

МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

Направление 02.03.03

Математическое обеспечение и администрирование информационных систем

Отчёт по теме:

Тестирование

Группа: 5130203/10101

Обучающийся: _____

Нгуен Тхе Хунг

Бу Хоай Нам

Евгений Александрович Жабко

Преподаватель: _____

Пархоменко Владимир Андреевич


« _____ » _____ 20 ____ г.

Санкт-Петербург 2024г

Содержание

1	What is symbolic Execution	3
1.1	Путь выполнения	3
1.2	Символы	3
1.3	Что решает символическое выполнение ?	4
1.4	Как можно применять символическое выполнение	4
2	Angr	6
2.1	Путь выполнения	6
2.2	State Explosion	6
2.3	Внедрение символа	7
3	Бинарные файлы	8
3.1	Семейство C/C++	9
3.2	GCC complier	9
3.3	Visual Studio	10
3.4	Для других языков	11
3.4.1	C#	11
3.4.2	Python	12
4	Пример работы ANGR	14
4.1	Пример 1	14
4.2	Пример 2	16
4.3	Пример 3	20

1 What is symbolic Execution

 **Symbolic execution = symbolic + execution**

Символьные выполнение простыми словами - это нахождение пути **выполнения** с помощью **символов**.

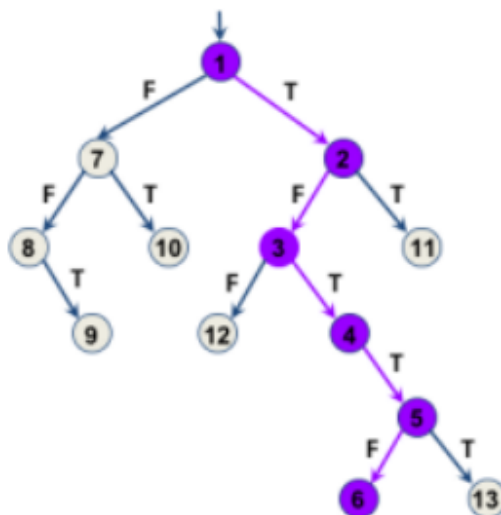
1.1 Путь выполнения

Программа - это набор инструкций, которые описывают пути выполнения код для решения какой-то определенной задачи. Если разбить программу на шаги, мы получим какой-то путь выполнения. Программа может содержать конструкции ветвления (IF ELSE), поэтому мы не сможем, выполнив программу, получить единую линию выполнения. Наши пути выполнения образуют некое древо.

Поэтому программу можно рассматривать как бинарное дерево, которое имеет бесконечную длину.

В этом древе:

- Каждый узел представляет выполнение условного оператора
- Каждое ребро представляет выполнение последовательности необусловленных операторов
- Каждый путь в древе представляет эквивалентный класс входных данных



1.2 Символы

Когда речь идёт о символьном выполнении, символы в этом контексте означают, что нужно рассматривать входные данные как абстрактные. то есть входные данные не имеют никакого отношения к конкретным значениям.

Например, x - символ в уравнении, при каком-то конкретном значении x получаем какое-то значение выражения.

$$x^2 + 2x + 3$$

Можно сказать, что x зависит от уравнения, ограничивающего его.

Символ же в свою очередь зависит от пути выполнения, ограничивающего его.

1.3 Что решает символьное выполнение ?

До появления символьного выполнения было популярно случайное тестирование, то есть в качестве входных данных выбирались какие-то случайные значения, затем программа запускалась с этими конкретными значениями, и изучался результат и поведение программы. Это подход может быть быстрее и проще, если вариаций значений входных переменных мало, например `boolean`. А как решать задачи, если домен переменной может достичь значения в 1 миллион ? Миллиард ?!.

```
void test_me(int x) {  
    if (x == 94389) {  
        ERROR;  
    }  
}
```

Probability of **ERROR**:
 $1/2^{32} \approx 0.000000023\%$

В таких случаях подход символьного выполнения приходит к нам на помощь. Поскольку как мы уже упоминали ранее, этот подход смотрит на данные не как на какие-то конкретные значение, а как символы, и анализирует путь выполнения на базе математической теории.

Символьное выполнение имеет следующие преимущества:

- Полное покрытие путей выполнения
- Обнаружение ошибок, которые могут быть пропущены при рандомном тестировании
- Анализ безопасности
- Улучшение качества кода

1.4 Как можно применять символьное выполнение

В настоящее время имеется определенный ряд библиотек, которые поддерживают символьное выполнение, например:

- angr
- miasm
- s2e

- manticore
- klee

В нашей работе мы выберем angr, так как эта библиотека имеет следующие преимущества

- Написана на Python, поэтому легче использовать, чем другие, которые написаны на C++
- Хорошо документирована
- Имеет версии GUI, которые легко использовать, не надо создавать никакие значения
- Доступен ряд дополнительных функций : - Control-flow analysis, Disassembly, - Decompilation, ...

2 Angr

2.1 Путь выполнения

Путь выполнения представляет собой возможное выполнение программы, которое начинается и заканчивается в определённых точках.

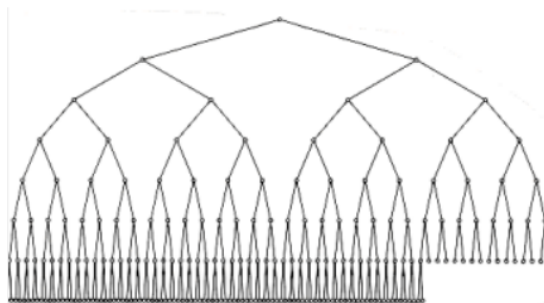
Мы можем задать начальное или финальное место

```
# start at entry point
project.factory.entry_state()
# giving start address
startAddress = 0x123456
initState = project.factory.blank_state(addr=startAddress)
# giving end address
endAddress = 0x654321
simulation.explore(find=endAddress)
```

2.2 State Explosion

Если же мы ничего не задаём, тогда программа найдёт все возможные пути сама, и конечно это будет не эффективно для больших программ, в которых мы точно знаем, какое условие работает корректно и его можно пропустить.

