Лекция 21

ВВЕДЕНИЕ В БИНАРНЫЕ УЯЗВИМОСТИ
КРАТКИЙ ОБЗОР ОСНОВНЫХ ВИДОВ УЯЗВИМОСТЕЙ

rev. 5

Что такое бинарные уязвимости

Бинарные уязвимости

- Бинарные уязвимости – уязвимости, которые возникают непосредственно в исполняемых файлах
- Могут приводить к:
  - Падению приложения
  - Раскрытию какой-то секретной информации
  - Удаленному исполнению кода
  - Повышению привилегий
  - Обходу прочих механизмов безопасности
- Как можно видеть, по последствиям эти уязвимости очень похожи на те, что мы рассматривали в рамках раздела web
  - Однако, технически такие уязвимости обычно очень сильно отличаются

Бинарные уязвимости

- С такими уязвимостями можно столкнуться:
  - Открыв неудачный документ или медиафайл уязвимым просмотрщиком
  - Зайдя на сайт уязвимым браузером
  - Забыв обновить Windows или любую программу, доступную из локальной сети или интернета
  - И, конечно, написав уязвимую программу самостоятельно
- В CTF раздел с такими задачами называется Pwn или Exploit
  - Само слово pwn – искаженная форма own (предположительно из-за соседства на клавиатуре), часто применяется в leet speak

Немного терминологии

- Эксплойт – программа для эксплуатации уязвимости (отсюда и название), обычно является небольшим скриптом на языке Python. Термин применим как к программам для эксплуатации удаленных уязвимостей, так и локальных
  - Не очень умные антивирусы могут посчитать эксплойты вредоносным ПО, хотя оно обычно не наносит никакого вреда непосредственному пользователю
- 0day (или уязвимость нулевого дня) – уязвимость, для которой еще нет исправления, и о которой вообще обычно неизвестно автору ПО. Самый ценный (и опасный) вид уязвимостей, поскольку их использование возможно даже на последних версиях ПО

```
root@hacker1337:~/HackingApps$ ./hackthebank.py -h zelenybank.bank
Connecting to bank...
####################################
Fooling protection...
root@zelenybank:~$
```

Немного терминологии

- CVE (Common Vulnerabilities and Exposures) – база данных уязвимостей с присвоенными им номерами вида CVE-год-номер. CVE-идентификаторы – самый распространенный способ для ссылки на конкретную уязвимость
  - В отличие от модных названий, CVE-номера есть у большинства уязвимостей
- DoS (Denial of Service) – отказ в обслуживании, вид атаки, при котором нарушается функционирование некоторого ПО (например, оно падает или виснет)
- RCE (Remote Code Execution) – удаленное исполнение кода, конечная цель большинства атак
- PoC (Proof of Concept) эксплойт – эксплойт, который призван только продемонстрировать возможность атаки, обычно не позволяет выполнить полезную нагрузку (или же, например, работает только в одной версии программы). Можно считать альфа-версией эксплойта. Довольно часто исследователи ИБ заканчивают исследование уязвимости на этом этапе

Известные представители

- HeartBleed (CVE-2014-0160) – уязвимость раскрытия информации в OpenSSL, позволяла удаленно читать случайную память HTTP-сервера, в том числе получая оттуда закрытые ключи шифрования SSL
- EternalBlue (CVE-2017-0144) – уязвимость удаленного исполнения кода (с правами SYSTEM) в реализации SMB Windows, попала в новости в связи с трояном-шифровальщиком WannaCry. Не требует изменения настроек по умолчанию в Windows, в связи с чем была разрушительна для корпоративных сетей
- DirtyCOW (CVE-2016-5195) – уязвимость повышения привилегий в Linux, позволяла получить права пользователя root любому пользователю. Также использовалась для получения root на Android
  - Из более свежего того же плана – DirtyPipe (CVE-2022-0847)
- CVE-2024-1086 – свежая уязвимость повышения привилегий в Linux, можете попробовать ее у себя :)

Скрипт-кидди на заметку

- Если у вас нет желания разбираться в деталях работы конкретной уязвимости, но есть большое желание ей воспользоваться – можно попробовать поискать готовые эксплойты в интернете
  - Осторожно, в этом случае есть риск наткнуться на "rm -rf /" под видом эксплойта
- В этом случае вам могут пригодиться следующие инструменты:
  - Metasploit Framework – наиболее известный инструмент для проведения удаленных атак, мечта любого скрипт-кидди; наиболее известные уязвимости обычно обзаводятся реализацией эксплойта для Metasploit Framework (на Ruby). Самый легкий способ его установить – скачать Kali Linux.
  - Веб-сайты с эксплойтами, например https://exploit-db.com или https://0day.today/
- В рамках этого курса, однако, мы будем изучать более фундаментальные подходы, чем использование готовых эксплойтов

