Analisi simbolica con angr

Enrico Bacis

- Hackers eat Pizza 2019 -

$ whoami

Enrico Bacis
PhD student @ UniBG

Organizzazioni:

- Unibg Seclab
- BgLUG
- Hacklab BITM

Ambiti di ricerca:

- Access Control
- Database Security
- Mobile Security

Outline

- Tecniche di Binary Analysis
- Come funziona l’analisi simbolica
- Il framework angr
- Utilizzo base
- Altre informazioni su angr

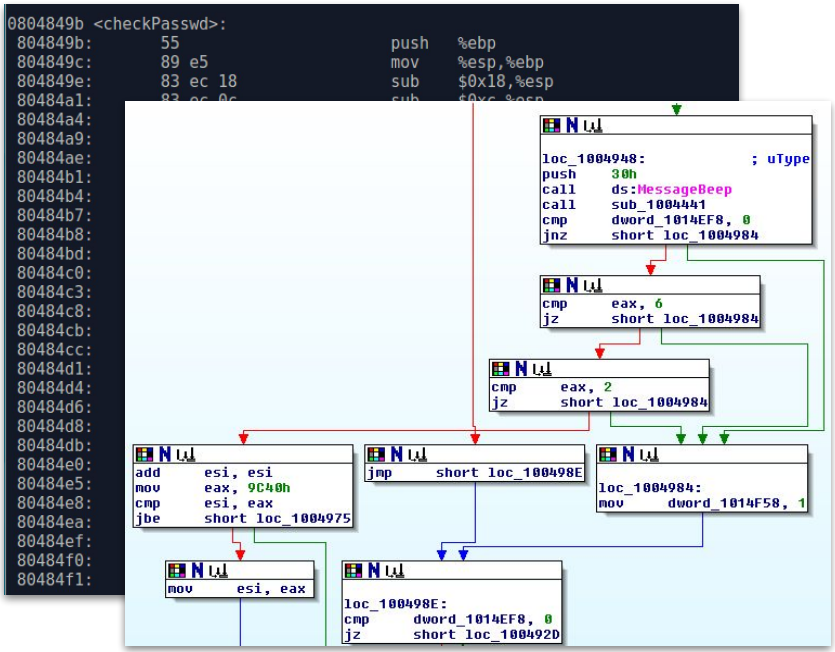
Tecniche di Binary Analysis

Static Analysis

```
0804849b <checkPasswd>:
 804849b:       55                      push   %ebp
 804849c:       89 e5                   mov    %esp,%ebp
 804849e:       83 ec 18                sub    $0x18,%esp
 80484a1:       83 ec 0c                sub    $0xc,%esp
 80484a4:       68 b0 85 04 08          push   $0x80485b0
 80484a9:       e8 a2 fe ff ff          call   8048350 <printf@plt>
 80484ae:       83 c4 10                add    $0x10,%esp
 80484b1:       83 ec 0c                sub    $0xc,%esp
 80484b4:       8d 45 e8                lea    -0x18(%ebp),%eax
 80484b7:       50                      push   %eax
 80484b8:       e8 a3 fe ff ff          call   8048360 <gets@plt>
 80484bd:       83 c4 10                add    $0x10,%esp
 80484c0:       83 ec 08                sub    $0x8,%esp
 80484c3:       68 c4 85 04 08          push   $0x80485c4
 80484c8:       8d 45 e8                lea    -0x18(%ebp),%eax
 80484cb:       50                      push   %eax
 80484cc:       e8 6f fe ff ff          call   8048340 <strcmp@plt>
 80484d1:       83 c4 10                add    $0x10,%esp
 80484d4:       85 c0                   test   %eax,%eax
 80484d6:       74 12                   je     80484ea <checkPasswd+0x4f>
 80484d8:       83 ec 0c                sub    $0xc,%esp
 80484db:       68 cc 85 04 08          push   $0x80485cc
 80484e0:       e8 8b fe ff ff          call   8048370 <puts@plt>
 80484e5:       83 c4 10                add    $0x10,%esp
 80484e8:       eb 05                   jmp    80484ef <checkPasswd+0x54>
 80484ea:       e8 03 00 00 00          call   80484f2 <granted>
 80484ef:       90                      nop
 80484f0:       c9                      leave
 80484f1:       c3                      ret
```

