Execution Prevention) — запрет на выполнение кода в сегментах данных
- Stack Canaries — специальные метки, проверяющие целостность стека

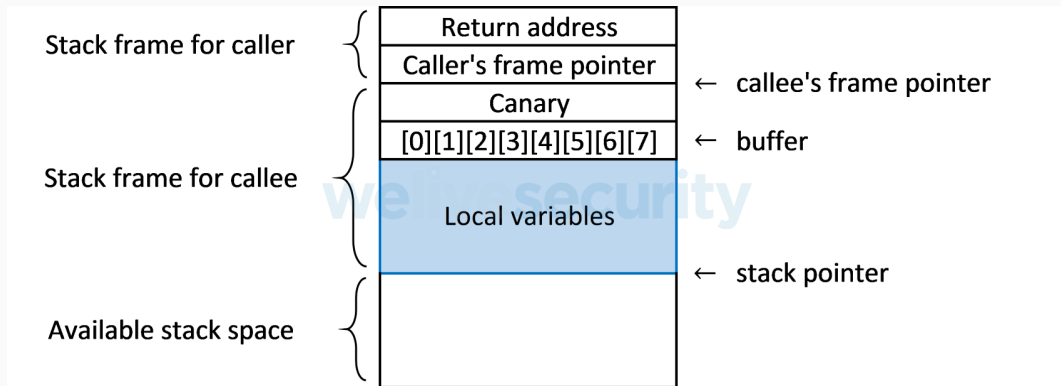


Рис. 3: ASLR, DEP и Stack Canaries

В стек между локальными переменными и адресом возврата добавляется специальное значение (канарейка).

Если оно изменено — программа аварийно завершается, предотвращая выполнение вредоносного кода.

Случайное расположение стека, кучи, сегментов кода и библиотек в памяти делает сложнее предугадать адреса, куда можно направить управление.

Запрещает выполнение кода в сегментах данных, например, в стеке и куче.
Даже если злоумышленник внедрил код — он не выполнится.

- Языки: Python, Java, Rust — встроенные механизмы защиты памяти
- Компиляторы: включают защиту на этапе сборки

Дополнительно:


- Флаги безопасности (`-fstack-protector`, `-D_FORTIFY_SOURCE`)
- Автоматическое внедрение защит (stack smashing protection, PIE)

Современные процессоры реализуют защиту на уровне архитектуры:

- NX бит в x86 — запрещает выполнение кода в данных
- ARM — контроль исполнения, защита доступа, тегирование памяти

Примеры атак и защиты

```
...  
  
#include <stdio.h>  
#include <string.h>  
void vulnerable_function(char *input) {  
    char buffer[8];  
    strcpy(buffer, input);  
    printf("Вы ввели: %s\n", buffer);  
}  
int main() {  
    char user_input[32];  
    printf("Введите данные: ");  
    gets(user_input);  
    vulnerable_function(user_input);  
    return 0;  
}  
...
```

```
  
#include <stdio.h>  
#include <string.h>  
void secure_function(char *input) {  
    char buffer[8];  
    strncpy(buffer, input, sizeof(buffer) - 1);  
    buffer[sizeof(buffer) - 1] = '\\0';  
    printf("Вы ввели: %s\\n", buffer);  
}  
int main() {  
    char user_input[32];  
    printf("Введите данные: ");  
    fgets(user_input, sizeof(user_input), stdin);  
    secure_function(user_input);  
    return 0;  
}
```

Заключение

Ни один метод не даёт 100% гарантии.

Только комплексный подход — безопасное программирование, флаги компиляции, аппаратная защита и рандомизация памяти — может эффективно предотвращать атаки на переполнение буфера.

1. Aleph One. *Smashing The Stack For Fun And Profit*. Phrack Magazine, 1996.
2. Microsoft Security. *Data Execution Prevention (DEP)*. Microsoft Docs.
3. PaX Team. *Address Space Layout Randomization*. PaX Documentation.