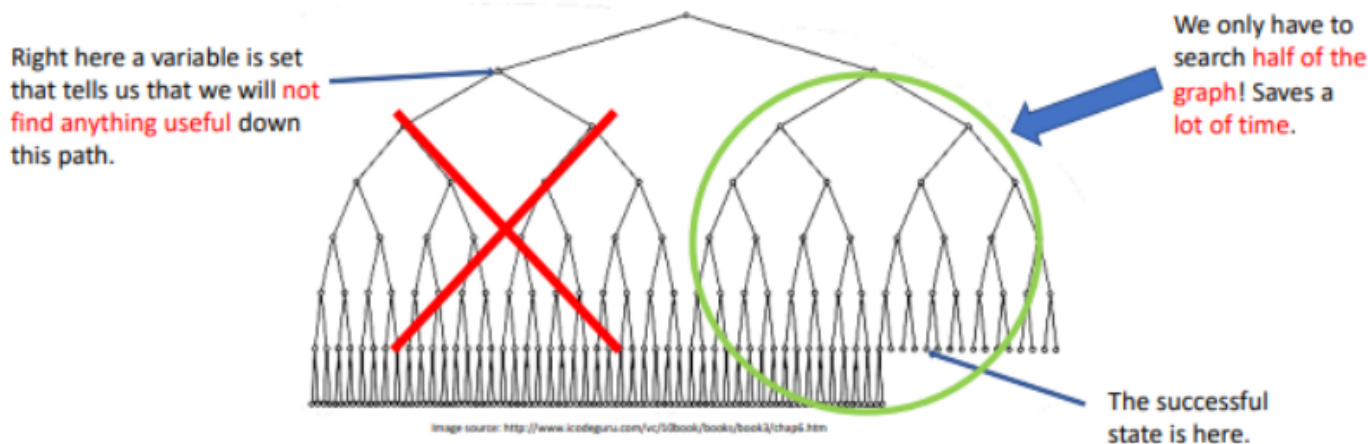
Полный поиск



Angr предоставляет нам возможность выбрать пути выполнения, которые нас интересуют. Это реализует алгоритм Find and Avoid. Суть алгоритма: создаётся две предикаты и они проверяются в каждом состоянии программы.

- `Is_found_path`: проверяет, что в данном состоянии финальная искомая точка. Если это верно, тогда обход завершается.
- `Is_avoid_path`: проверяет, что для данного состояния можно пропустить проверку. (то есть нас не интересует данный путь)

```
simulation.explore(find=is_found_path, avoid=is_avoid_path)
# we also can giving address as find and avoid
simulation.explore(find=0x123456, avoid=0x123458)
```



Это ускоряет процесс обхода графа, также код становится понятнее.

2.3 Внедрение символа

Самая интересная возможность символьного выполнения - это внедрение символа в программу. Программа рассматривается на уровень абстракции выше.

В Angr мы можем внедрять символы с помощью битовых векторов.

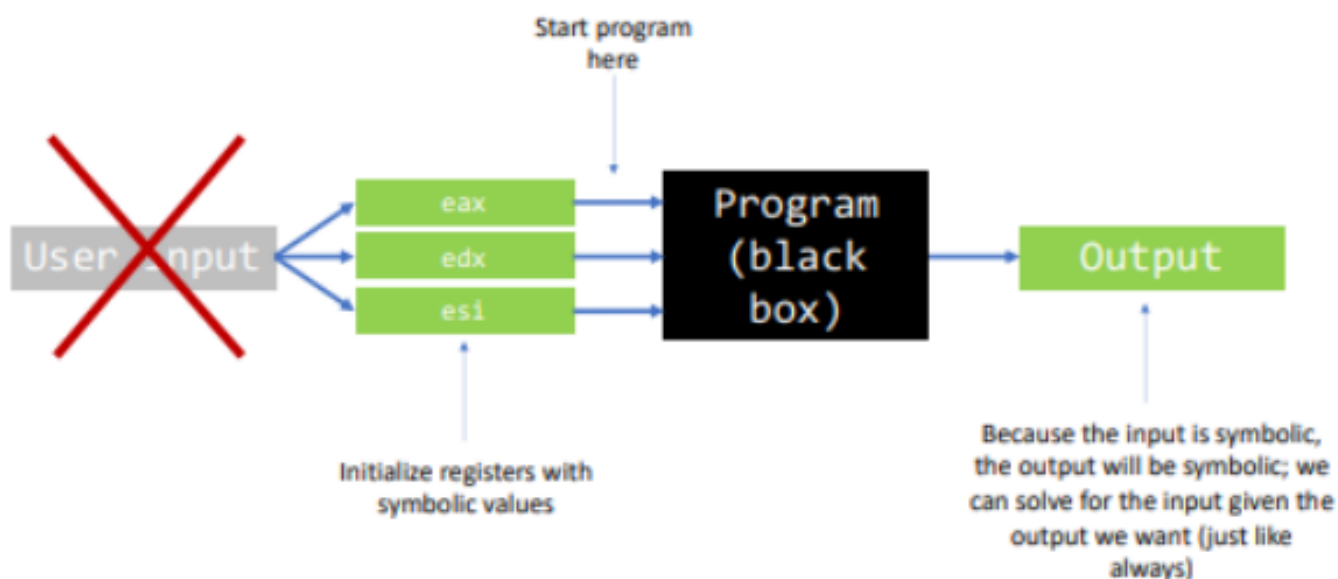
bitvectors

Битовый вектор, также известный как битовый массив или битовая строка, представляет собой структуру данных, используемую в информатике для хранения последовательности битов. Каждый бит в векторе может быть равен 0 или 1, представляя два возможных состояния

Битовый вектор может представлять собой любой тип, который может в него поместиться. Таким образом, битовый вектор может хранить любую переменную, если мы задали для нее правильный размер.