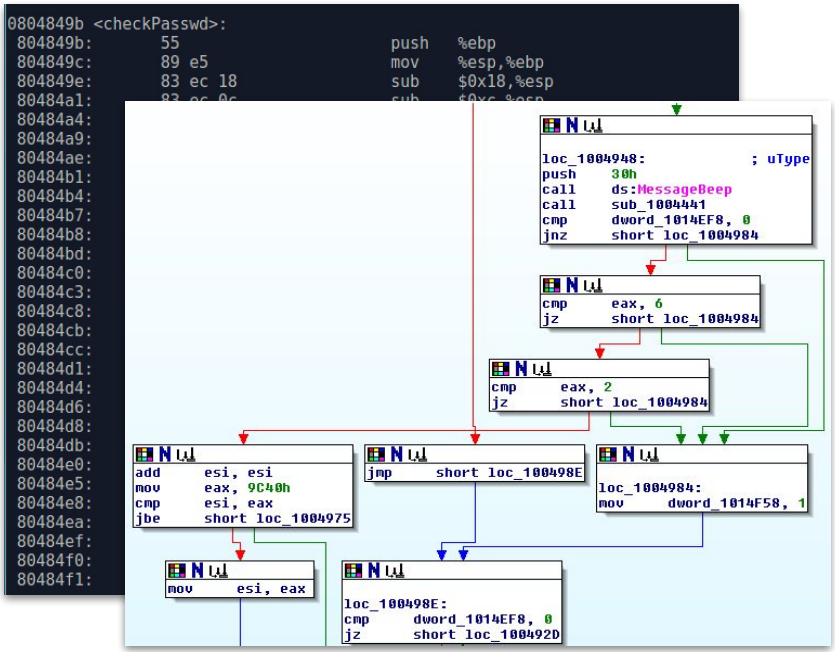
- objdump

Static Analysis



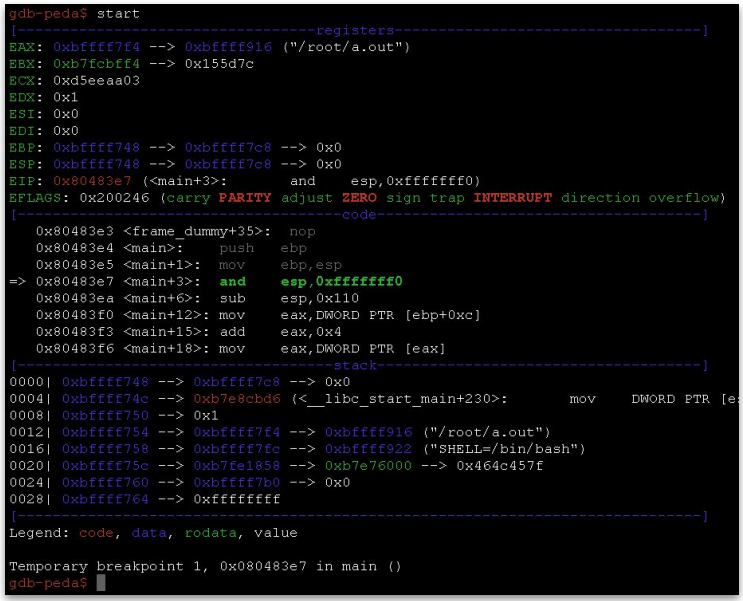
- objdump
- IDA

Static Analysis



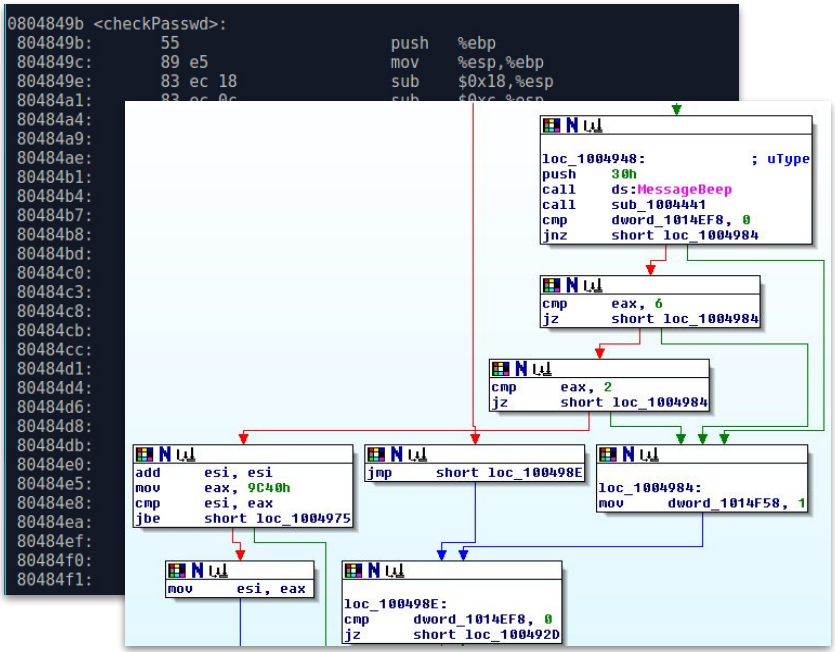
- objdump
- IDA

Dynamic Analysis



- gdb (& friends)

