Triton and Symbolic execution on GDB

bananaappletw @ HITCON

2017/08/26

$whoami

- 陳威伯(bananaappletw)
- Master of National Chiao Tung University
- Organizations:
  - Software Quality Laboratory
  - Bamboofox member
  - Vice president of NCTUCSC
- Specialize in:
  - symbolic execution
  - binary exploit
- Talks:
  - HITCON CMT 2015

Outline

- Why symbolic execution?
- Symbolic execution?
- Triton
- SymGDB

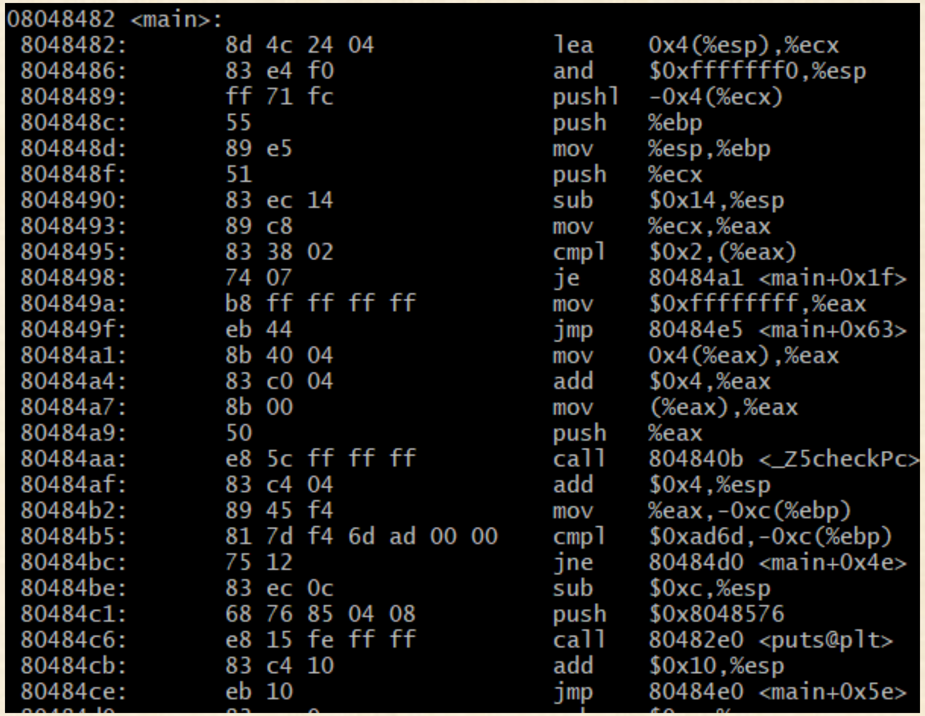
Why symbolic execution?