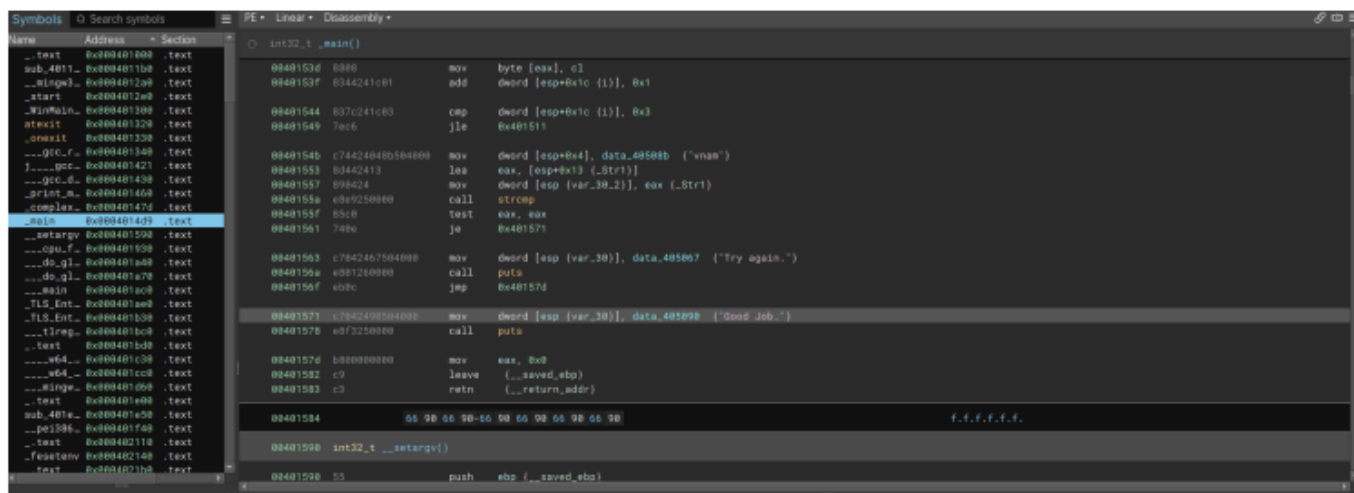
Суть внедрения символов:

- создаётся битовый вектор, который будет представлять значение регистра (должен иметь тот же размер, что и этот регистр)
- этот битовый вектор будет обрабатываться как символ в программе, значение состояния будет храниться в этом битовом векторе
- когда мы достигнем нужного нам состояния, мы вычислим битовый вектор, чтобы получить конкретное значение



3 Бинарные файлы

Поскольку библиотека `angr` принимает бинарный файл в качестве входных данных для тестирования, появляется возможность тестировать практически любые программы, написанные на разных языках программирования. Для любых языков программирования мы можем получить из кода бинарных файл, после выполнения ряда манипуляций.



Бинарный файл является результатом компиляции и содержит машинный код. Бинарные файлы могут отличаться друг от друга в зависимости от языка программирования и компилятора. Например, в ОС Windows бинарный файл имеет расширение .exe, а в UNIX можно встретить файл без расширения.

Далее рассмотрим, как можно получить бинарный файл из исходного кода.

3.1 Семейство C/C++

Для семейства языков C и C++ можно использовать GCC compiler, чтобы получить бинарный файл, либо можно использовать Visual Studio.

Рассмотрим код, бинарный файл которого мы тестировали ранее.

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

#define USERDEF "VNAMAAA"
#define LEN_USERDEF 7

int complex_function(int value, int i)
{
    #define LAMBDA 3
    if (!('A' <= value && value <= 'Z'))
    {
        printf("Try again.\n");
        exit(1);
    }
    return ((value - 'A' + (LAMBDA * i)) % ('Z' - 'A' + 1)) + 'A';
}

int main(int argc, char *argv[])
{
    char buffer[9];

    printf("Enter the password: ");
    scanf("%8s", buffer);
    for (int i = 0; i < LEN_USERDEF; ++i)
    {
        buffer[i] = complex_function(buffer[i], i);
    }
    if (strcmp(buffer, USERDEF))
    {
        printf("Try again.\n");
    }
    else
    {
        printf("Good Job.\n");
    }
}
```

3.2 GCC compiler

В первом подходе преобразования исходного кода в бинарный вид используем GCC compiler на cmd, чтобы получить бинарный файл.