Static Analysis

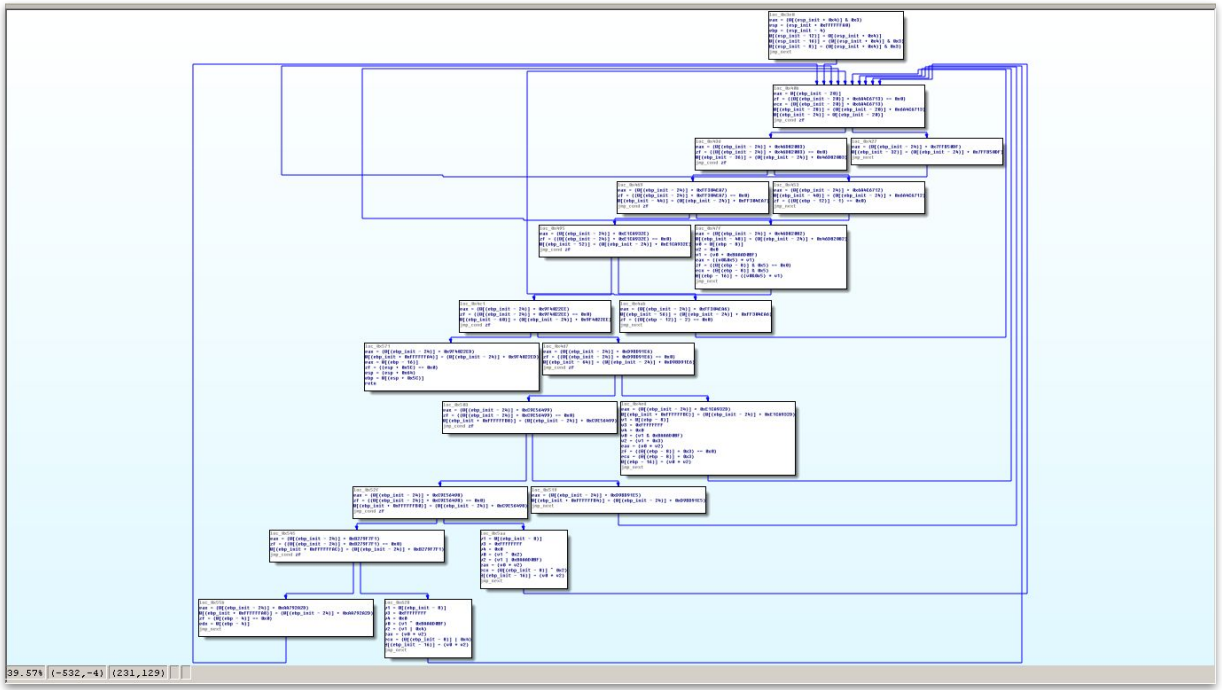


- objdump
- IDA

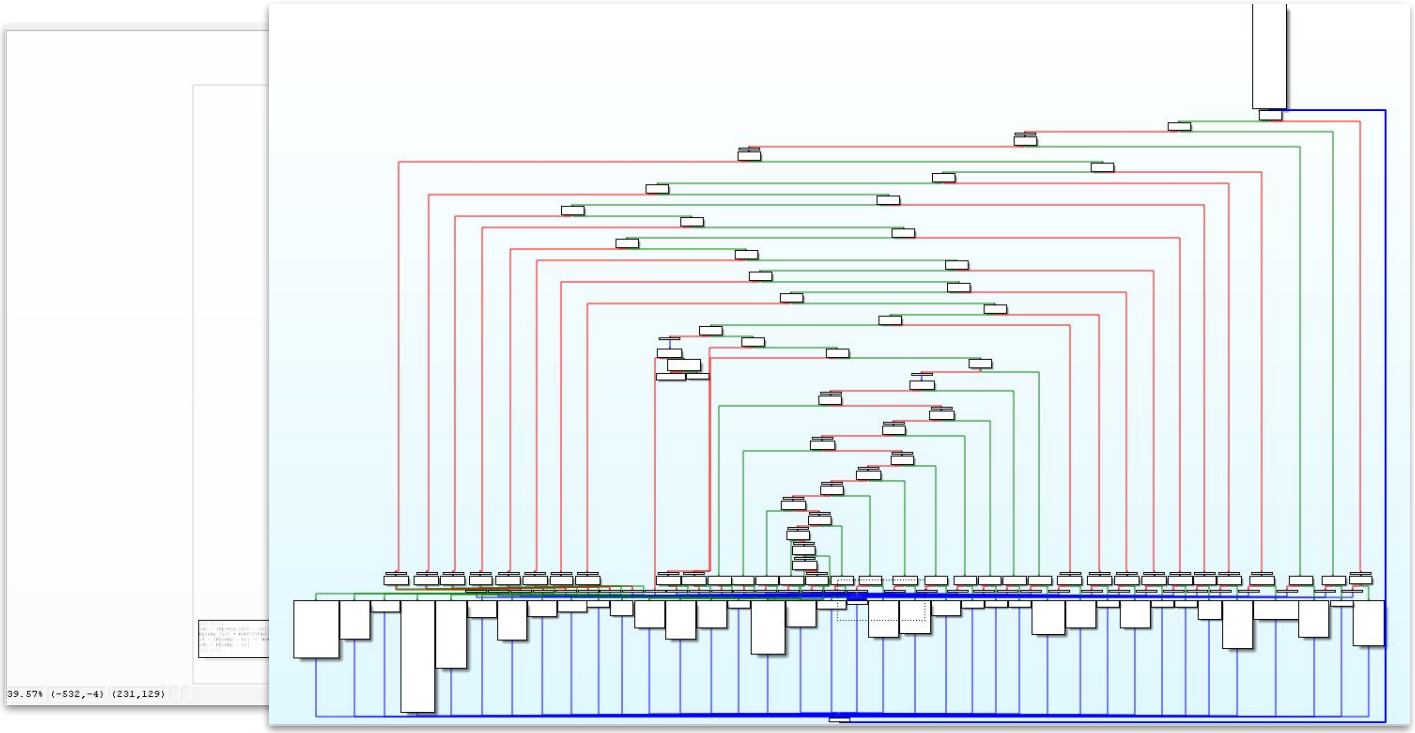
Dynamic Analysis

- gdb (& friends)
- radare2

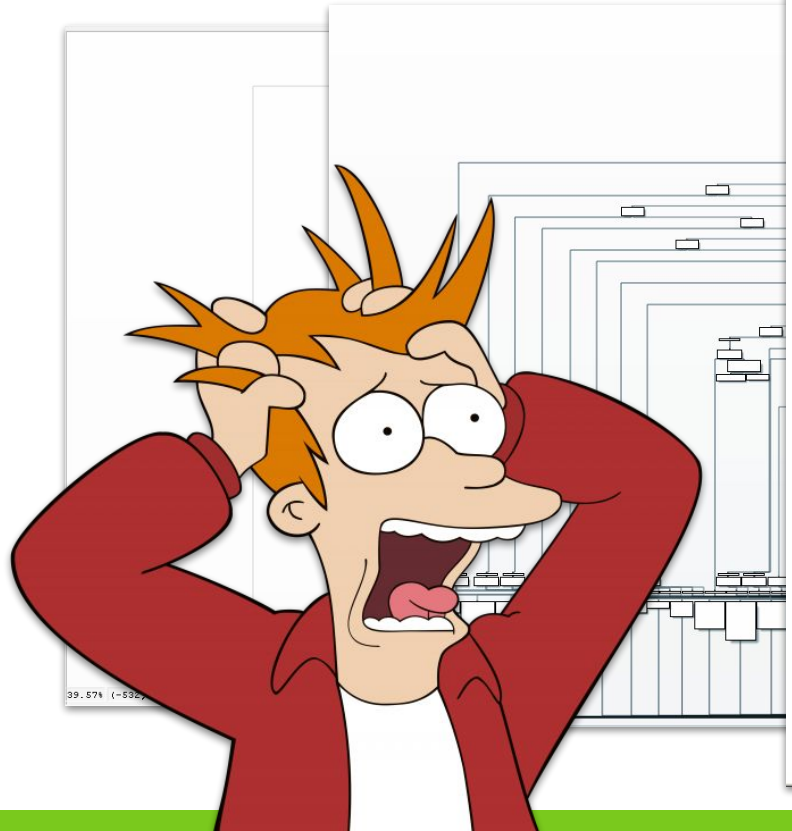
Limiti

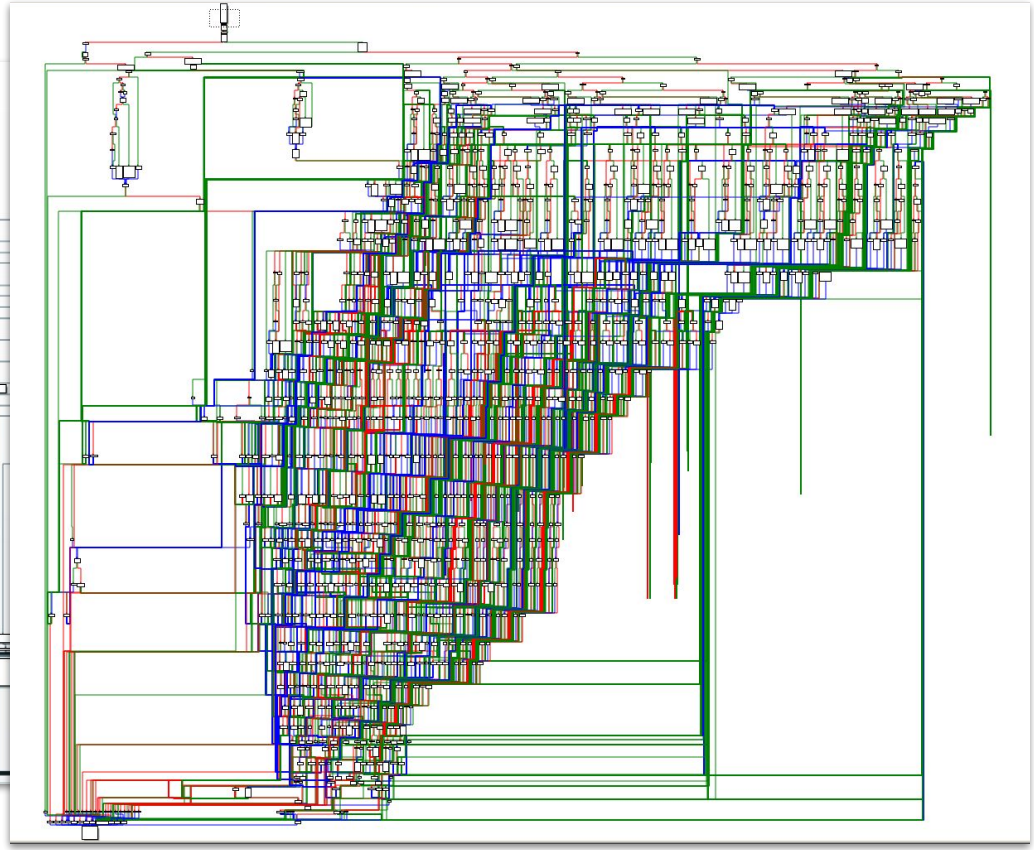


Limiti



Limiti

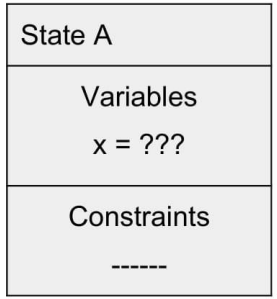