In the old days

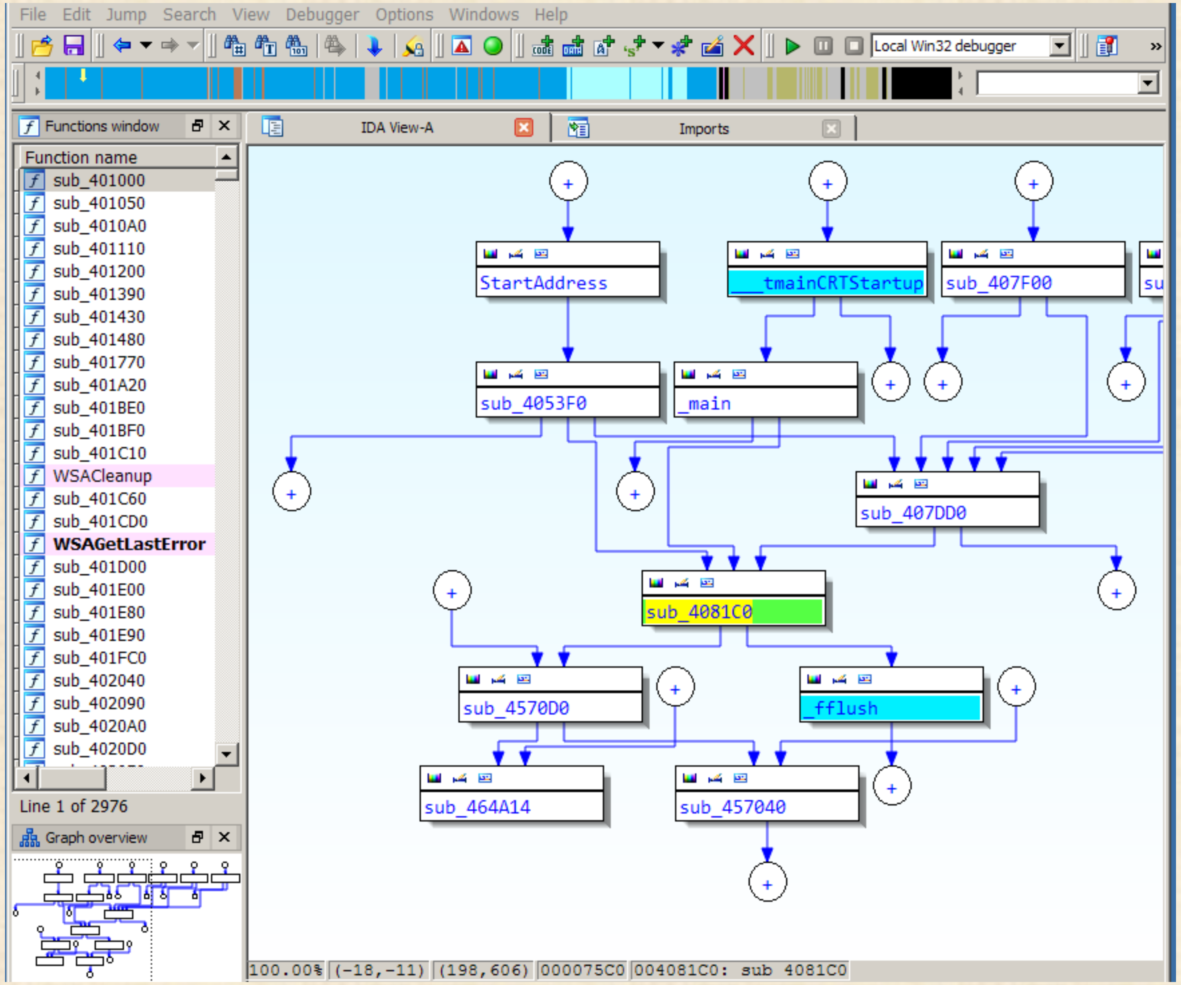
- Static analysis
- Dynamic analysis

Static analysis

- objdump
- IDA PRO



```
08048482 <main>:
 8048482:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048486:       83 e4 f0                and    $0xfffffff0,%esp
 8048489:       ff 71 fc                pushl  -0x4(%ecx)
 804848c:       55                      push   %ebp
 804848d:       89 e5                   mov    %esp,%ebp
 804848f:       51                      push   %ecx
 8048490:       83 ec 14                sub    $0x14,%esp
 8048493:       89 c8                   mov    %ecx,%eax
 8048495:       83 38 02                cmpl   $0x2,(%eax)
 8048498:       74 07                   je     80484a1 <main+0x1f>
 804849a:       b8 ff ff ff ff          mov    $0xffffffff,%eax
 804849f:       eb 44                   jmp    80484e5 <main+0x63>
 80484a1:       8b 40 04                mov    0x4(%eax),%eax
 80484a4:       83 c0 04                add    $0x4,%eax
 80484a7:       8b 00                   mov    (%eax),%eax
 80484a9:       50                      push   %eax
 80484aa:       e8 5c ff ff ff          call   804840b <_Z5checkPc>
 80484af:       83 c4 04                add    $0x4,%esp
 80484b2:       89 45 f4                mov    %eax,-0xc(%ebp)
 80484b5:       81 7d f4 6d ad 00 00    cmpl   $0xad6d,-0xc(%ebp)
 80484bc:       75 12                   jne    80484d0 <main+0x4e>
 80484be:       83 ec 0c                sub    $0xc,%esp
 80484c1:       68 76 85 04 08          push   $0x8048576
 80484c6:       e8 15 fe ff ff          call   80482e0 <puts@plt>
 80484cb:       83 c4 10                add    $0x10,%esp
 80484ce:       eb 10                   jmp    80484e0 <main+0x5e>
```



Function name

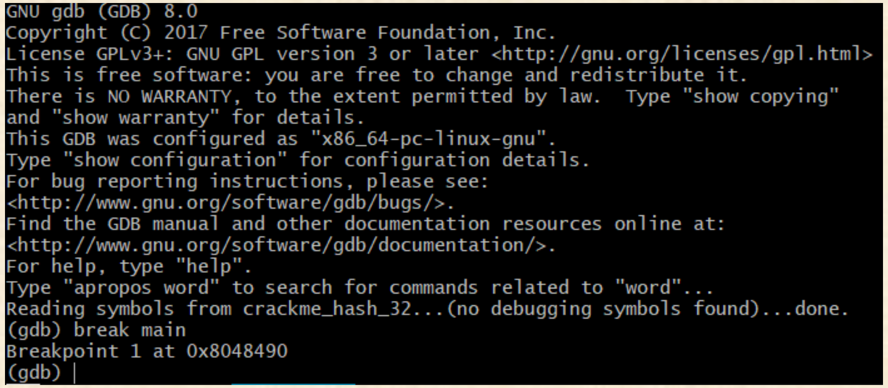
sub_401000
sub_401050
sub_4010A0
sub_401110
sub_401200
sub_401390
sub_401430
sub_401480
sub_401770
sub_401A20
sub_401BE0
sub_401BF0
sub_401C10
WSACleanup
sub_401C60
sub_401CD0
WSAGetLastError
sub_401D00
sub_401E00
sub_401E80
sub_401E90
sub_401FC0
sub_402040
sub_402090
sub_4020A0
sub_4020D0

Line 1 of 2976

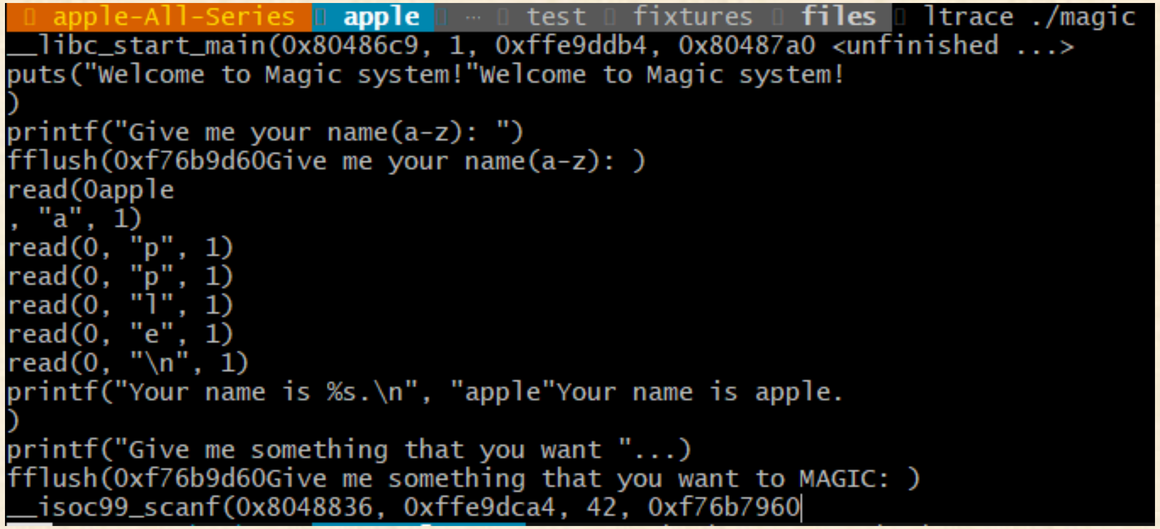
100.00% (-18,-11) (198,606) 000075C0 004081C0: sub 4081C0

Dynamic analysis

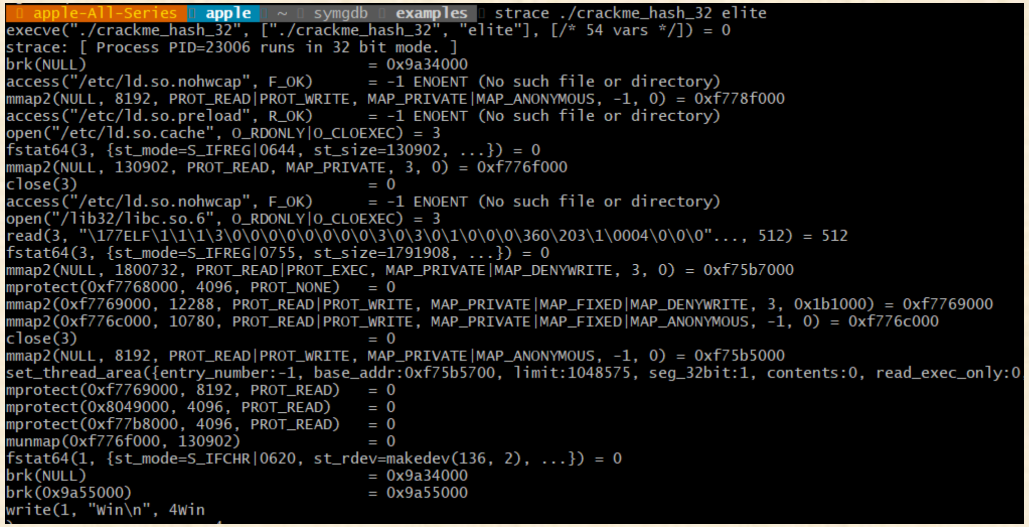
- GDB
- ltrace
- strace



```
GNU gdb (GDB) 8.0
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from crackme_hash_32...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x8048490
(gdb) |
```