Как возникают уязвимости

Как возникают уязвимости

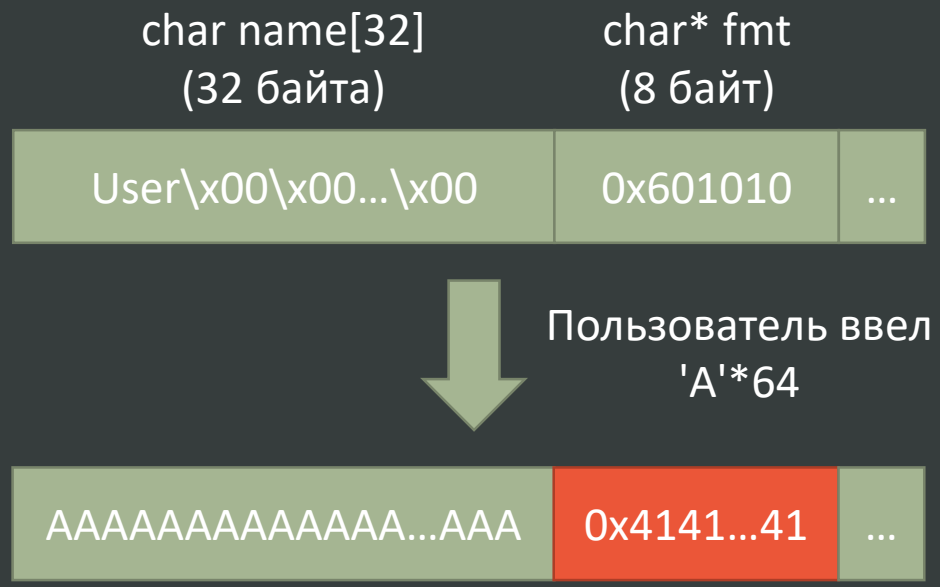
- Наиболее на слуху уязвимости, связанные с работой с памятью:
  - Переполнения буфера различного рода
  - Off-by-one
  - Buffer overread
  - Выход за границы массива
  - Висячие указатели
  - Уязвимость форматной строки
- Также встречаются и следующие уязвимости, которые напрямую с памятью не связаны:
  - Целочисленное переполнение
  - Состояние гонки
- Это далеко не полный список уязвимостей

Переполнение буфера

- В англоязычной литературе называется "buffer overflow"
- Эта уязвимость возникает в тот момент, когда ваша программа пытается записать в буфер больше данных, чем он может вместить
- В этом случае возникает повреждение данных, хранящихся в непосредственной близости от этого буфера
  - А там, в свою очередь, могут храниться различные строки, указатели и прочие важные для работы программы вещи
- Может возникнуть везде, где есть чтение или копирование данных в память без проверки их длины
  - Как на куче, так и на стеке и в .data / .bss
- Потенциально может привести к удаленному исполнению кода
  - Ну или хотя бы к падению программы

Переполнение буфера

```
static char name[32] = {'U','s','e','r'};
static char* fmt = "Hello %s\n";
int main()
{
    printf("What is your name? ");
    gets(name);
    printf(fmt, name);
    return 0;
}
```