```
PS D:\angr_ctf-master\play_ground> gcc .\app.c -o app.exe
PS D:\angr_ctf-master\play_ground> .\app.exe
Enter the password:
```

Получили исполняемый файл:

app.exe	19/03/2024 12:13 SA	Application	42 KB
---------	---------------------	-------------	-------

Для этого используем MinGW

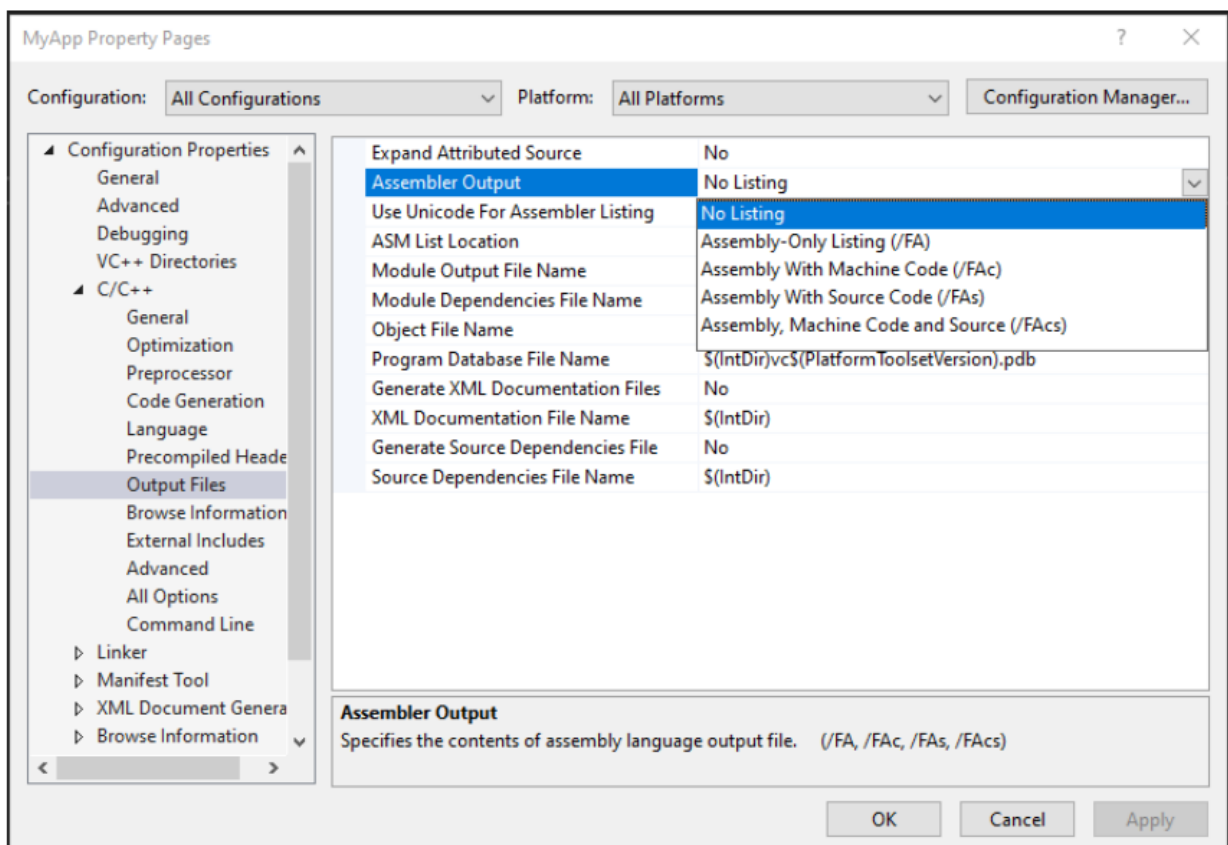
```
C:\Users\Admin>gcc --version
gcc (MinGW.org GCC-6.3.0-1) 6.3.0
Copyright (C) 2016 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

3.3 Visual Studio



Visual Studio - это очень функциональная IDE для программирования и поддерживает массу различных языков, в том числе C/C++. В ней можно задавать различные параметры, чтобы получать бинарные файлы и не только.

По умолчанию после запуска Visual Studio автоматически файл будет исполняемым, если мы хотим получить файл assembly, можем сделать следующие настройки:

Это открывается по пути project/ properties/C/C++/ Output files



Файл, полученный в результате:

 MyApp.exe	19/03/2024 12:26 SA	Application	85 KB
 MyApp.pdb	19/03/2024 12:26 SA	Program Debug D...	1.532 KB

Стоит заметить, что бинарный файл, генерируемый с помощью Visual Studio сложнее и больше, чем бинарный файл, который мы создали ранее вручную с помощью gcc compiler. Это происходит, потому что Visual Studio внедряет свои инструкции для защиты бинарного файла и некоторую информацию для компилирования.

Инструкции от Visual Studio.

```
00A64455 B9 66 20 A7 00      mov     ecx,offset _2F4A54EF_MyApp@cpp (0A72066h)
00A6445A E8 FC CF FF FF      call    @__CheckForDebuggerJustMyCode@4 (0A6145Bh)
```

3.4 Для других языков

3.4.1 C#

Для C# алгоритм похожий на C или C++. С помощью Visual Studio мы можем также получить файл бинарный.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp1.FileLearn
{
    internal class StreamMain
    {
        public static void Main()
        {
            StreamMain stream = new MemoryStream();
            for (int i = 0; i < 122; i++)
            {
                stream.WriteByte((byte)i);
            }







            stream.Position = 0;
            byte[] buffer = new byte[10];
            int offset = 0;
            int count = 5;
            int res = 0;
            do
            {
                res = stream.Read(buffer, offset, count);
                for (int i = 0; i < res; i++)
                {
```

```

        ConsoleApp1.Write(string.Format("{0,5}", buffer[i]));
    }
    ConsoleApp1.WriteLine();
}
while (res != 0);
}
}
}
}

```

Используем Visual Studio, получаем файл:

	ConsoleApp1.deps.json	14/03/2024 11:10 CH	JSON Source File	1 KB
	ConsoleApp1.dll	15/03/2024 1:25 CH	Application exten...	9 KB
	ConsoleApp1.exe	15/03/2024 1:25 CH	Application	140 KB
	ConsoleApp1.pdb	15/03/2024 1:25 CH	Program Debug D...	13 KB
	ConsoleApp1.runtimeconfig.json	24/02/2024 1:02 CH	JSON Source File	1 KB
	test.txt	14/03/2024 11:25 CH	Text Document	1 KB

3.4.2 Python

Для языка Python нам потребуются библиотеки pyinstaller, чтобы преобразовывать исходный код в бинарные файлы. Для установки можем выполнить следующую команду:

```
pip install pyinstaller
```

```

import angr
import sys

def main(argv):
    path_to_binary = './02_angr_find_condition'
    project = angr.Project(path_to_binary)
    initial_state = project.factory.entry_state(
        add_options = {
            angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
            angr.options.SYMBOL_FILL_UNCONSTRAINED_REGISTERS}
    )
    simulation = project.factory.simgr(initial_state)

    def is_successful(state):
        stdout_output = state.posix.dumps(sys.stdout.fileno())
        if b'Good Job.' in stdout_output:
            return True
        return False

    def should_abort(state):
        stdout_output = state.posix.dumps(sys.stdout.fileno())

```

