

Analisi simbolica con angr

Enrico Bacis
- Hackers eat Pizza 2019 -



\$ whoami

Enrico Bacis

PhD student @ UniBG

Organizzazioni:

- Unibg Seclab
- BgLUG
- Hacklab BITM

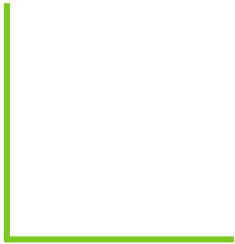
Ambiti di ricerca:

- 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

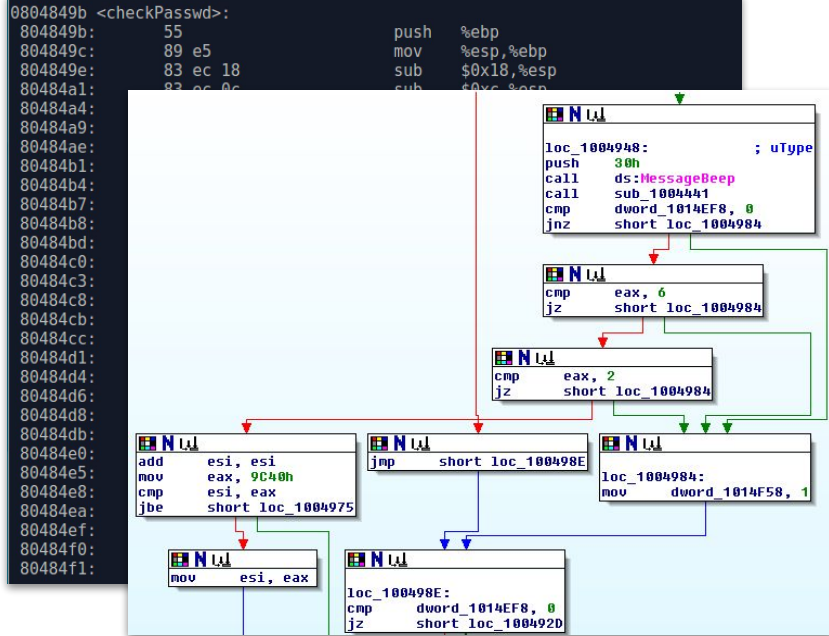


Static Analysis

```
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
```

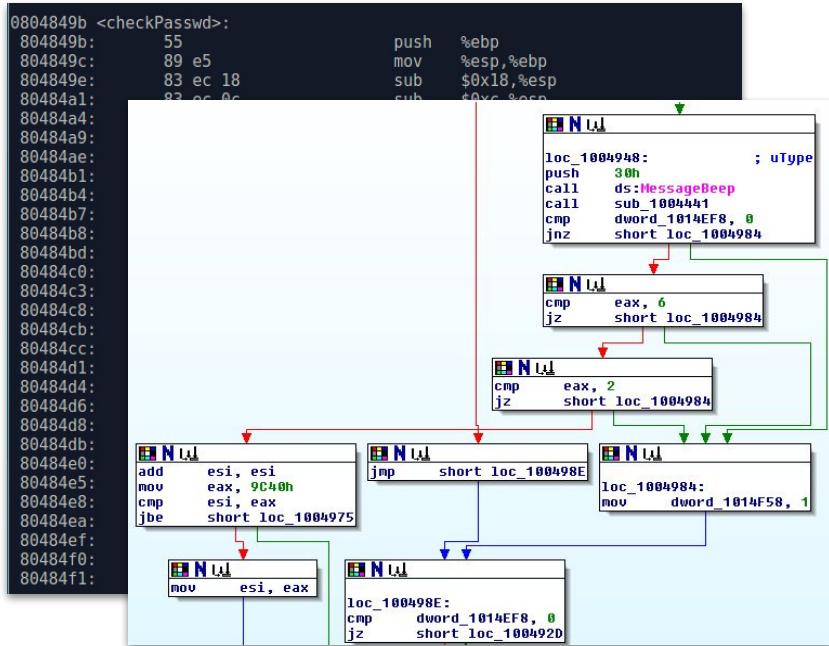
- objdump

Static Analysis



- objdump
- IDA

Static Analysis



- objdump
- IDA

Dynamic Analysis

```
gdb-peda$ start
[----- registers -----]
EAX: 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
EBX: 0xb7fcbff4 --> 0x155d7c
ECX: 0xd5eeaa03
EDX: 0x1
ESI: 0x0
EDI: 0x0
EBP: 0xbffff748 --> 0xbffff7c8 --> 0x0
ESP: 0xbffff748 --> 0xbffff7c8 --> 0x0
EIP: 0x80483e7 (<main+3>: and esp,0xffffffff)
EFLAGS: 0x200246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[----- code -----]
0x80483e3 <frame dummy+35>: nop
0x80483e4 <main>: push ebp
0x80483e5 <main+1>: mov ebp,esp
=> 0x80483e7 <main+3>: and esp,0xffffffff