- Поскольку gets() нигде не спрашивает размер буфера, пользователь может ввести сколько угодно букв (например, 64 буквы A)
- В итоге эти буквы вывалятся за границы буфера и перезапишут переменную fmt

```
What is your name? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

Переполнение буфера

- Следующие функции скорее всего доставят вам неприятности:
  - gets – из-за того, что эта функция в принципе не принимает длину строки в качестве параметра, она является небезопасной всегда, ее за это даже из стандарта C11 выгнали
  - scanf("%s", …) без указания размера – по сути тот же gets()
  - sprintf – опасна не всегда, однако в некоторых случаях (если вы неудачно подобрали длину строки под формат или добавили в формат %s) также может переполнить буфер
  - strcat, strcpy – функции для сложения и копирования строк, не принимающие длину целевой строки в качестве параметра
- Их, соответственно, предлагается заменить на:
  - fgets – функция, которая читает из указанного файла указанное число байтов (осторожно, вместо gets она сохраняет символ переноса строки)
  - snprintf, strncat / strlcat, strncpy / strlcpy – функции, также принимающие число байтов / букв в качестве дополнительного аргумента

Время задач

Эхо кибервойны

Категория: Lesson 21 / Pwn basics

Решивших: 0

Время: 00:00:02

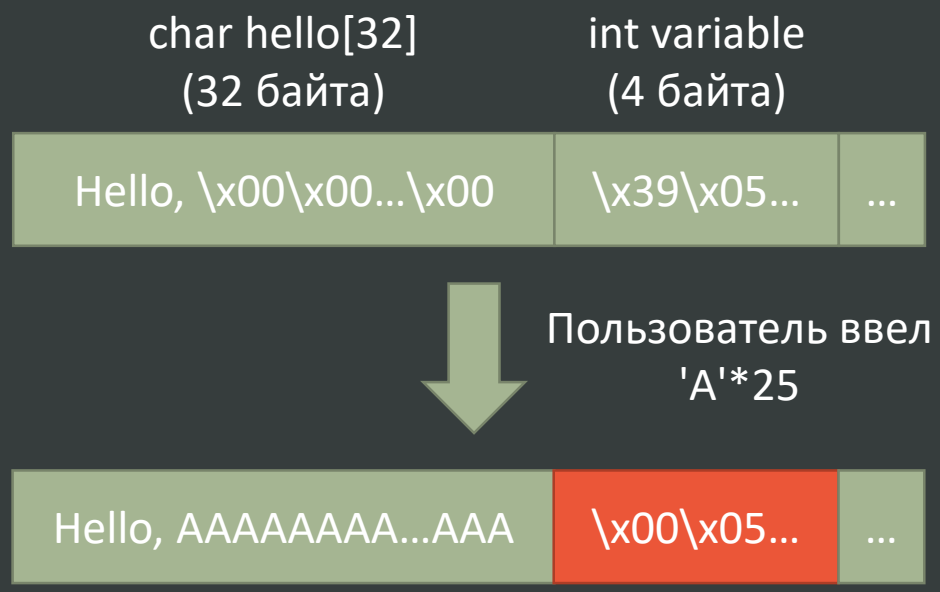
- Доступ к задачам можно получить как всегда на nsuctf.ru

Off-by-one

- Также называется «ошибка на единицу»
- Целый класс ошибок, которые можно встретить, если случайно не обработать первый / последний элемент, в каком-либо массиве
  - Например, начав обработку с первого (1) элемента массива, а не с нулевого
  - Также такую ошибку можно получить, поставив условием конца цикла <= вместо < (или наоборот)
- В мире ИБ очень часто связана с нуль-терминированными строками
  - У таких строк длины N букв длина в памяти составляет N+1 (потому что на конце есть еще нулевой символ)
  - На такие грабли можно наступить при проверке длины (в байтах) строки через strlen
- Также «безопасная» функция strncat в этом смысле небезопасна – она в качестве аргумента принимает число букв, а не байтов (в отличие от snprintf, например)

Off-by-one