```

    if b'Try again.' in stdout_output:
        return True
    return False

simulation.explore(find=is_successful, avoid=should_abort)

if simulation.found:
    solution_state = simulation.found[0]
    solution = solution_state.posix.dumps(sys.stdin.fileno())
    print("[+] Success! Solution is: {}".format(solution.decode("utf-8")))
else:
    print('Could not find the solution')

if __name__ == '__main__':
    main(sys.argv)

```

Далее используем следующую команду, чтобы получить бинарный файл:

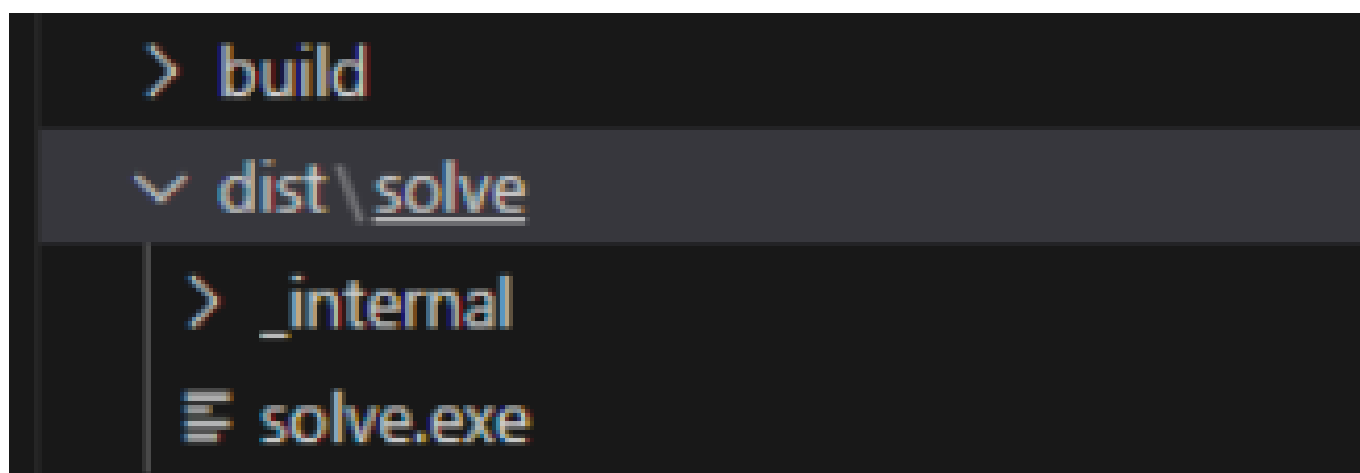
```
pyinstaller [file-name]
```

```

PS D:\angr_ctf-master\C> pyinstaller .\solve.py
479 INFO: PyInstaller: 6.4.0, contrib hooks: 2024.2
480 INFO: Python: 3.12.0
512 INFO: Platform: Windows-10-10.0.19042-SP0
513 INFO: wrote D:\angr_ctf-master\C\solve.spec
522 INFO: Extending PYTHONPATH with paths
['D:\\angr_ctf-master\\C']
1068 INFO: checking Analysis
1068 INFO: Building Analysis because Analysis-00.toc is non existent
1068 INFO: Initializing module dependency graph...

```

Получаем бинарный файл:



Приведём ещё ряд примеров работы ANGR.

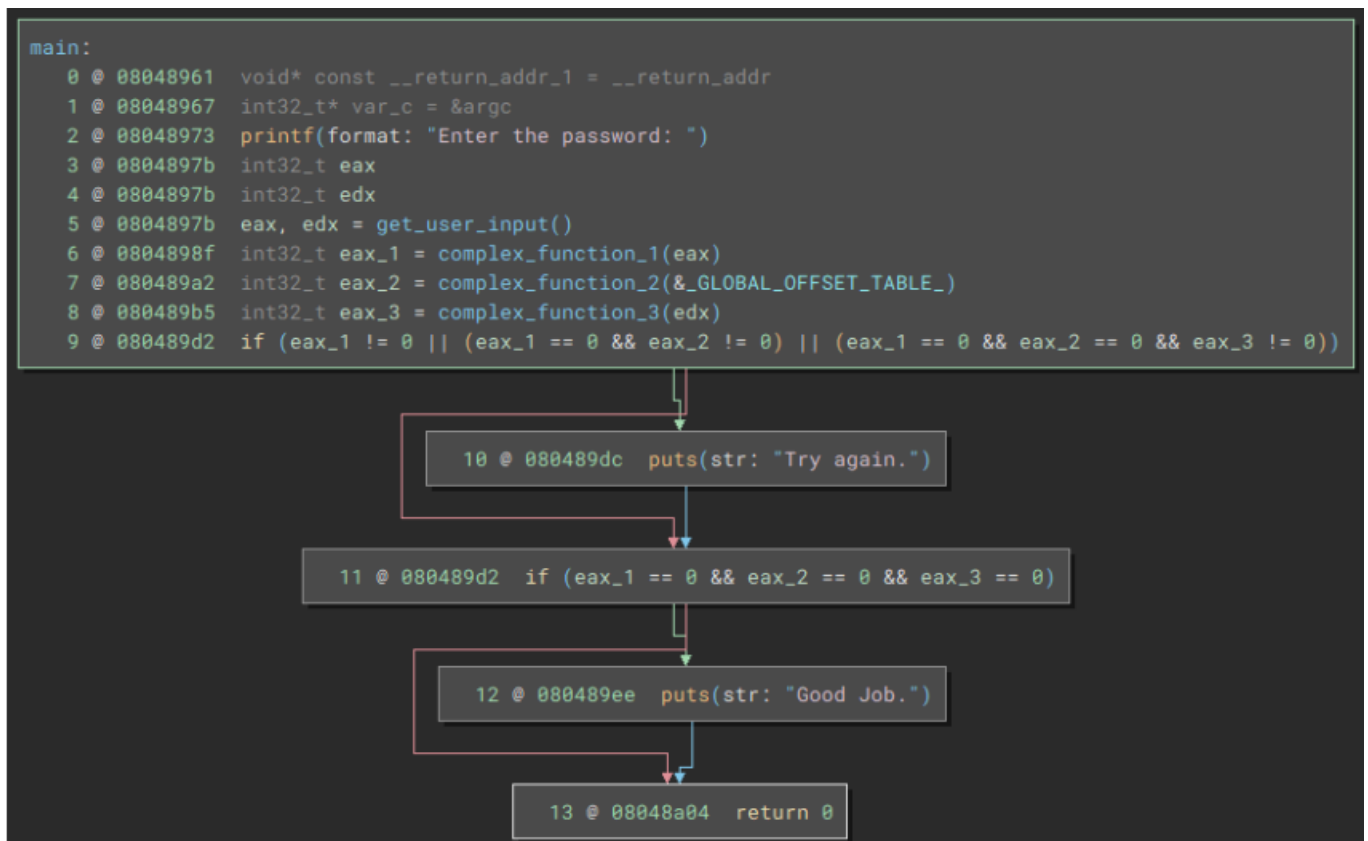
4 Пример работы ANGR

4.1 Пример 1

У нас есть программа в двоичном файле, которая запрашивает у нас пароль для входа в систему. Мы можем использовать Angr и символическое выполнение, чтобы получить секретный пароль.

Первый шаг, нам нужно дизассемблировать файл, чтобы получить представление о программе.

На этом шаге мы используем двоичный файл `pinja` для разборки файла



В коде есть функция `get_user_input`, которая используется для получения входных данных от пользователя, нас интересует эта часть