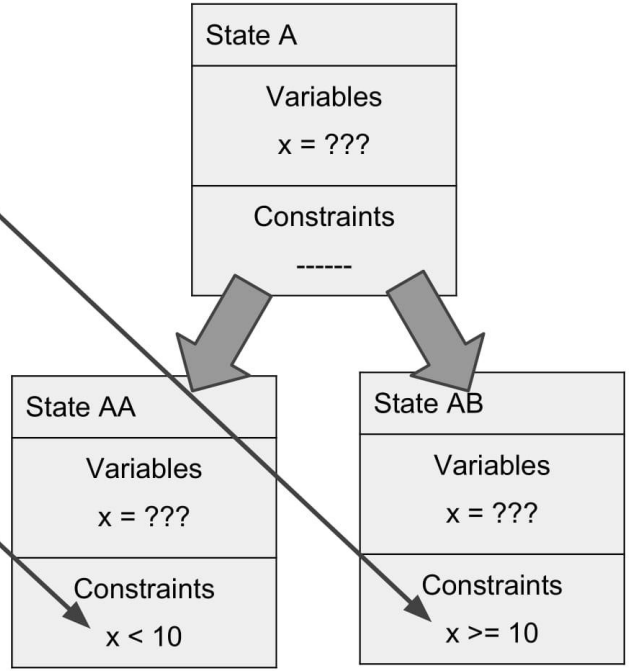
Come funziona l'analisi simbolica

(a.k.a. quattro slide rubate da qualsiasi presentazione su angr mai fatta)

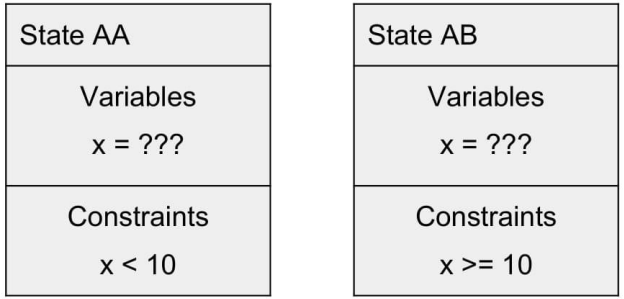
```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



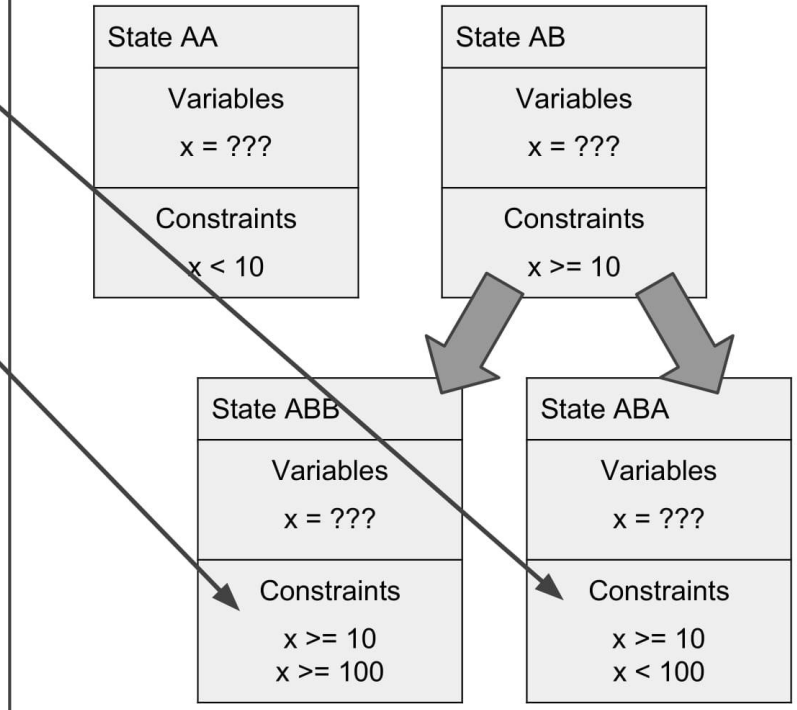
```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```