```
static char name[32];
static char hello[32] = "Hello, ";
int variable = 1337;
int main() {
    printf("What is your name? ");
    fgets(name, 32, stdin);
    strncat(hello, name, 32-strlen(hello));
    puts(hello);
    printf("Variable == 0x%x\n", variable);
    return 0;
}
```



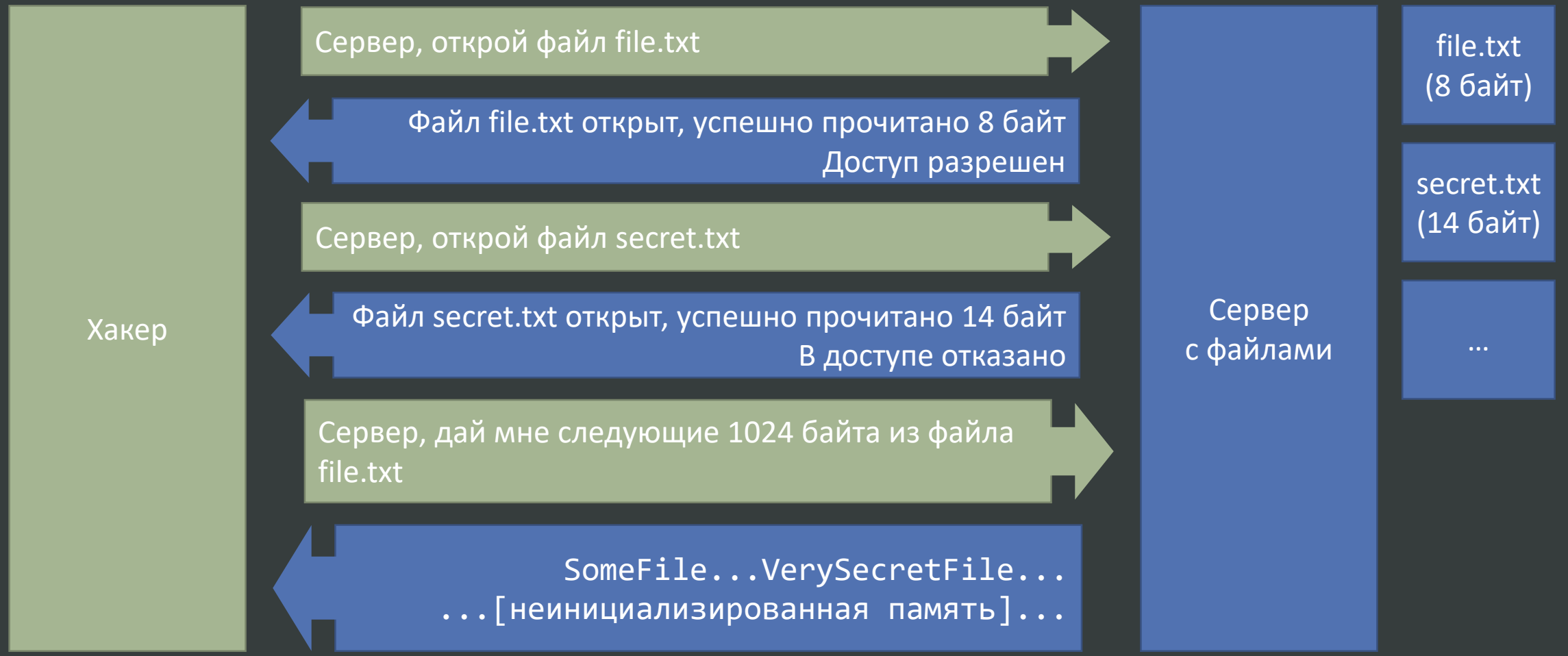
- Поскольку strncat спрашивал размер в буквах, а не в байтах, при вводе от 25 букв 'A' нульбайт вывалится за пределы переменной hello и обнулит один байт переменной variable

```
What is your name? AAAAAAAAAAAAAAAAAAAAAAAAA
Hello, AAAAAAAAAAAAAAAAAAAAAAAAA
Variable == 0x500
```

Buffer overread

- В то время как buffer overflow записывает в буфер лишнего, buffer overread – наоборот читает из буфера лишнего
- Эта уязвимость возникает в тот момент, когда пользователь может неконтролируемо влиять на длину данных, читаемых из буфера
  - Как правило, это касается сетевых протоколов, где отдельным полем указывается длина отправляемого блока информации
- Обычно приводит только к чтению информации из памяти сервера
- Представителем этого типа уязвимостей является HeartBleed

Buffer overread



```c
#include <stdio.h>
#include <unistd.h>
static char name[1024] = {'S','e','c','r','3','t','n','a','m','3'};
int main()
{
    int len;
    setvbuf(stdout, NULL, _IONBF, 0);
    printf("How long is your name (in bytes)? ");
    scanf("%d", &len);
    if (len > 1024)
        return 1;
    printf("What is your name? ");
    read(STDIN_FILENO, name, len);
    printf("Hello ");
    write(STDOUT_FILENO, name, len);
    printf("\n");
    return 0;
}
```

```
How long is your name (in bytes)? 128
What is your name? a
Hello a
cr3tnam3
```

Buffer overread

- Здесь потенциально могут доставить неприятности функции, в которые нужно указать максимальный размер буфера, а не точный
  - К таким функциям относятся, например, read / recv и write / send – они возвращают число успешно прочитанных / записанных байт и оно может совершенно не совпадать со значением, которое вы передали
- Кстати, у scanf также есть возвращаемое значение и если вы его проигнорируете, переменные просто не будут перезаписаны, если пользователь приведет их в некорректном формате (правда это уже не совсем buffer overread)