0x80483ea <main+6>: sub esp,0x110
0x80483f0 <main+12>: mov eax,DWORD PTR [ebp+0xc]
0x80483f3 <main+15>: add eax,0x4
0x80483f6 <main+18>: mov eax,DWORD PTR [eax]
[----- stack -----]
0000| 0xbffff748 --> 0xbffff7c8 --> 0x0
0004| 0xbffff74c --> 0xb7e8cbd6 (<_libc_start_main+230>: mov DWORD PTR [e
0008| 0xbffff750 --> 0x1
0012| 0xbffff754 --> 0xbffff7f4 --> 0xbffff916 ("/root/a.out")
0016| 0xbffff758 --> 0xbffff7fc --> 0xbffff922 ("SHELL=/bin/bash")
0020| 0xbffff75c --> 0xb7fe1858 --> 0xb7e76000 --> 0x464c457f
0024| 0xbffff760 --> 0xbffff7b0 --> 0x0
0028| 0xbffff764 --> 0xffffffff
Legend: code, data, rodata, value
Temporary breakpoint 1, 0x80483e7 in main ()
gdb-peda$
```

- gdb (& friends)

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:
80484a9:
80484ae:
80484b1:
80484b4:
80484b7:
80484b8:
80484bd:
80484c0:
80484c3:
80484c8:
80484cb:
80484cc:
80484d1:
80484d4:
80484d6:
80484db:
80484de:
80484e0:
80484e8:
80484ea:
80484ef:
80484f0:
80484f1:
```

```
loc_1004948:                ; uType
push    30h
call    ds:MessageBeep
call    sub_1004441
cmp     dword_1014EF8, 0
jnz     short loc_1004984