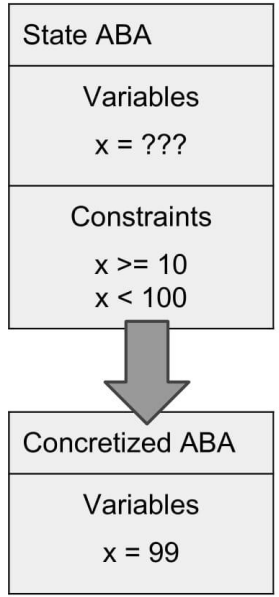
```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

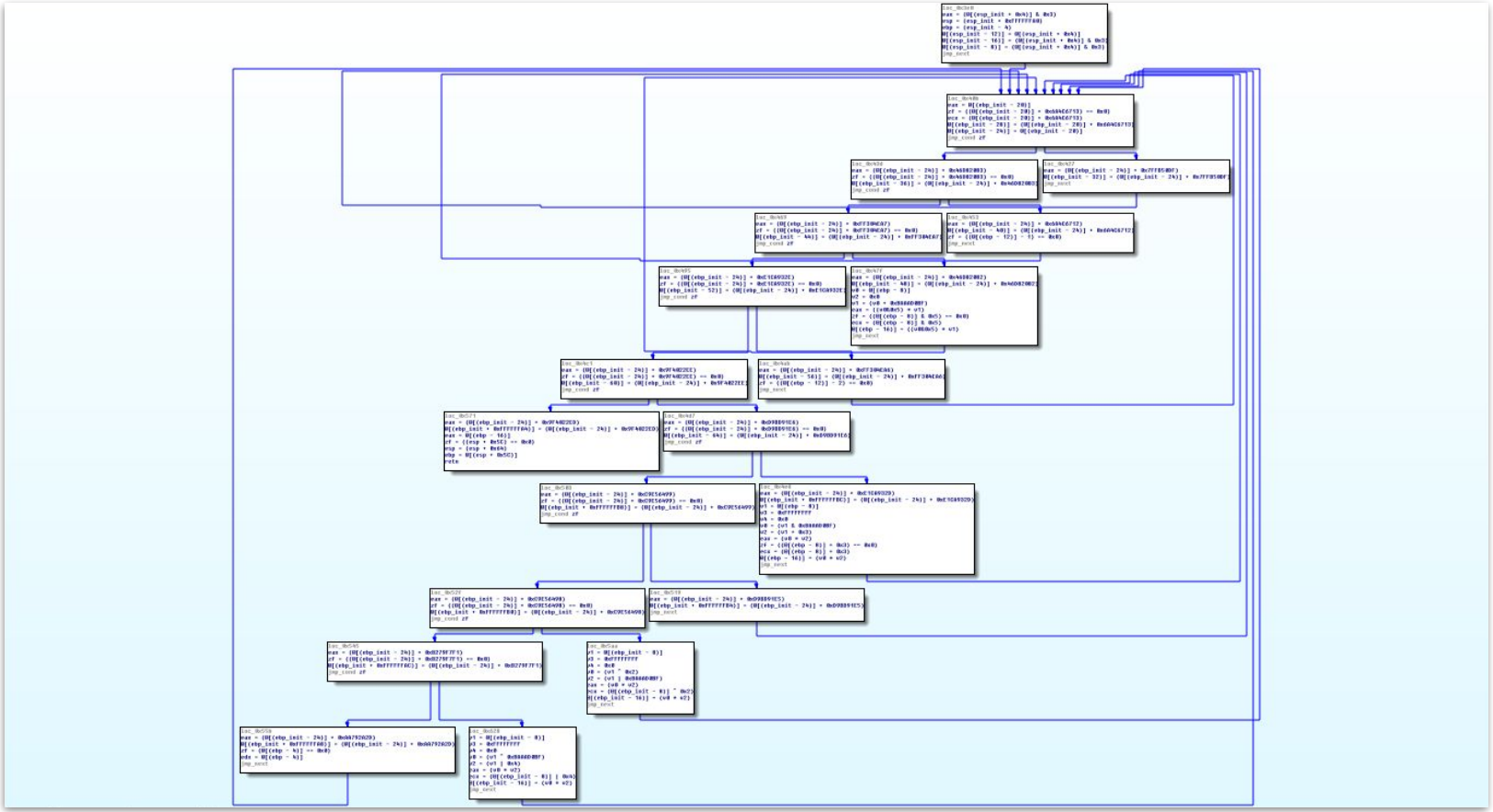


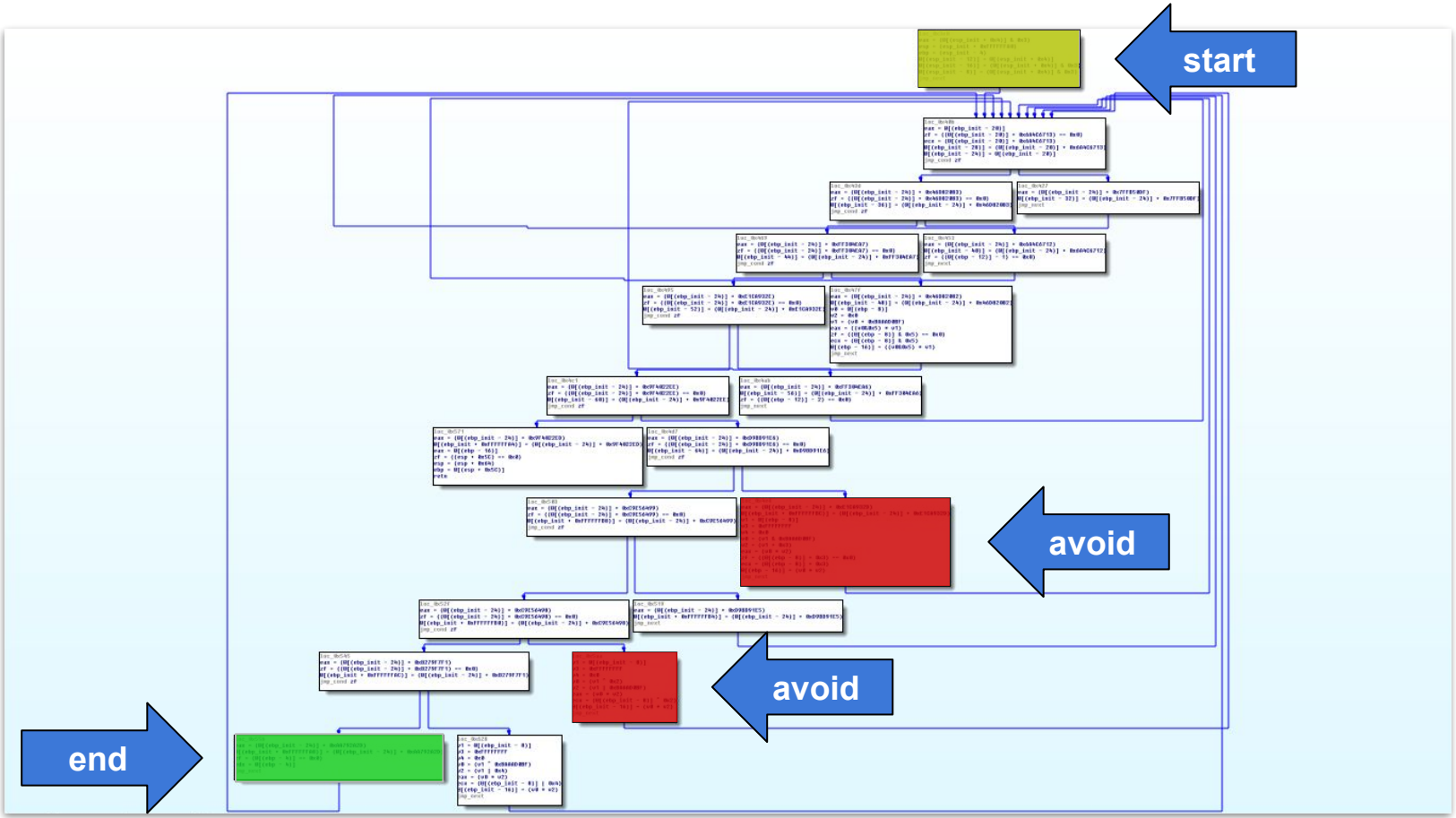
```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