```c
int main() {
    int guess, backup;
    guess = rand();
    backup = guess;
    printf("Guess the number: ");
    scanf("%d", &guess);
    printf((backup == guess) ? "You won!\n" : "You lose!\n");
}
```

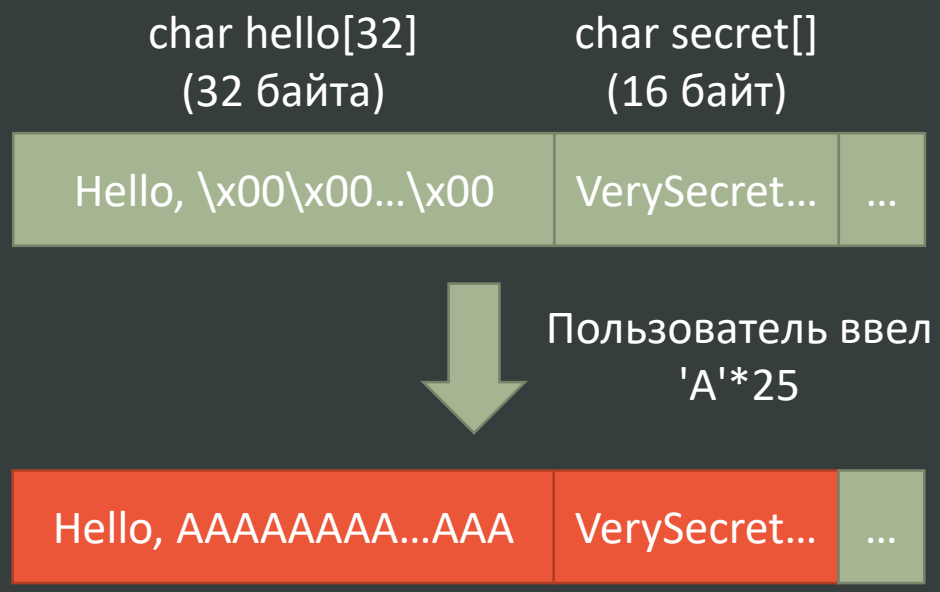
```
Guess the number: 123
You lose!


Guess the number: qwe
You won!
```

Buffer overread для строк

- Похожая проблема может возникнуть и в случае, если сервер ожидал получить строку с нуль-байтом на конце, но клиент его подвел
  - В этом случае получившаяся строка будет содержать и данные, которые лежат непосредственно за строкой до первого нуль-байта
- Некоторые функции, в свою очередь, просто не заморачиваются установкой null-байтов в конце строки
  - Например, к таким относится функция readlink(), читающая значение символьной ссылки
  - Также, как ни странно, к таким функциям относится и strncpy

Buffer overread для строк

```
static char name[32];
static char hello[32] = "Hello, ";
static char secret[] = "VerySecretString";
int main() {
    printf("What is your name? ");
    fgets(name, 32, stdin);
    strncpy(hello + strlen(hello), name, 32 -
strlen(hello));
    puts(hello);
    return 0;
}
```



- Функция strncpy обычно дополняет весь остаток буфера len нулями
  - Но поскольку длина name оказалась предельно допустимой, нульбайтов оказалось ровно ноль и строки «склеились», в результате чего мы смогли прочесть содержимое secret

```
What is your name? AAAAAAAAAAAAAAAAAAAAAAAA
Hello, AAAAAAAAAAAAAAAAAAAAAAAAVerySecretString
```

Выход за границы массива

- В англоязычной литературе называется "out-of-bounds access"
- Эта уязвимость возникает, если пользователь может контролировать индексы какого-то массива, а вы их не проверяете на принадлежность к допустимому диапазону
  - Пользователь может использовать и отрицательные и положительные индексы, чтобы ходить в обе стороны от массива
  - Также уязвимость такого рода может возникнуть и при ручной работе с указателями
- Очень похожа на предыдущие две, но зачастую позволяет записывать и читать только нужные байты, а не все подряд
- В зависимости от характера доступа может привести как к чтению данных, так и к удаленному исполнению кода
  - И практически наверняка приводит к возможности уронить программу

Выход за границы массива

```c
int fib[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
int main()
{
    int d;
    printf("Enter index of fibonacci number: ");
    scanf("%d", &d);
    printf("Fibonacci number %d is %d\n", d, fib[d]);
    return 0;
}
```

```
Enter index of fibonacci number: 1337
Segmentation fault (core dumped)
```

Время задач

Заметки

Категория: Lesson 21 / Pwn basics

Решивших: 0

Время: 00:00:01

- Доступ к задачам можно получить как всегда на nsuctf.ru

Висячие указатели

- Висячий указатель (dangling pointer) – указатель, который уже не указывает на корректный объект, но все еще может быть использован где-то в коде
- Может возникнуть в следующих случаях:
  - Если указатель раньше указывал на какую-то память, но ее освободили вызовом функции free (также такой сценарий зачастую называют Use-After-Free), основной сценарий
  - Если программист неправильно использовал функцию realloc (после ее выполнения указатель вообще-то может переехать в другое место)
  - Если программист зачем-то решил вернуть указатель на переменную, выделенную на стеке (впрочем, в этом случае программа скорее всего упадет и программист заметит неладное)
- Потенциально может привести к удаленному исполнению кода и чтению информации из памяти сервера

Висячие указатели / Use-after-free