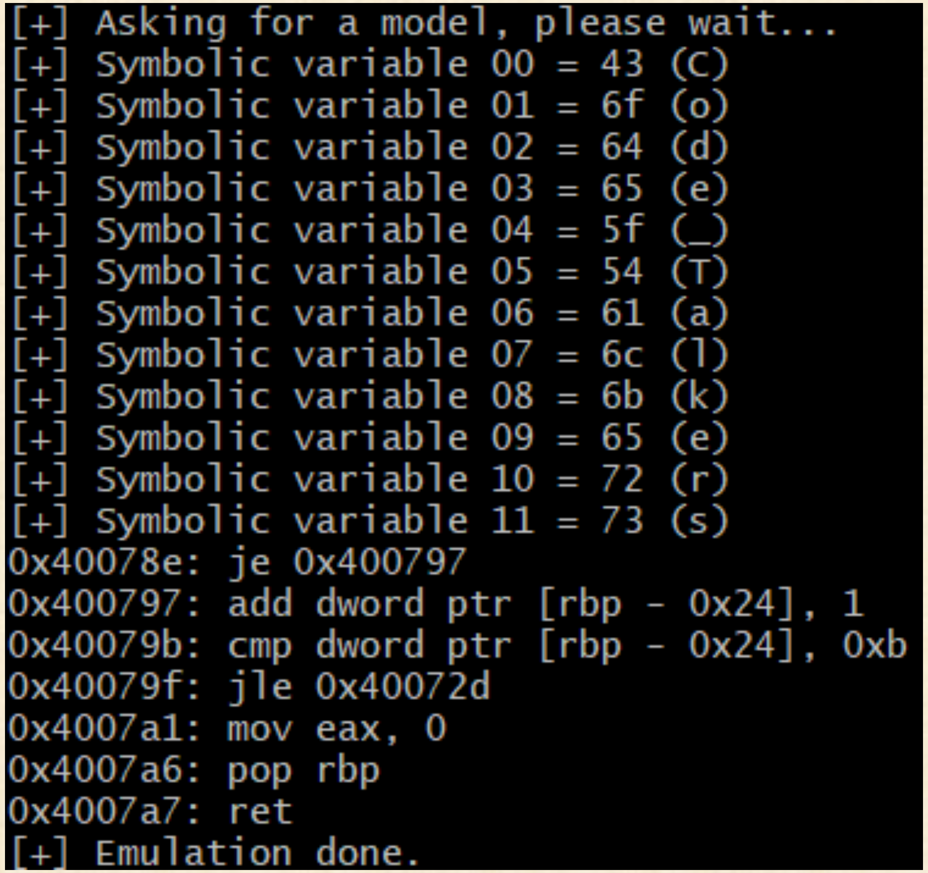


```
 apple-All-Series  apple  …  test  fixtures  files  ltrace ./magic
__libc_start_main(0x80486c9, 1, 0xffe9ddb4, 0x80487a0 <unfinished ...>
puts("Welcome to Magic system!"Welcome to Magic system!
)
printf("Give me your name(a-z): ")
fflush(0xf76b9d60Give me your name(a-z): )
read(0apple
, "a", 1)
read(0, "p", 1)
read(0, "p", 1)
read(0, "l", 1)
read(0, "e", 1)
read(0, "\n", 1)
printf("Your name is %s.\n", "apple"Your name is apple.
)
printf("Give me something that you want "...)
fflush(0xf76b9d60Give me something that you want to MAGIC: )
__isoc99_scanf(0x8048836, 0xffe9dca4, 42, 0xf76b7960|
```



```
 apple-All-Series  apple  ~  symgdb  examples  strace ./crackme_hash_32 elite
execve("./crackme_hash_32", ["./crackme_hash_32", "elite"], [/* 54 vars */]) = 0
strace: [ Process PID=23006 runs in 32 bit mode. ]
brk(NULL)                               = 0x9a34000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf778f000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=130902, ...}) = 0
mmap2(NULL, 130902, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf776f000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\360\203\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1791908, ...}) = 0
mmap2(NULL, 1800732, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf75b7000
mprotect(0xf7768000, 4096, PROT_NONE)   = 0
mmap2(0xf7769000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b1000) = 0xf7769000
mmap2(0xf776c000, 10780, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf776c000
close(3)                                = 0
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf75b5000
set_thread_area({entry_number:-1, base_addr:0xf75b5700, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0
mprotect(0xf7769000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xf77b8000, 4096, PROT_READ)   = 0
munmap(0xf776f000, 130902)              = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 2), ...}) = 0
brk(NULL)                               = 0x9a34000
brk(0x9a55000)                          = 0x9a55000
write(1, "Win\n", 4Win
```

Symbolic execution!!!