```

get_user_input:
0804890c  push    ebp {__saved_ebp}
0804890d  mov     ebp, esp {__saved_ebp}
0804890f  sub     esp, 0x18
08048912  mov     ecx, dword [gs:0x14]
08048919  mov     dword [ebp-0xc {var_10}], ecx
0804891c  xor     ecx, ecx {0x0}
0804891e  lea     ecx, [ebp-0x10 {var_14}]
08048921  push    ecx {var_14} {var_20}
08048922  lea     ecx, [ebp-0x14 {var_18}]
08048925  push    ecx {var_18} {var_24}
08048926  lea     ecx, [ebp-0x18 {var_1c}]
08048929  push    ecx {var_1c} {var_28}
0804892a  push    data_8048a93 {var_2c} {"%x %x %x"}
0804892f  call    __isoc99_scanf
08048934  add     esp, 0x10
08048937  mov     ecx, dword [ebp-0x18 {var_1c}]
0804893a  mov     eax, ecx
0804893c  mov     ecx, dword [ebp-0x14 {var_18}]
0804893f  mov     ebx, ecx
08048941  mov     ecx, dword [ebp-0x10 {var_14}]
08048944  mov     edx, ecx
08048946  nop
08048947  mov     ecx, dword [ebp-0xc {var_10}]
0804894a  xor     ecx, dword [gs:0x14]
08048951  je      0x8048958

```

```

08048958  leave   {__saved_ebp}
08048959  retn    {__return_addr}

```

```

08048953  call    __stack_chk_fail
{ Does not return }

```

Мы видим, что значение input будет храниться в регистрах eax, ebx, edx, поэтому, чтобы получить пароль, нам нужно ввести символ в эти регистры

```

# Create a symbolic bitvector (the datatype Angr uses to inject symbolic
# values into the binary.) The first parameter is just a name Angr uses
# to reference it.
# We will have to construct multiple bitvectors. Copy the two lines below
# and change the variable names. To figure out how many (and of what size)
# We need, disassemble the binary and determine the format parameter passed
# to scanf.
password_size_in_bits = 32 # :integer
eax = claripy.BVS('eax', password_size_in_bits)
ebx = claripy.BVS('ebx', password_size_in_bits)
edx = claripy.BVS('edx', password_size_in_bits)

# Set a register to a symbolic value. This is one way to inject symbols into
# the program.

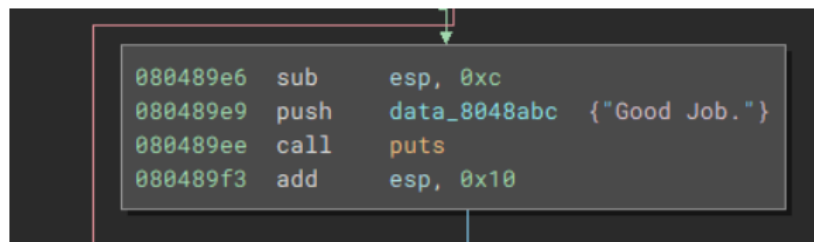
```

```

# initial_state.regs stores a number of convenient attributes that reference
→ registers by name.
# We will have to set multiple registers to distinct bitvectors. Copy and
# paste the line below and change the register. To determine which registers
# to inject which symbol, disassemble the binary and look at the instructions
# immediately following the call to scanf.
initial_state.regs.eax = eax
initial_state.regs.ebx = ebx
initial_state.regs.edx = edx

```

Стоит заметить, что нас интересует значение этих регистров только в том состоянии, когда пароль успешно введён. Итак, когда мы получим состояние успешно введённого пароля, запишем значение регистра в этом состоянии.



```

if simulation.found:
    solution_state = simulation.found[0]

    # Solve for the symbolic values. If there are multiple solutions, we only
    # care about one, so we can use eval, which returns any (but only one)
    # solution. Pass eval the bitvector we want to solve for.
    solution0 = format(solution_state.solver.eval(eax), 'x')
    solution1 = format(solution_state.solver.eval(ebx), 'x')
    solution2 = format(solution_state.solver.eval(edx), 'x')

    # Aggregate and format the solutions we computed above, and then print
    # the full string. Pay attention to the order of the integers, and the
    # expected base (decimal, octal, hexadecimal, etc).
    solution = solution0 + " " + solution1 + " " + solution2
    print("[+] Success! Solution is: {}".format(solution))
else:
    raise Exception('Could not find the solution')

```

Результат выполнения программы: успешно получен верный пароль.



4.2 Пример 2

У нас есть программа в двоичном файле, которая запросит у нас пароль для входа в систему.

Нам нужно использовать Angr и symbolic extract, чтобы получить секретный пароль. На первом шаге, нам нужен дизассемблированный файл, чтобы получить представление о программе. На этом шаге мы используем angr-management для разборки файла.