cmp     eax, 6
jz      short loc_1004984

cmp     eax, 2
jz      short loc_1004984

add     esi, esi
mov     eax, 9C40h
cmp     esi, eax
jbe     short loc_1004975

jmp     short loc_100498E

loc_1004984:
mov     dword_1014F58, 1

loc_100498E:
cmp     dword_1014EF8, 0
jz      short loc_100492D
```

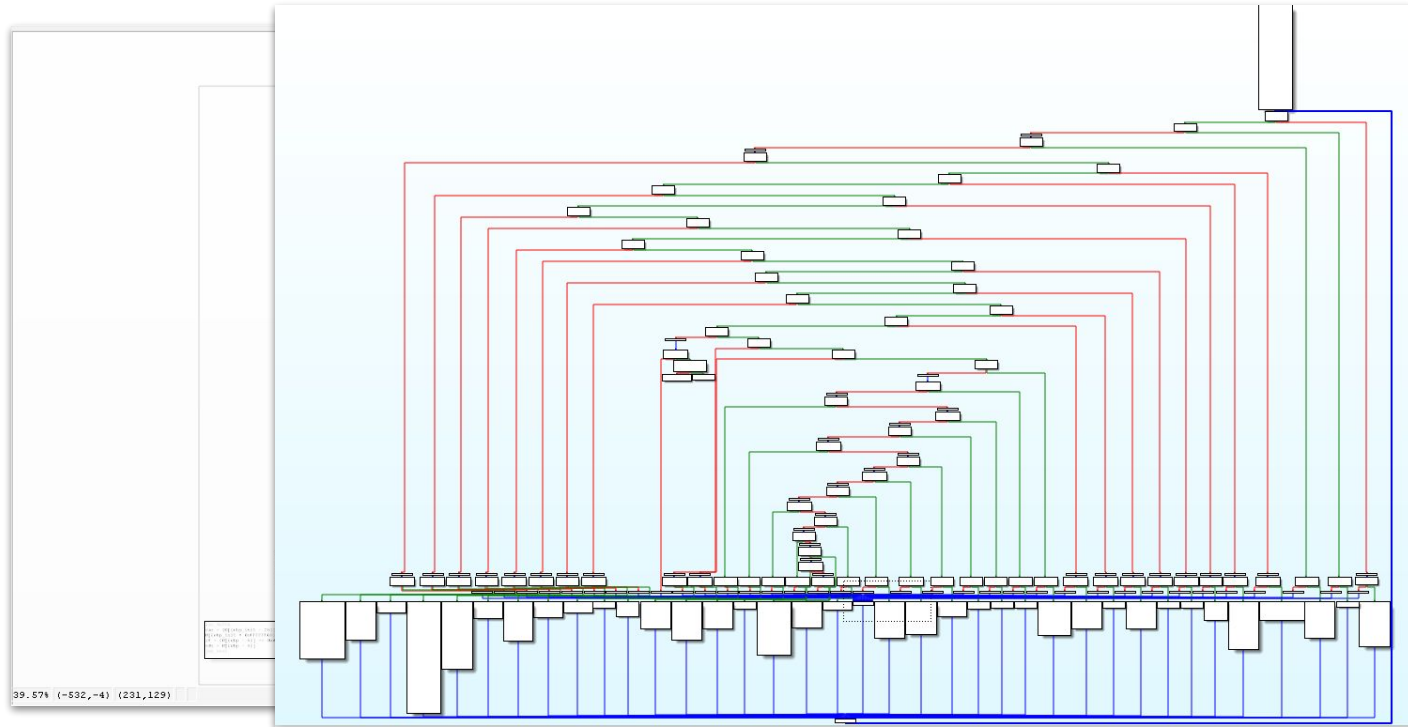
- objdump
- IDA

Dynamic Analysis

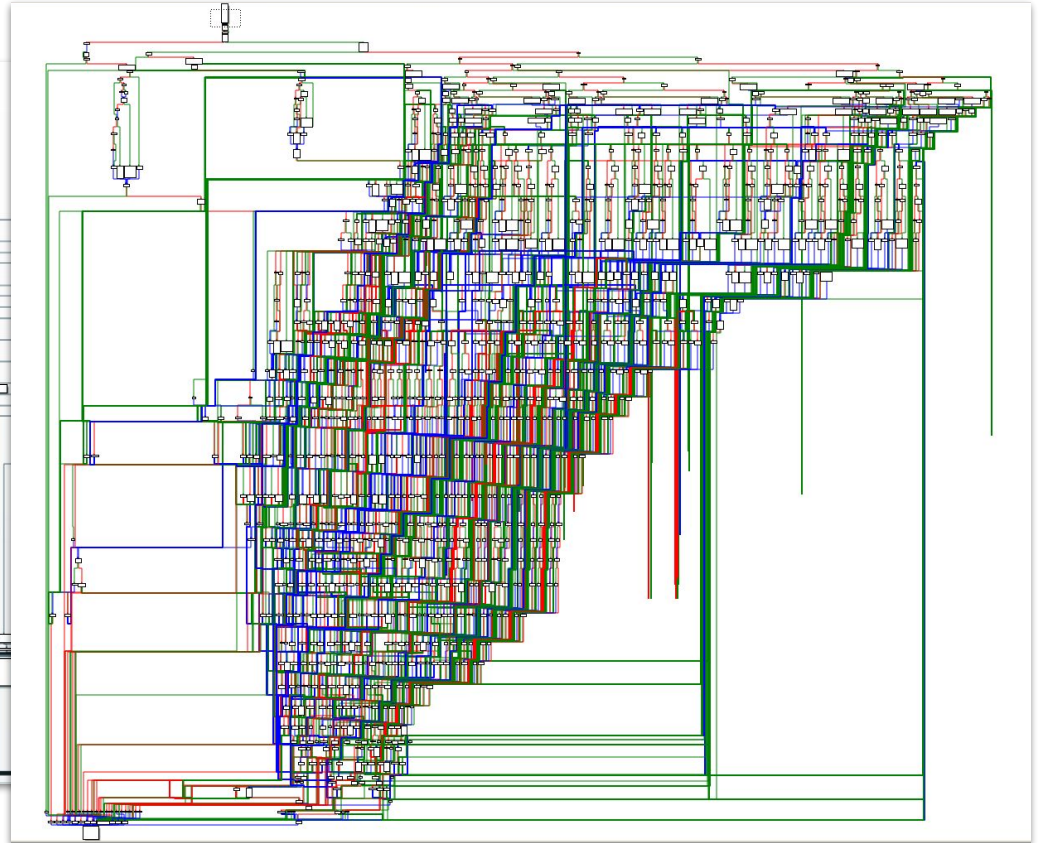
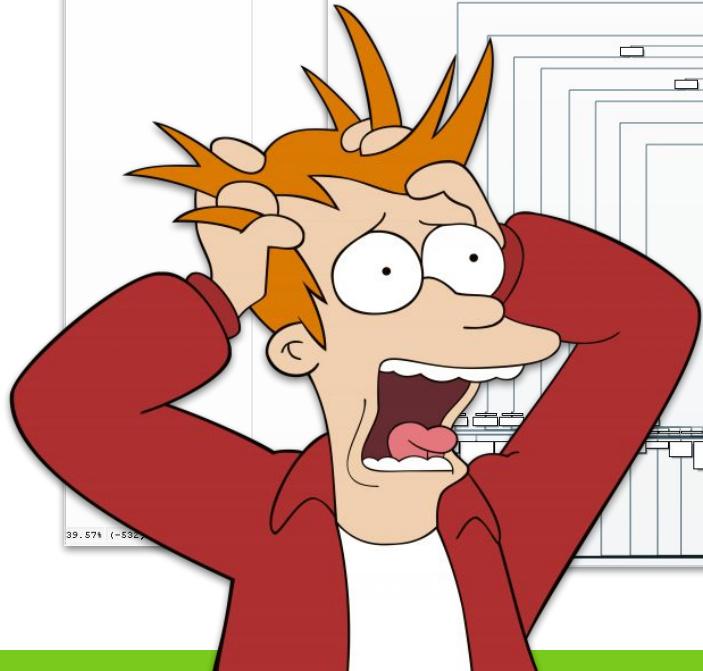
```
gdb-peda$ start
[----- registers -----]
EAX: 0xbffff7e4 --> 0xbffff916 ("/root/a.out")
EBX: 0xb7fcbff4 --> 0x155d7c
ECX: 0xd5eaa03
EDX: 0x1
ESI: [0x08048471 185 /root/IOLI-crackme/crackme0x03]> ?0;f tmp;s.. @ sym.test+3
EDI: - offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