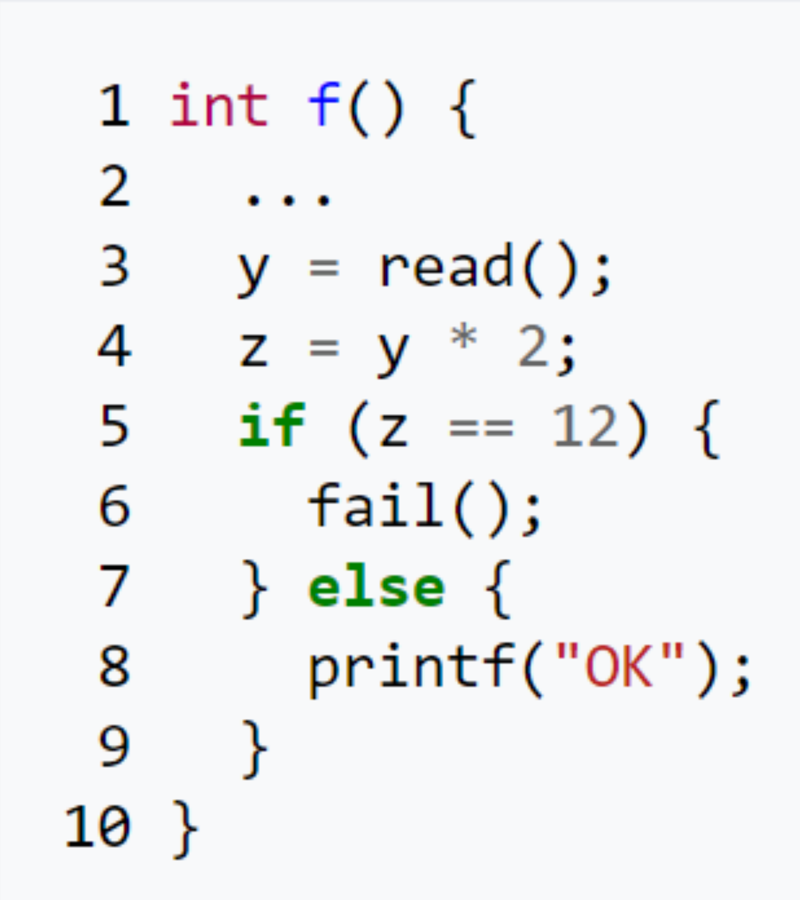


```
[+] Asking for a model, please wait...
[+] Symbolic variable 00 = 43 (C)
[+] Symbolic variable 01 = 6f (o)
[+] Symbolic variable 02 = 64 (d)
[+] Symbolic variable 03 = 65 (e)
[+] Symbolic variable 04 = 5f (_)
[+] Symbolic variable 05 = 54 (T)
[+] Symbolic variable 06 = 61 (a)
[+] Symbolic variable 07 = 6c (l)
[+] Symbolic variable 08 = 6b (k)
[+] Symbolic variable 09 = 65 (e)
[+] Symbolic variable 10 = 72 (r)
[+] Symbolic variable 11 = 73 (s)
0x40078e: je 0x400797
0x400797: add dword ptr [rbp - 0x24], 1
0x40079b: cmp dword ptr [rbp - 0x24], 0xb
0x40079f: jle 0x40072d
0x4007a1: mov eax, 0
0x4007a6: pop rbp
0x4007a7: ret
[+] Emulation done.
```

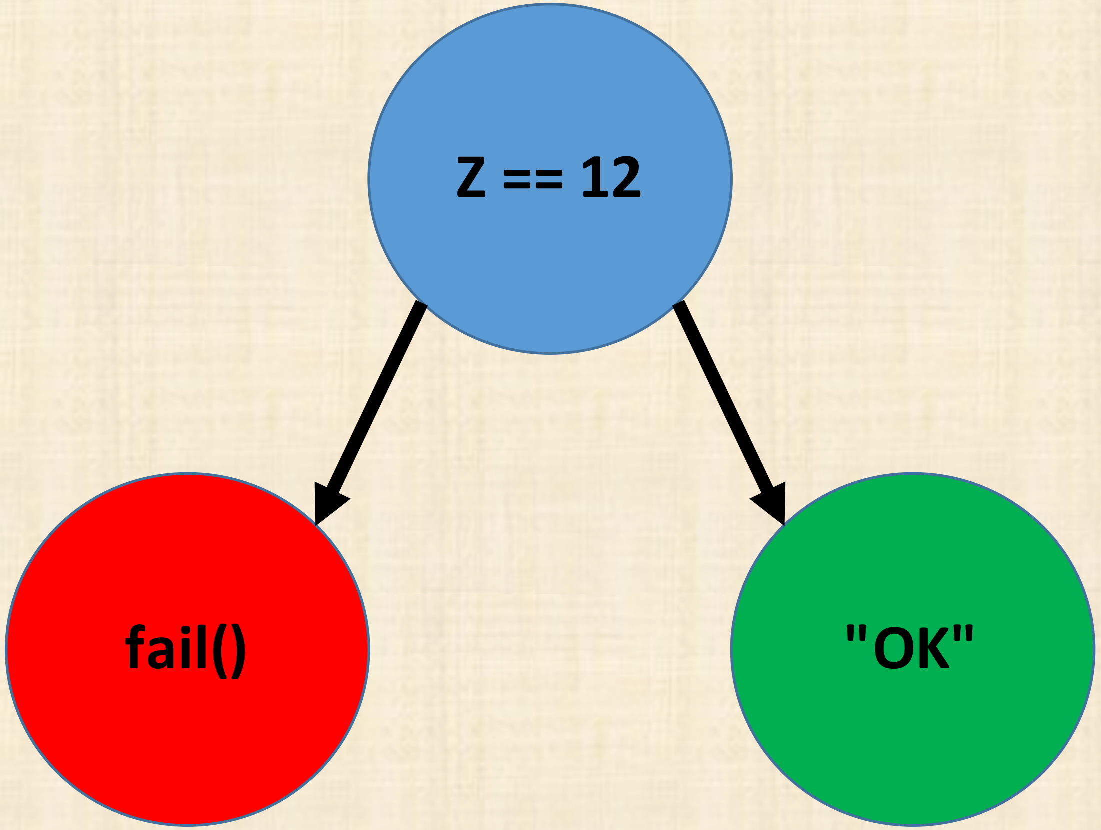
What is symbolic execution?

- Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute
- System-level
    - S2e(https://github.com/dslab-epfl/s2e)
- User-level
    - Angr(http://angr.io/)
    - Triton(https://triton.quarkslab.com/)
- Code-based
    - klee(http://klee.github.io/)