```
int main()
{
    char* a = malloc(128);
    free(a);
    char* b = malloc(128);
    strcpy(b, "SomeString");
    strcpy(a, "This shouldn't be here");
    printf("%s\n", b);
    return 0;
}
```

- При последовательном выполнении malloc-free-malloc с одинаковым размером выданные адреса наверняка совпадут (т.е. a == b)
- В этом случае запись по более недействительному указателю a запишет в b

```
This shouldn't be here
```

Уязвимость форматной строки

- Отдельный тип уязвимости, специфичный для языка Си и его функций вроде printf
- Возникает когда пользователь может влиять на содержимое форматной строки
- Может привести как к удаленному исполнению кода, так и чтению информации из памяти сервера

```c
#include <stdio.h>
static char string[32];
int main()
{
    printf("Enter some string: ");
    fgets(string, 32, stdin);
    printf("You entered: ");
    printf(string);
    return 0;
}
```

```
n0n3m4@localhost:~$ ./a.out
Enter some string: qweqwe
You entered: qweqwe
n0n3m4@localhost:~$ ./a.out
Enter some string: %s
Segmentation fault (core dumped)
```

Целочисленное переполнение

- Целочисленное переполнение – ситуация, при которой результат математической операции не влазит в используемый тип данных
  - Обычно в этом случае операция производится по модулю 2^N для беззнаковых чисел, где N – размер типа в битах
- Само по себе переполнение не является уязвимостью (и может даже использоваться как функция языка), однако есть следующие проблемы:
  - Для знаковых чисел вы можете столкнуться с термином «неопределенное поведение» – компилятор Си может соптимизировать вашу программу, использующую переполнение, совершенно непредсказуемым образом
  - Эта возможность может использоваться для обхода различных проверок (в том числе проверок границ массивов), если они реализованы неудачно (в частности присутствует только проверка на «меньше»)

Целочисленное переполнение

```c
int numbers[16];
int main()
{
    long n;
    printf("How many numbers would you like to store? ");
    scanf("%ld", &n);
    if (n * sizeof(int) > sizeof(numbers))
        return 1;
    for (int i=0;i<n;i++)
        scanf("%d",&numbers[i]);
    return 0;
}
```

```
How many numbers would you like to store? 4611686018427387904
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ... 1 1 1
Segmentation fault (core dumped)
```

```
4611686018427387904 * 4
== 2**64 == 0 (mod 2**64)
```

Состояние гонки

- Состояние гонки (race condition) – состояние, при котором итоговое выполнение программы зависит от того, в каком порядке выполнится код из разных потоков
  - Например, два потока пытаются записать в одну и ту же память какие-нибудь временные данные, самый легкий способ этого добиться – возвращать данные из функции в статическом буфере (или просто хранить состояние где-нибудь в статической переменной, как в функции strtok)
- В случае с информационной безопасностью речь скорее идет об ошибке TOCTOU – time of check to time of use
  - Это когда какие-то данные сначала проверяются на корректность, а потом используются, причем за это время они вполне могли измениться
- В случае с setuid-программами или ядром ошибка подобного рода может привести к повышению привилегий
  - В частности, DirtyCOW использует как раз race condition в ядре

Состояние гонки

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE * f = fopen("./script.sh", "r");
    fseek(f, 0L, SEEK_END);
    int sz = ftell(f);
    fclose(f);
    if (sz > 10)
    {
        printf("File too large\n");
        return 1;
    }
    system("./script.sh");
    return 0;
}
```

```python
import os
while True:
    os.rename('script.sh', '_script.sh')
    os.rename('xscript.sh', 'script.sh')
    os.rename('script.sh', 'xscript.sh')
    os.rename('_script.sh', 'script.sh')
```