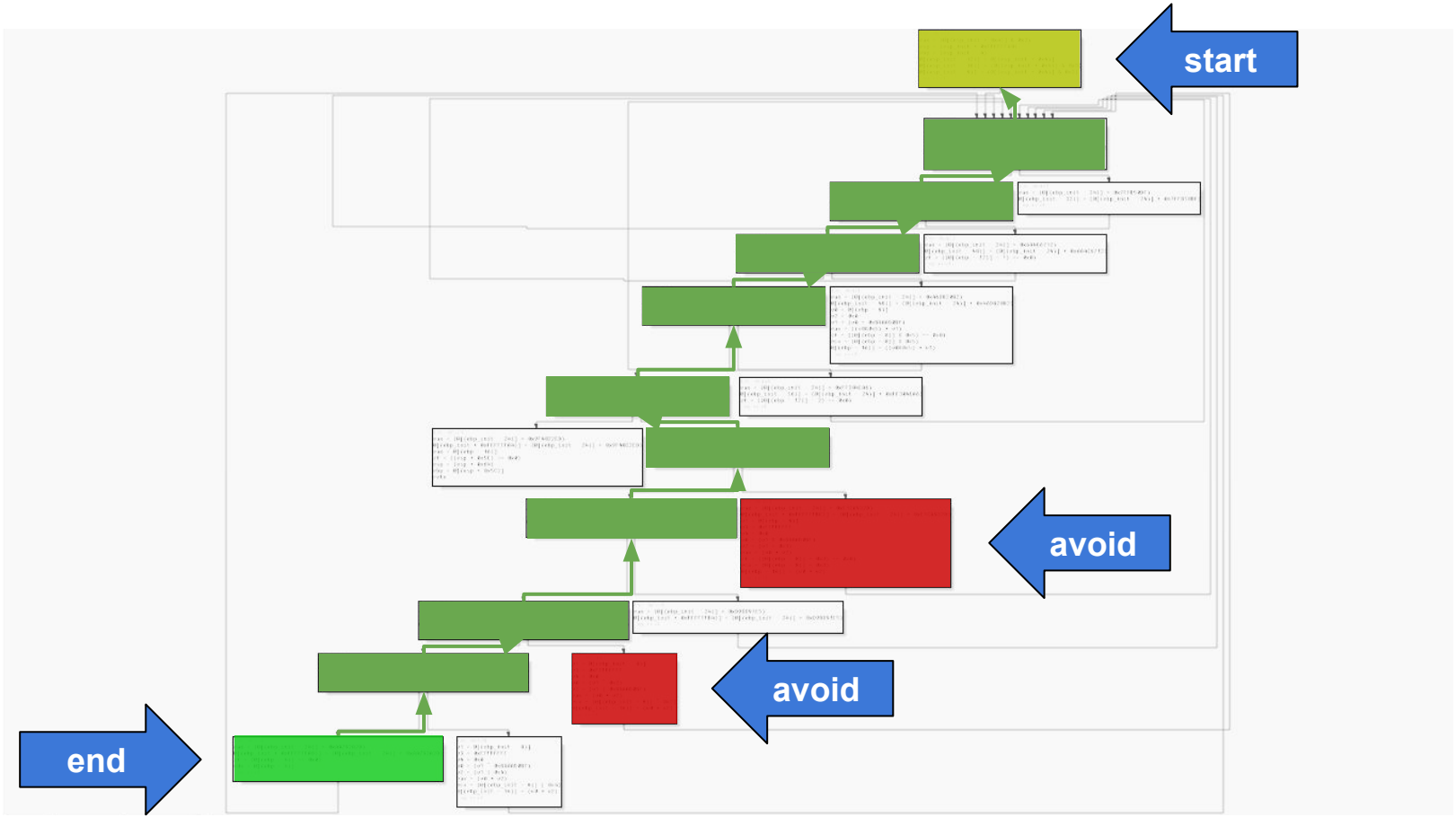


```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```






start
avoid
avoid
end


start
avoid
avoid
end

angr

https://angr.io

Cosa è angr

- Framework per binary analysis in python che combina analisi statica e dinamica simbolica ("*concolic analysis*" da ***conc***rete e symb***olic***)
- Sviluppato da UCSB (terzo posto DARPA Cyber Grand Challenge)
- Basato su VEX (Valgrind), supporta moltissime architetture
- Flusso di analisi:
  - L'eseguibile viene caricato nel framework
  - Il codice binario viene trasformato in IR (intermediate representation)
  - L'analisi viene eseguita

Utilizzo base

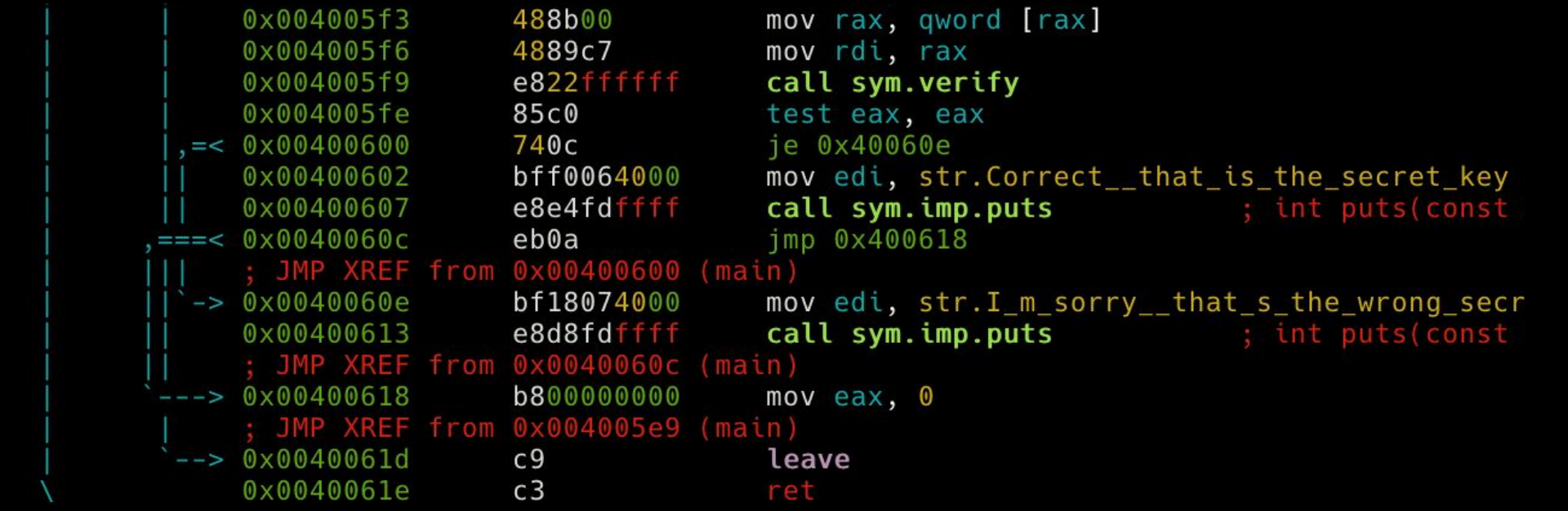
ais3 crackme

- https://github.com/angr/angr-doc/tree/master/examples/ais3_crackme
- Si esegue il binario con un argomento
- Se l'argomento è corretto
  - stdout: "`Correct! that is the secret key!`"
- Altrimenti
  - stdout: "`I'm sorry, that's the wrong secret key!`"

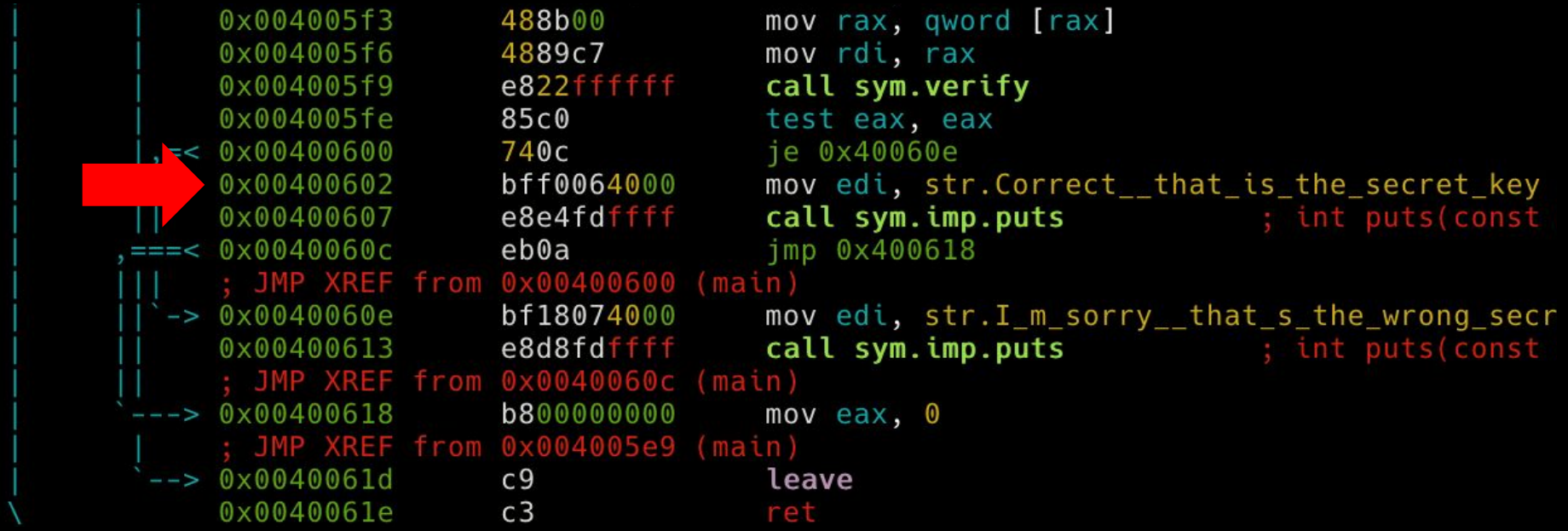
Target