Symbolic execution



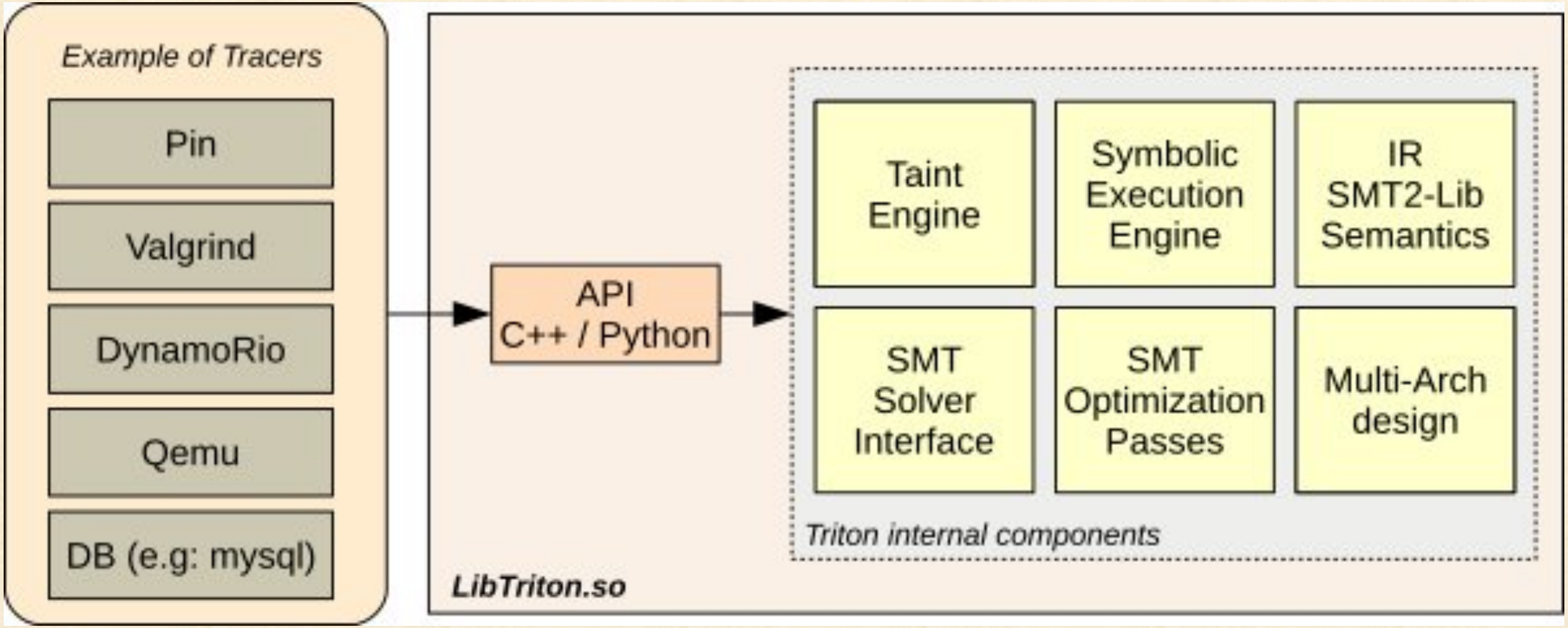
```
int f() {
  ...
  y = read();
  z = y * 2;
  if (z == 12) {
    fail();
  } else {
    printf("OK");
  }
}
```



Triton

- Website: https://triton.quarkslab.com/
- A dynamic binary analysis framework written in C++.
  - developed by Jonathan Salwan
- Python bindings
- Triton components:
  - Symbolic execution engine
  - Tracer
  - AST representations
  - SMT solver Interface

Triton Structure



Symbolic execution engine

- The symbolic engine maintains:
  - a table of symbolic registers states
  - a map of symbolic memory states
  - a global set of all symbolic references

Step	Register	Instruction	Set of symbolic expressions
init	eax = UNSET	None	⊥
1	eax = ϕ1	mov eax, 0	{ϕ1=0}
2	eax = ϕ2	inc eax	{ϕ1=0,ϕ2=ϕ1+1}
3	eax = ϕ3	add eax, 5	{ϕ1=0,ϕ2=ϕ1+1,ϕ3=ϕ2+5}

Triton Tracer

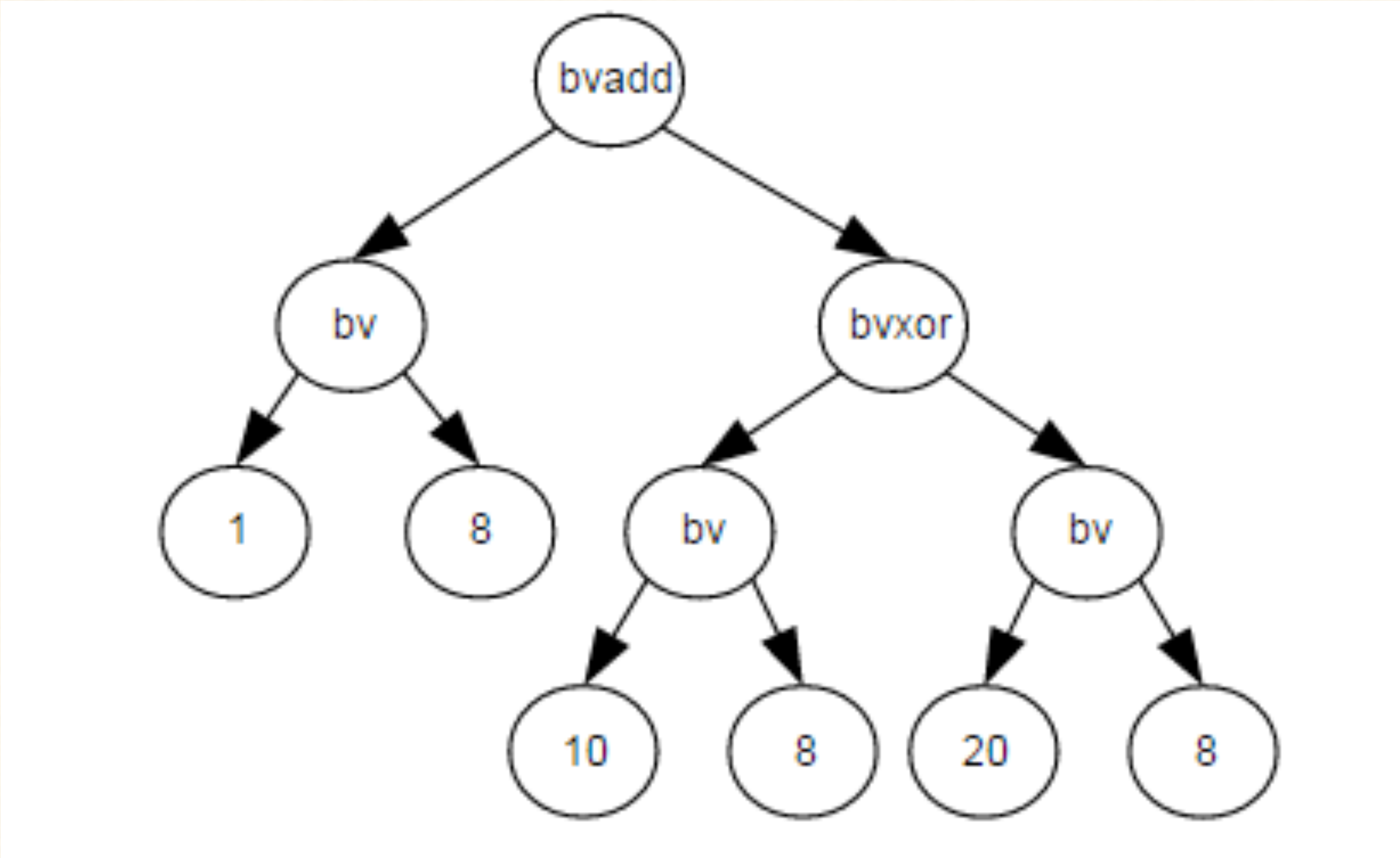
- Tracer provides:
  - Current opcode executed
  - State context (register and memory)
- Translate the control flow into **AST Representations**
- Pin tracer support