```
int (32 bits) main()
Args: ()
s_20 -0x20
s_1c -0x1c
s_10 -0x10
s_c -0xc
s_8 -0x8
s_4 -0x4
s_0 0x0
ret_addr 0x4
s_-8 0x8
0804870c lea     ecx, [esp+0x4] {s_-8}
08048710 and     esp, 0xffffffff
08048713 push   dword ptr [ecx-0x4] {s_0}
08048716 push   ebp
08048717 mov    ebp, esp
08048719 push   ecx
0804871a sub    esp, 0x4
0804871d sub    esp, 0xc
08048720 push   0x80487de "Enter the password: "
08048725 call   printf
0804872a add    esp, 0x10
0804872d call   handle_user
08048732 mov    eax, 0x0
08048737 mov    ecx, dword ptr [ebp-0x4] {s_8}
0804873a leave
0804873b lea    esp, [ecx-0x4]
0804873e ret
```

Мы видим функцию `handle_user`, которая используется для обработки входных данных от пользователя, нас интересует эта часть

```

void handle_user()
Args: ()
s_2c -0x2c
s_28 -0x28
s_24 -0x24
s_20 -0x20
s_18 -0x18
s_10 -0x10
s_c -0xc
s_0 0x0
ret_addr 0x4
08048690 push    ebp
08048691 mov     ebp, esp
08048693 sub     esp, 0x18
08048696 sub     esp, 0x4
08048699 lea     eax, [ebp-0x10] {s_10}
0804869c push    eax
0804869d lea     eax, [ebp-0xc] {s_c}
080486a0 push    eax
080486a1 push    0x80487c3 "%u %u"
080486a6 call   __isoc99_scanf
080486ab add     esp, 0x10
080486ae mov     eax, dword ptr [ebp-0xc] {s_c}
080486b1 sub     esp, 0xc
080486b4 push    eax
080486b5 call   complex_function0
080486ba add     esp, 0x10
080486bd mov     dword ptr [ebp-0xc] {s_c}, eax
080486c0 mov     eax, dword ptr [ebp-0x10] {s_10}
080486c3 sub     esp, 0xc
080486c6 push    eax
080486c7 call   complex_function1
080486cc add     esp, 0x10
080486cf mov     dword ptr [ebp-0x10] {s_10}, eax
080486d2 mov     eax, dword ptr [ebp-0xc] {s_c}
080486d5 cmp     eax, 0x7c315173
080486da jne     0x80486e6

```

Мы хотим начать после вызова scanf. Обратите внимание, что это происходит в середине функции. Поэтому мы должны уделить особое внимание тому, с чего мы начинаем, иначе мы введем условие, при котором стек будет настроен неправильно. Чтобы определить, с чего начать после scanf, нам нужно посмотреть на разборку вызова и инструкцию, непосредственно следующую за ним:

```

08048696 sub     esp, 0x4
08048699 lea     eax, [ebp-0x10] {s_10}
0804869c push    eax
0804869d lea     eax, [ebp-0xc] {s_c}
080486a0 push    eax
080486a1 push    0x80487c3 "%u %u"
080486a6 call   __isoc99_scanf
080486ab add     esp, 0x10

```

Мы начинаем с инструкции, которая следует за scanf (add esp, 0x10). Рассмотрим что делает "add esp, 0x10". это связано с параметрами scanf, которые помещаются в стек перед вызовом

функции. Учитывая, что мы не вызываем `scanf` в нашем моделировании Angr, с чего нам следует начать тестирование. Адрес, с которого мы начинаем, - это адрес `mov eax, dword ptr [ebp-0xc]`.

```
080486ae  mov     eax, dword ptr [ebp-0xc] {s_c}
```

```
start_address = 0x080486ae
initial_state = project.factory.blank_state(
    addr=start_address,
    add_options = { angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
                    angr.options.SYMBOL_FILL_UNCONSTRAINED_REGISTERS}
)
```

Мы переходим к середине функции. Следовательно, нам нужно учитывать, как функция использует стек. Вторая инструкция функции такова:

```
08048691  mov     ebp, esp
```

В этот момент он выделяет ту часть стекового фрейма, на которую мы планируем нацелиться:

```
08048693  sub     esp, 0x18
```

Поскольку мы начинаем после `scanf`, мы пропускаем этот шаг построения стека. Чтобы компенсировать это, нам нужно создать стек самостоятельно. Давайте начнем с инициализации `ebp` точно так же, как это делает программа.

```
initial_state.regs.ebp = initial_state.regs.esp
```

После этого мы собираемся уменьшить указатель стека на значение 8 (помните, что стек растет вниз, поэтому мы фактически увеличиваем его размер), чтобы обеспечить заполнение, прежде чем помещать наши символические значения в стек.

```
padding_length_in_bytes = 0x08
initial_state.regs.esp -= padding_length_in_bytes
```

Теперь пришло время создать наши символичные битовые векторы и поместить их в стек. Помните, что программа ожидает два целых значения без знака (мы поняли это по строке формата `%u %u`).

```
080486a1  push    0x80487c3 "%u %u"
```

Таким образом, размер символических битовых векторов будет составлять 32 бита, поскольку это размер целого числа без знака в архитектуре x86.

```
password_size_in_bits = 32
password0 = claripy.BVS('password0', password_size_in_bits)
password1 = claripy.BVS('password1', password_size_in_bits)

initial_state.stack_push(password0)
initial_state.stack_push(password1)
```

Конечно, нам интересно значение состояния, когда пароли успешно вставлены. Итак, когда мы получим значение успешного состояния

```
loc_0x80486f8:
080486f8  sub     esp, 0xc
080486fb  push    0x80487d4 "Good Job."
08048700  call    puts
08048705  add     esp, 0x10
08048708  nop
```

Результат выполнения программы: успешно получен правильный пароль!!!