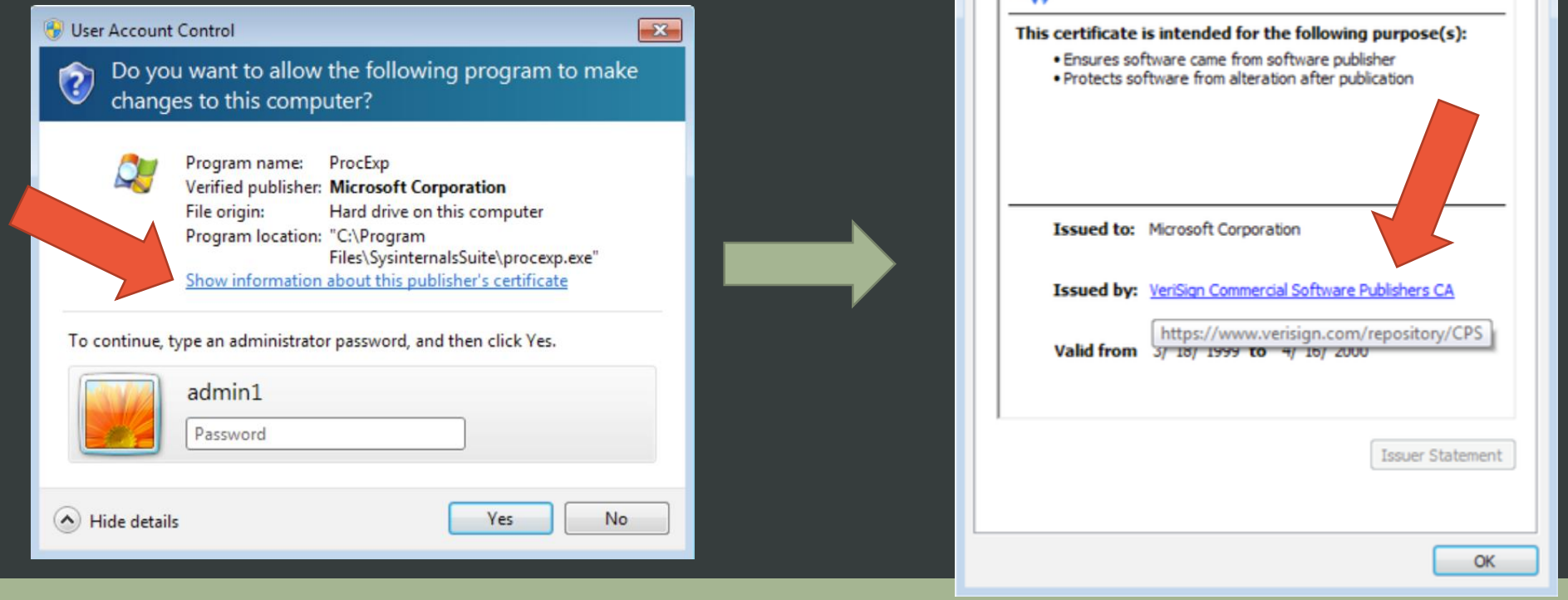
```
n0n3m4@localhost:~$ ./a.out
Segmentation fault (core dumped)
n0n3m4@localhost:~$ ./a.out
sh: 1: ./script.sh: not found
n0n3m4@localhost:~$ ./a.out
youarehacked
n0n3m4@localhost:~$ ./a.out
File too large
```

Как защититься от уязвимостей?

- Писать код внимательно и аккуратно
  - Не слишком популярный вариант, как можно догадаться
- Использовать статические анализаторы кода
- Тестировать свои программы при помощи автоматических инструментов (например, AFL fuzzer)
  - Этот вариант не слишком надежен – что-то вполне может и не найтись
- Не использовать Си
  - Такие языки как Java и Python гарантированно защищают от бед с памятью, для высокопроизводительных программ некоторые предпочитают Rust и Go
- Использовать костыли для Си
  - ASLR, stack canary, DEP, Intel CET и прочие технологии, обеспечивающие «безопасность», с которыми мы встретимся в следующих лекциях

Логические уязвимости

- От логических уязвимостей, к сожалению, автоматические инструменты вас не спасут
  - К таким отчасти относится и race condition
  - Очень хороший пример такой уязвимости – CVE-2019-1388



С чего начинается взлом

С чего начинается взлом

- Обычно взлом начинается с поиска падений в программе
  - Каждое падение – потенциальная возможность для несанкционированного чтения или записи памяти, поскольку падение по определению является незапланированным поведением программы
- В случае CTF падения во многих случаях ищутся руками
  - Путем анализа кода программы на небезопасные конструкции
  - Ручным запихиванием AAAAAAAA…AAAAA, %s и всего такого во все поля программы
- В реальности падения обычно ищут фаззерами, которые автоматически занимаются примерно тем же самым – запихивают нестандартные значения во все поля
  - Впрочем, фаззер едва ли может попасть во все ветви программы самостоятельно, так что без анализа кода тоже не обходится