EBP: 0xbfd97790 ec85 0408 1819 f4b7 c877 d9bf 1185 0408 .....W.....
ESP: 0xbfd977a0 1000 0000 242b 0500 0000 0000 bb84 d5b7 ...$.+.....
EIP: 0xbfd977b0 dc33 eeb7 f881 0408 0c9f 0408 242b 0500 .3.....$.+..
EFLA: 0xbfd977c0 4602 0000 1000 0000 0000 0000 5614 d4b7 F.....V...
0: eax 0x00000010 ebx 0x00000000 ecx 0x00000000 edx 0x000001ec
0: esi 0x00000001 edi 0xb7ee3000 esp 0xbfd97790 ebp 0xbfd97798
=> 0: eip 0x08048483 eflags C1ASI oeax 0xffffffff
0: 0x08048471 83ec08 sub esp, 8
0: 0x08048474 8b4508 mov eax, dword [arg_8h]
0: 0x08048477 3b450c cmp eax, dword [arg_ch]
0: 0x0804847a 740e je 0x0804848a
0: 0x0804847c c70424ec8504. mov dword [esp], str.LqydoLg_Sdvvz
[-----]
0000 ;-- eip:
0004 0x08048483 e88cffffff call sym.shift
0008 0x08048488 eb0c jmp 0x08048496
0012 0x0804848a c70424fe8504. mov dword [esp], str.Sdvvzrug_RN_
0016 0x08048491 e87effffff call sym.shift
0020 ; JMP XREF from 0x08048488 (sym.test)
0024 0x08048496 c9 leave
0028 0x08048497 c3 ret
[-----]
Temp\
gdb-] ;-- main:
(fcn) sym.main 128
sym.main ();
```

- gdb (& friends)
- radare2

Limiti



Limiti





Come funziona l'analisi simbolica

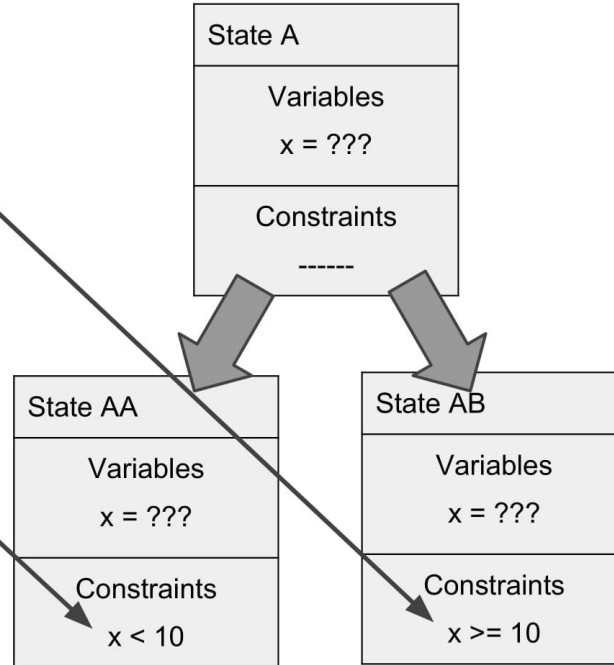
(a.k.a. quattro slide rubate da qualsiasi presentazione su angr mai fatta)



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

State A
Variables x = ???
Constraints -----

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

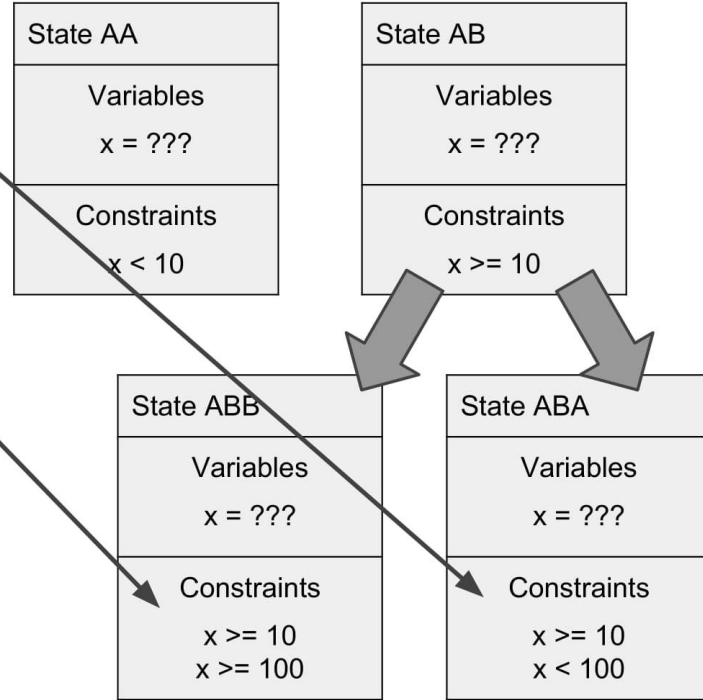