AST representations

- Triton converts the x86 and the x86-64 instruction set semantics into AST representations
- Triton's expressions are on **SSA form**
- Instruction: add rax, rdx
- Expression: **ref!41** = (bvadd ((_ extract 63 0) **ref!40**) ((_ extract 63 0) **ref!39**))
- **ref!41** is the new expression of the RAX register
- **ref!40** is the previous expression of the RAX register
- **ref!39** is the previous expression of the RDX register

AST representations

- mov al, 1
- mov cl, 10
- mov dl, 20
- xor cl, dl
- add al, cl



Static single assignment form(SSA form)

- Each variable is assigned exactly **once**
- y := 1
- y := 2
- x := y

Turns into

- y1 := 1
- y2 := 2
- x1 := y2

Why SSA form?

~~y1 := 1~~ (This assignment is not necessary)

y2 := 2

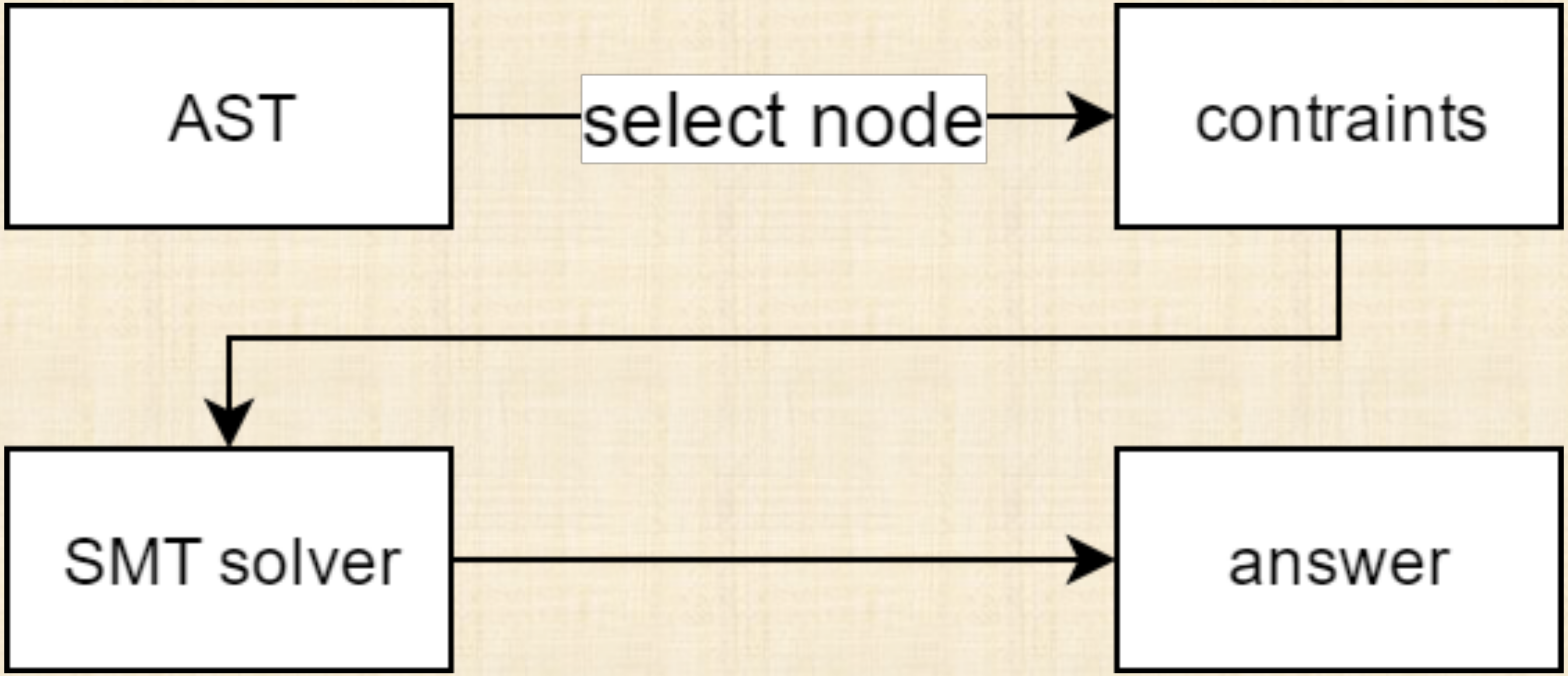
x1 := y2

- When Triton process instructions, it could ignore some unnecessary instructions.
- It saves **time** and **memory**.

Symbolic variables

- Imagine symbolic is a infection
- Make ecx as symbolic variable
- convertRegisterToSymbolicVariable(REG.ECX)
- isRegisterSymbolized(REG.ECX) == True
- test ecx, ecx (ZF = ECX & ECX = ECX)
- je +7 (isRegisterSymbolized(REG.EIP) == True)(jump to nop if ZF=1)
- mov edx, 0x64
- nop