```
[+] Success! Solution is: 2089710965 12847883
```

4.3 Пример 3

У нас есть программа в двоичном файле, которая запросит у нас пароли для входа в систему. Нам нужно использовать Angr и symbolic extract для получения секретных паролей

Первый шаг, нам нужен файл дизассемблирования, чтобы получить представление о программе. На этом шаге мы используем angr-management для разборки файла

```
int (32 bits) main()
Args: ()
s_40 -0x40
s_3c -0x3c
s_38 -0x38
s_34 -0x34
s_30 -0x30
s_2c -0x2c
s_28 -0x28
s_24 -0x24
s_1c -0x1c
s_10 -0x10
s_8 -0x8
s_4 -0x4
s_0 0x0
ret_addr 0x4
s_-8 0x8
080485bf lea     ecx, [esp+0x4] {s_-8}
080485c3 and     esp, 0xffffffff
080485c6 push    dword ptr [ecx-0x4] {ret_addr}
080485c9 push    ebp
080485ca mov     ebp, esp
080485cc push    ecx
080485cd sub     esp, 0x14
080485d0 sub     esp, 0x4
080485d3 push    0x21
080485d5 push    0x0
080485d7 push    user_input
080485dc call    memset
080485e1 add     esp, 0x10
080485e4 sub     esp, 0xc
080485e7 push    0x804872e "Enter the password: "
080485ec call    printf
080485f1 add     esp, 0x10
080485f4 sub     esp, 0xc
080485f7 push    0xab232d8
080485fc push    0xab232d0
08048601 push    0xab232c8
08048606 push    user_input
0804860b push    0x8048743 "%8s %8s %8s %8s"
08048610 call    __isoc99_scanf
08048615 add     esp, 0x20
08048618 mov     dword ptr [ebp-0xc] {s_10}, 0x0
0804861f jmp     0x804864e
```

Мы можем видеть, что первый блок устанавливает стек и вызывает `scanf()`. Мы знаем, что он принимает в качестве входных данных строку формата и ряд аргументов, которые зависят от формата строки. Используемое здесь соглашение о вызове (cdecl) диктует, что аргументы функций должны быть помещены в стек справа налево, поэтому мы знаем, что последним параметром, помещенным в стек непосредственно перед вызовом `scanf()`, будет сама строка, которая в данном случае равна `%8s %8s %8s %8s`.

```
0804860b  push    0x8048743 "%8s %8s %8s %8s"
```

Адрес, с которого мы начинаем, - это адрес `MOV DWORD [EBP - 0xC], 0x0` после вызова `scanf()` и его последующего добавления `ESP, 0x20`. После настройки нашего пустого состояния мы создаем четыре символьных битовых вектора, которые заменят наши входные данные. Обратите внимание, что их размер равен 64 битам, поскольку строки по 8 байт длиной.

```
password_size_in_bits = 64

password0 = claripy.BVS('password0', password_size_in_bits)
password1 = claripy.BVS('password1', password_size_in_bits)
password2 = claripy.BVS('password2', password_size_in_bits)
password3 = claripy.BVS('password3', password_size_in_bits)
```

Давайте обратим внимание на эти четыре адреса (три показанных и адрес `user_input`)

```
080485bf  lea     ecx, [esp+0x4] {s_-8}
080485c3  and     esp, 0xffffffff
080485c6  push    dword ptr [ecx-0x4] {ret_addr}
080485c9  push    ebp
080485ca  mov     ebp, esp
080485cc  push    ecx
080485cd  sub     esp, 0x14
080485d0  sub     esp, 0x4
080485d3  push    0x21
080485d5  push    0x0
080485d7  push    user_input
080485dc  call    memset
080485e1  add     esp, 0x10
080485e4  sub     esp, 0xc
080485e7  push    0x804872e "Enter the password: "
080485ec  call    printf
080485f1  add     esp, 0x10
080485f4  sub     esp, 0xc
080485f7  push    0xab232d8
080485fc  push    0xab232d0
08048601  push    0xab232c8
08048606  push    user_input
0804860b  push    0x8048743 "%8s %8s %8s %8s"
08048610  call    __isoc99_scanf
08048615  add     esp, 0x20
08048618  mov     dword ptr [ebp-0xc] {s_10}, 0x0
0804861f  jmp     0x804864e
```

Мы определяем адрес `0xab232c0`, по которому будет сохранен первый символьный битовый вектор. Остальные три символьных битовых вектора должны храниться соответственно в `0xab232c8`, `0xab232d0` и `0xab232d8`, которые являются `password0_address + 0x8`, `+ 0x10` и `+ 0x18`.

```
password0_address = 0xab232c0
initial_state.memory.store(password0_address, password0)
initial_state.memory.store(password0_address + 0x8, password1)
initial_state.memory.store(password0_address + 0x10, password2)
initial_state.memory.store(password0_address + 0x18, password3)
```

Здесь мы могли бы просто принять к сведению адрес блока кода, который приводит к “Good job”. и двух блоков кода, которые приводят к “Попробуйте еще раз”., но мы можем просто определить две функции `is_successful`, `should_abort`, которые проверят выходные данные программы и позволят `angr` принять решение отбросить или нет этот путь.

```
def is_successful(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    if b'Good Job.' in stdout_output:
        return True
    return False # :boolean

def should_abort(state):
    stdout_output = state.posix.dumps(sys.stdout.fileno())
    if b'Try again.' in stdout_output:
        return True
    return False # :boolean
```

Мы проверяем, достигло ли какое-либо состояние желаемого пути к коду, мы конкретизируем символические битовые векторы в реальные строки (на самом деле это байты, мы расшифруем их как строки, когда будем печатать), мы объединяем их и, наконец, печатаем решение.

```
solution0 = solution_state.solver.eval(password0, cast_to=bytes)
solution1 = solution_state.solver.eval(password1, cast_to=bytes)
solution2 = solution_state.solver.eval(password2, cast_to=bytes)
solution3 = solution_state.solver.eval(password3, cast_to=bytes)

solution = solution0 + b" " + solution1 + b" " + solution2 + b" " + solution3
```

Результат выполнения программы: успешно получен правильный пароль!!!

```
[+] Success! Solution is: OJQVXIVX LLEA00DW UVCWUWVC AJXJMVKA
```