```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```

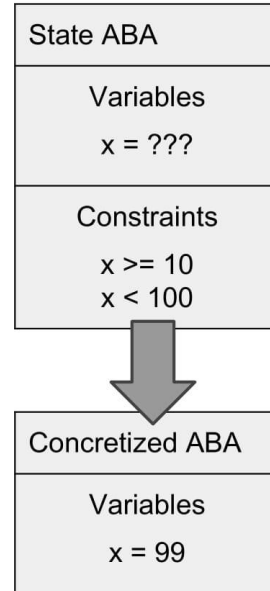
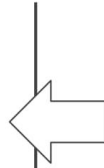
State AA
Variables x = ???
Constraints x < 10

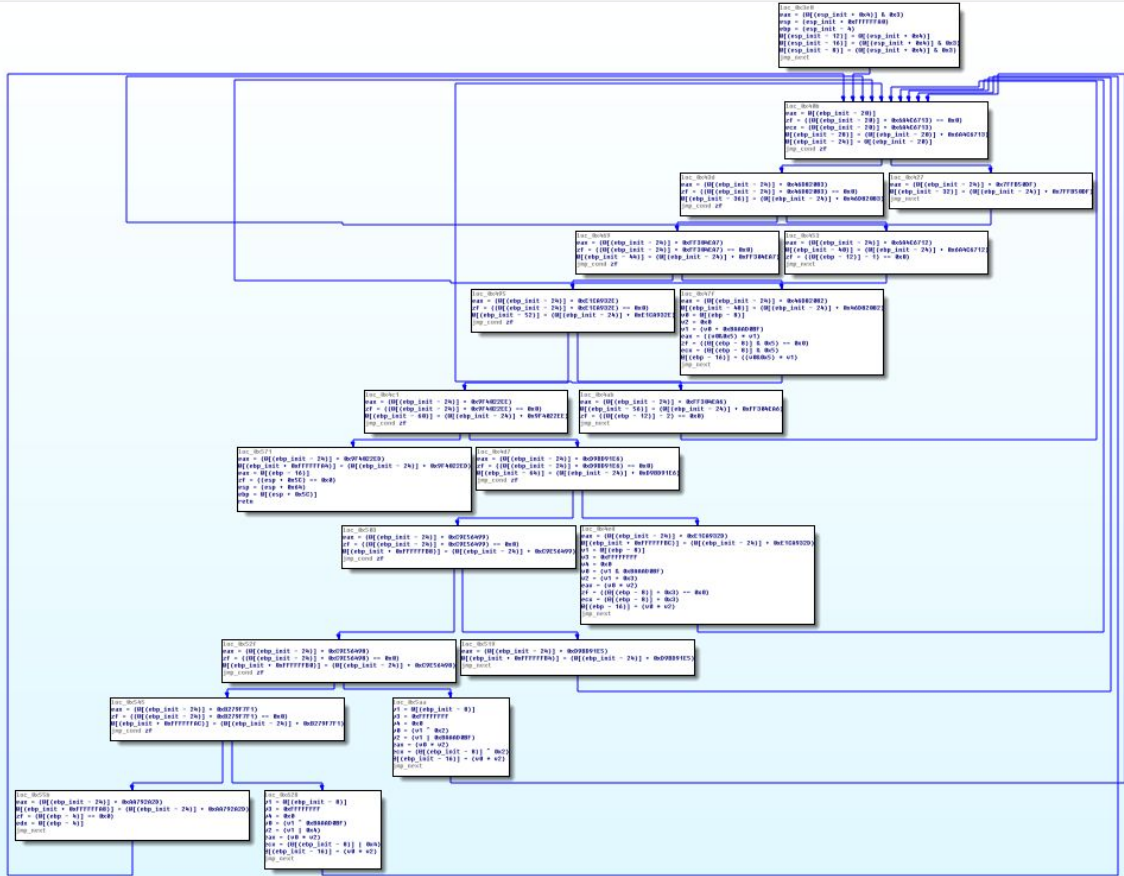
State AB
Variables x = ???
Constraints x >= 10