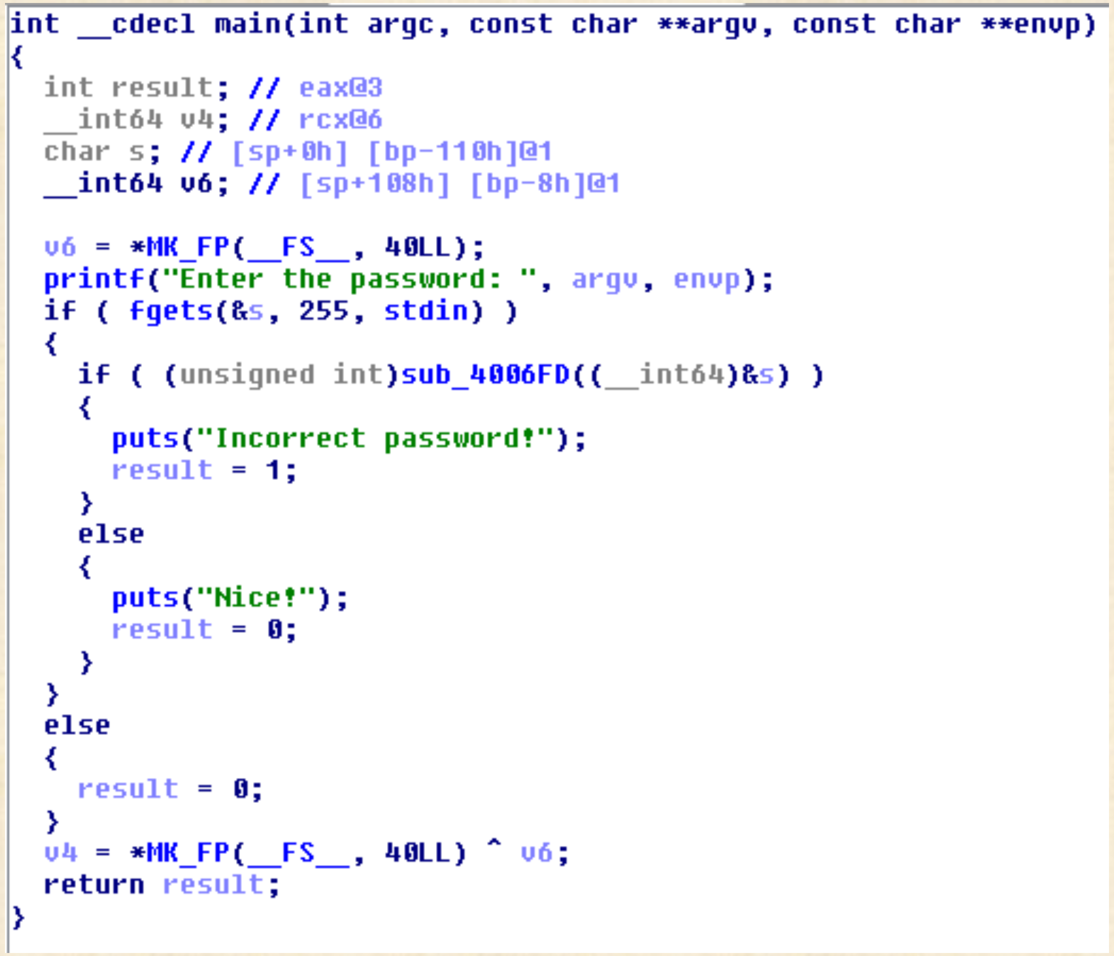
SMT solver Interface



Example

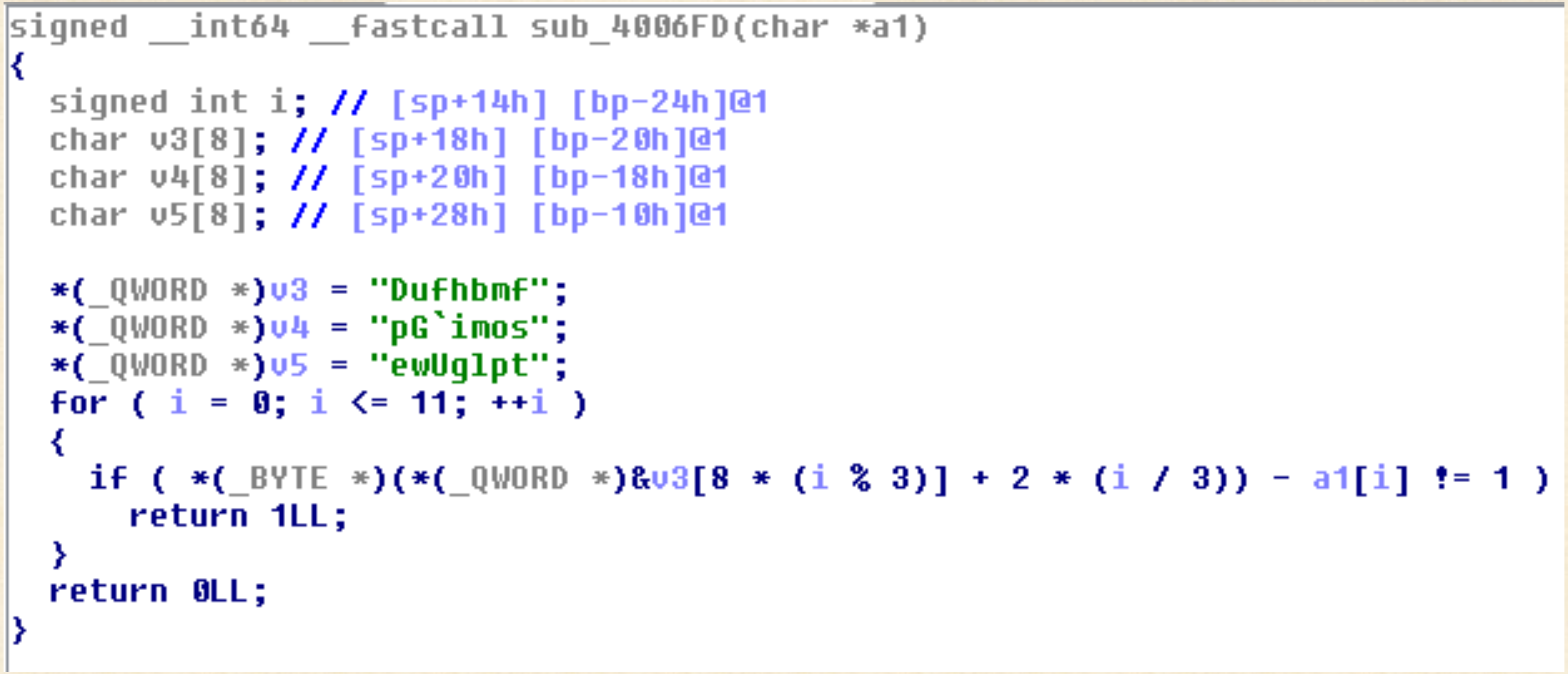
- Defcamp 2015 r100
- Program require to input the password
- Password length could up to 255 characters

Defcamp 2015 r100



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int result; // eax@3
  __int64 v4; // rcx@6
  char s; // [sp+0h] [bp-110h]@1
  __int64 v6; // [sp+108h] [bp-8h]@1

  v6 = *MK_FP(__FS__, 40LL);
  printf("Enter the password: ", argv, envp);
  if ( fgets(&s, 255, stdin) )
  {
    if ( (unsigned int)sub_4006FD((__int64)&s) )
    {
      puts("Incorrect password!");
      result = 1;
    }
    else
    {
      puts("Nice!");
      result = 0;
    }
  }
  else
  {
    result = 0;
  }
  v4 = *MK_FP(__FS__, 40LL) ^ v6;
  return result;
}
```

Defcamp 2015 r100

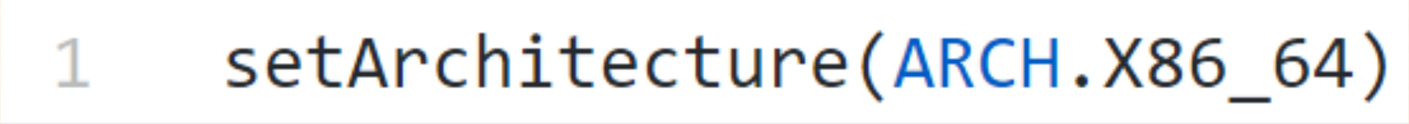