```
[0x00400410]> s main
[0x004005c5]> pdf
/ (fcn) main 90
|   main ();
|           ; var int local_10h @ rbp-0x10
|           ; var int local_4h @ rbp-0x4
|           ; DATA XREF from 0x0040042d (entry0)
|           0x004005c5      55              push rbp
|           0x004005c6      4889e5          mov rbp, rsp
|           0x004005c9      4883ec10        sub rsp, 0x10
|           0x004005cd      897dfc          mov dword [local_4h], edi
|           0x004005d0      488975f0        mov qword [local_10h], rsi
|           0x004005d4      837dfc02        cmp dword [local_4h], 2     ; [0x2:4]=-1 ; 2
|       ,=< 0x004005d8      7411            je 0x4005eb
|       |   0x004005da      bfc8064000      mov edi, str.You_need_to_enter_the_secret_key ; 0x4006c8 ; "You need to enter the secret key!"
|       |   0x004005df      e80cfeffff      call sym.imp.puts           ; int puts(const char *s)
|       |   0x004005e4      b8ffffffff      mov eax, 0xffffffff         ; -1
|      ,==< 0x004005e9      eb32            jmp 0x40061d
|      ||   ; JMP XREF from 0x004005d8 (main)
|      |`-> 0x004005eb      488b45f0        mov rax, qword [local_10h]
|      |    0x004005ef      4883c008        add rax, 8
|      |    0x004005f3      488b00          mov rax, qword [rax]
|      |    0x004005f6      4889c7          mov rdi, rax
|      |    0x004005f9      e822ffffff      call sym.verify
|      |    0x004005fe      85c0            test eax, eax
|      |,=< 0x00400600      740c            je 0x40060e
|      ||   0x00400602      bff0064000      mov edi, str.Correct__that_is_the_secret_key ; 0x4006f0 ; "Correct! that is the secret key!"
|      ||   0x00400607      e8e4fdffff      call sym.imp.puts           ; int puts(const char *s)
|     ,===< 0x0040060c      eb0a            jmp 0x400618
|     |||   ; JMP XREF from 0x00400600 (main)
|     ||`-> 0x0040060e      bf18074000      mov edi, str.I_m_sorry__that_s_the_wrong_secret_key ; 0x400718 ; "I'm sorry, that's the wrong secret key!"
|     ||    0x00400613      e8d8fdffff      call sym.imp.puts           ; int puts(const char *s)
|     ||    ; JMP XREF from 0x0040060c (main)
|     `---> 0x00400618      b800000000      mov eax, 0
|      |    ; JMP XREF from 0x004005e9 (main)
|      `--> 0x0040061d      c9              leave
\           0x0040061e      c3              ret
```

Target


0x004005f3 488b00 mov rax, qword [rax]
0x004005f6 4889c7 mov rdi, rax
0x004005f9 e822ffffff call sym.verify
0x004005fe 85c0 test eax, eax
0x00400600 740c je 0x40060e
0x00400602 bff0064000 mov edi, str.Correct__that_is_the_secret_key
0x00400607 e8e4fdffff call sym.imp.puts ; int puts(const
0x0040060c eb0a jmp 0x400618
; JMP XREF from 0x00400600 (main)
0x0040060e bf18074000 mov edi, str.I_m_sorry__that_s_the_wrong_secr
0x00400613 e8d8fdffff call sym.imp.puts ; int puts(const
; JMP XREF from 0x0040060c (main)
0x00400618 b800000000 mov eax, 0
; JMP XREF from 0x004005e9 (main)
0x0040061d c9 leave
0x0040061e c3 ret

Target


0x004005f3 488b00 mov rax, qword [rax]
0x004005f6 4889c7 mov rdi, rax
0x004005f9 e822ffffff call sym.verify
0x004005fe 85c0 test eax, eax
0x00400600 740c je 0x40060e
0x00400602 bff0064000 mov edi, str.Correct__that_is_the_secret_key
0x00400607 e8e4fdffff call sym.imp.puts ; int puts(const
0x0040060c eb0a jmp 0x400618
; JMP XREF from 0x00400600 (main)
0x0040060e bf18074000 mov edi, str.I_m_sorry__that_s_the_wrong_secr
0x00400613 e8d8fdffff call sym.imp.puts ; int puts(const
; JMP XREF from 0x0040060c (main)
0x00400618 b800000000 mov eax, 0
; JMP XREF from 0x004005e9 (main)
0x0040061d c9 leave
0x0040061e c3 ret