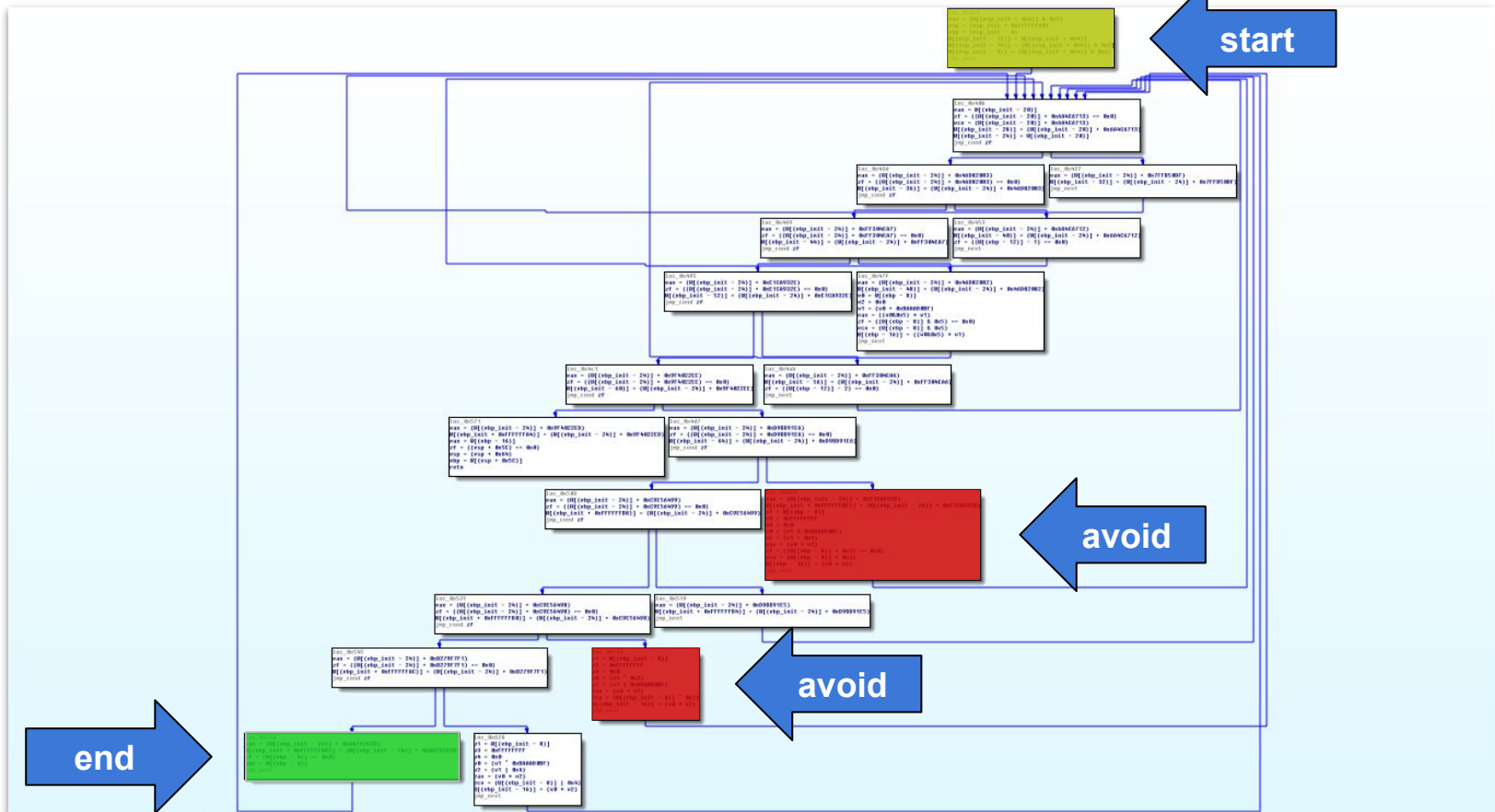
```
x = int(input())
if x >= 10:
    if x < 100:
        print "You win!"
    else:
        print "You lose!"
else:
    print "You lose!"
```




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







start

avoid

avoid

end



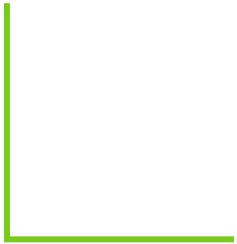
angr

<https://angr.io>

Cosa è angr

- Framework per binary analysis in python che combina analisi statica e dinamica simbolica ("*concolic analysis*" da **con**crete e symbol**ic**)
- 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 *)
0x004005e4      b8ffffff    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 *)
; <=
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 *)
; 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_sec
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))
```

```
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))
```




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>