Извлекаем пользу из падений

- Чтобы понять, где именно произошло падение и к какой памяти был получен несанкционированный доступ, лучше запускать программу под отладчиком
  - Как всегда для этого хорошо подойдет GDB
- В момент падения (т.е. когда программа получит сигнал от системы), GDB передаст вам управление и расскажет доступную ему информацию о падении

```
(gdb) r
Starting program: ./overflow
What is your name? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
__strchrnul_avx2 () at ../sysdeps/x86_64/multiarch/strchr-avx2.S:57
57      ../sysdeps/x86_64/multiarch/strchr-avx2.S: No such file or directory.
(gdb)
```

Различаем сигналы

- SIGSEGV (Segmentation fault) – наиболее популярный сигнал, который возникает при ошибках памяти, возникает в следующих основных случаях:
  - Вы попытались обратиться к несуществующей памяти
  - Вы попытались записать в память, доступную только для чтения
- SIGBUS (Bus error) – сигнал, который встречается значительно реже, но по смыслу напоминает SIGSEGV: возникает, например, при нарушении выравнивания данных
- SIGILL (Illegal instruction) – сигнал, который возникает, если процессор столкнулся с некорректной инструкцией, обычно свидетельствует о следующем:
  - Вам удалось как-то повлиять на счетчик команд и теперь процессор попал куда-то посередине инструкции (но все еще находится в корректном участке памяти, иначе вы бы получили SIGSEGV)
  - Ваши данные почему-то исполнились как код (а в них вряд ли лежали корректные x86-инструкции)

Извлекаем пользу из падений

- Продолжим исследование падения, которое мы получили в прошлом примере
- Хорошей идеей будет осмотреться при помощи команды "disas $pc,+0x32", чтобы понять, на какой инструкции случилась беда

```
(gdb) disas $pc,+0x32
Dump of assembler code from 0x7fffff18e1e8 to 0x7fffff18e21a:
=> 0x00007fffff18e1e8 <__strchrnul_avx2+24>:    vmovdqu ymm8,YMMWORD PTR [rdi]
```

- Можно предположить, что случилась какая-то проблема с разыменованием RDI, посмотрим, что в нем:

```
(gdb) p/x $rdi
$1 = 0x4141414141414141
```

- Судя по всему, это наши буквы AAAAAAAA, которые как-то стали адресом
  - В hex-представлении буква A это 0x41

Извлекаем пользу из падений

- Посмотрев backtrace, поймем, что проблема случилась в функции printf:

```
(gdb) backtrace
#0  __strchrnul_avx2 () at ../sysdeps/x86_64/multiarch/strchr-avx2.S:57
#1  0x00007ffff05b432 in __find_specmb (
    format=0x4141414141414141 <error: Cannot access memory at address
0x4141414141414141>) at printf-parse.h:108
#2  _IO_vfprintf_internal (s=0x7ffff3ec760 <_IO_2_1_stdout_>,
    format=0x4141414141414141 <error: Cannot access memory at address
0x4141414141414141>, ap=ap@entry=0x7fffffede80)
    at vfprintf.c:1320
#3  0x00007ffff064f26 in __printf (format=<optimized out>) at printf.c:33
#4  0x00000000080006eb in main ()
```

- Уже отсюда видно, что мы смогли повлиять на адрес форматной строки
  - Заменив его на какой-нибудь реальный адрес, мы могли бы читать произвольную память

Немного организационных вопросов

- В большинстве задач курса будет предполагаться получение удаленного исполнения кода
- Чтобы вам не приходилось возиться с различными вопросами уровня «как раздобыть IP-адрес для backconnect» или заниматься прочей рутиной вроде dup2() stdin и stdout на сокет, все программы будут использовать стандартный ввод и вывод
  - Таким образом, запуск программы на вашем компьютере через ./binary будет по поведению эквивалентен nc <удаленный_сервер> <удаленный_порт>
- Задачи будут поставляться вместе с исходным кодом, чтобы вам не приходилось их реверсить
  - Однако, вам все равно скорее всего понадобится дизассемблер, чтобы посмотреть какие-то адреса в памяти

Спасибо за внимание!
Задачи доступны на

nsuctf.ru

- Пожалуйста, используйте имя пользователя формата "Фамилия Имя"
  - e-mail можно забить любой, сервером он не проверяется
- Для вопросов по задачам рекомендую присоединиться к @NSUCTF в Telegram
  - Только, пожалуйста, без спойлеров