```
import angr, claripy
project = angr.Project("./ais3_crackme")
```

```
import angr, claripy
project = angr.Project("./ais3_crackme")

# create an initial state with a symbolic bit vector as argv1
argv1 = claripy.BVS("argv1", 100*8)  # 100 bytes
initial_state = project.factory.entry_state(args=["./ais3_crackme", argv1])
```

```
import angr, claripy
project = angr.Project("./ais3_crackme")

# create an initial state with a symbolic bit vector as argv1
argv1 = claripy.BVS("argv1", 100*8)  # 100 bytes
initial_state = project.factory.entry_state(args=["./ais3_crackme", argv1])

# create a path group using the created initial state
sm = project.factory.simulation_manager(initial_state)

# symbolically execute the program until we reach the wanted value of the IP
sm.explore(find=0x400602) # find a way to reach the address
found = sm.found[0]
```

```
import angr, claripy
project = angr.Project("./ais3_crackme")

# create an initial state with a symbolic bit vector as argv1
argv1 = claripy.BVS("argv1", 100*8)  # 100 bytes
initial_state = project.factory.entry_state(args=["./ais3_crackme", argv1])

# create a path group using the created initial state
sm = project.factory.simulation_manager(initial_state)

# symbolically execute the program until we reach the wanted value of the IP
sm.explore(find=0x400602) # find a way to reach the address
found = sm.found[0]

# ask the symbolic solver the value of argv1 in the reached state as a string
solution = found.solver.eval(argv1, cast_to=bytes)
print(repr(solution))
```

```python
import angr, claripy
project = angr.Project("./ais3_crackme")

# create an initial state with a symbolic bit vector as argv1
argv1 = claripy.BVS("argv1",100*8) # 100 bytes
initial_state = project.factory.entry_state(args=["./ais3_crackme", argv1])

# create a path group using the created initial state
sm = project.factory.simulation_manager(initial_state)

# symbolically execute the program until we reach the wanted value of the IP
sm.explore(find=0x400602) # find a way to reach the address
found = sm.found[0]

# ask the symbolic solver the value of argv1 in the reached state as a string
solution = found.solver.eval(argv1, cast_to=bytes)
print(repr(solution))
```

```
$ python3 solve.py

ais3{I_tak3_g00d_n0t3s}\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00
```

Altre informazioni su angr

Cosa altro può fare angr?

- Symbolic Procedures
- Automatic ROP chain building
- Automatic binary hardening
- Automatic exploit generation (*per DECREE e binari Linux semplici*)

Installazione

angr si installa via pip

```
$ mkvirtualenv angr
$ pip install angr
```

C’è anche un container docker:

```
$ docker run -it angr/angr
```

Riferimenti

- angr: https://github.com/angr
- angr-doc: https://github.com/angr/angr-doc
- angr-course: https://github.com/angr/acsac-course
- z3: https://github.com/mwrlabs/z3_and_angr_binary_analysis_workshop
- https://www.slideshare.net/bananaappletw/triton-and-symbolic-execution-on-gdbdef-con-china-97054877


HACKLAB
BITM