```
signed __int64 __fastcall sub_4006FD(char *a1)
{
  signed int i; // [sp+14h] [bp-24h]@1
  char v3[8]; // [sp+18h] [bp-20h]@1
  char v4[8]; // [sp+20h] [bp-18h]@1
  char v5[8]; // [sp+28h] [bp-10h]@1

  *(_QWORD *)v3 = "Dufhbmf";
  *(_QWORD *)v4 = "pG`imos";
  *(_QWORD *)v5 = "ewUglpt";
  for ( i = 0; i <= 11; ++i )
  {
    if ( *(_BYTE *)(*(_QWORD *)&v3[8 * (i % 3)] + 2 * (i / 3)) - a1[i] != 1 )
      return 1LL;
  }
  return 0LL;
}
```

Defcamp 2015 r100

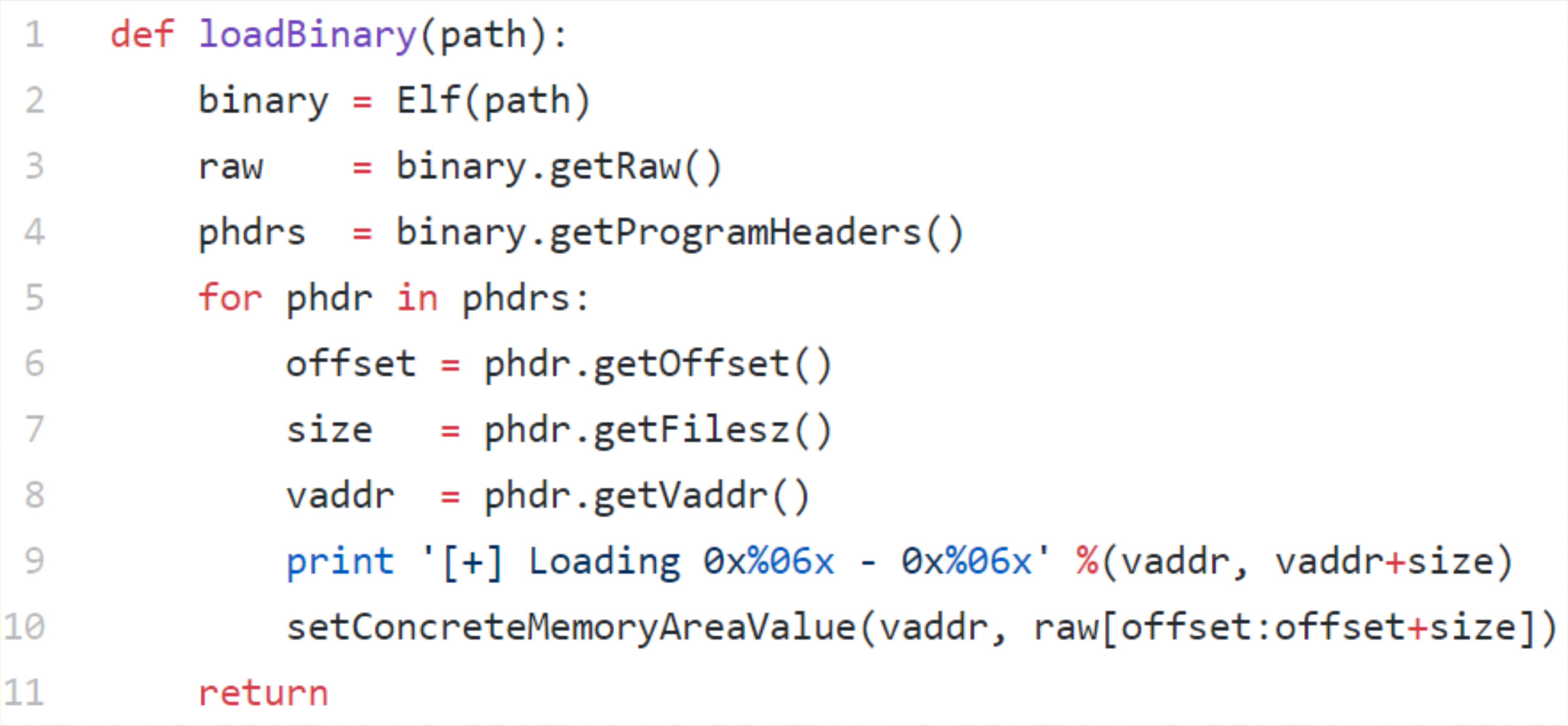
- Set Architecture
- Load segments into triton
- Define fake stack (RBP and RSP)
- Symbolize user input
- Start to processing opcodes
- Set constraint on specific point of program
- Get symbolic expression and solve it

Set Architecture



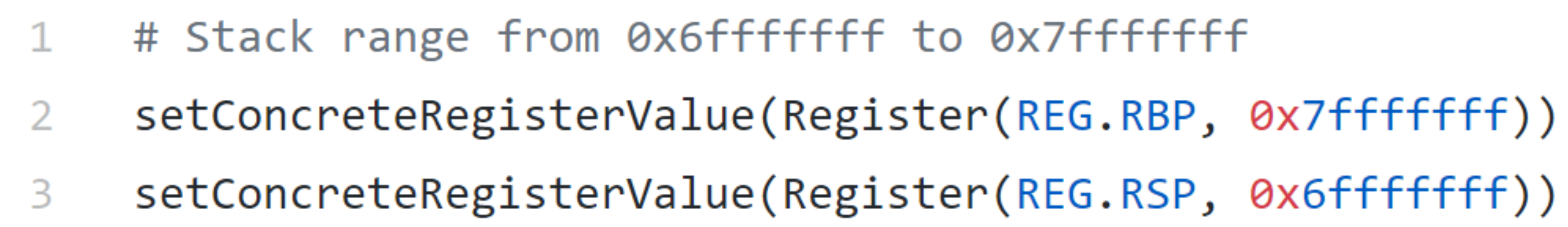
```
setArchitecture(ARCH.X86_64)
```

Load segments into triton



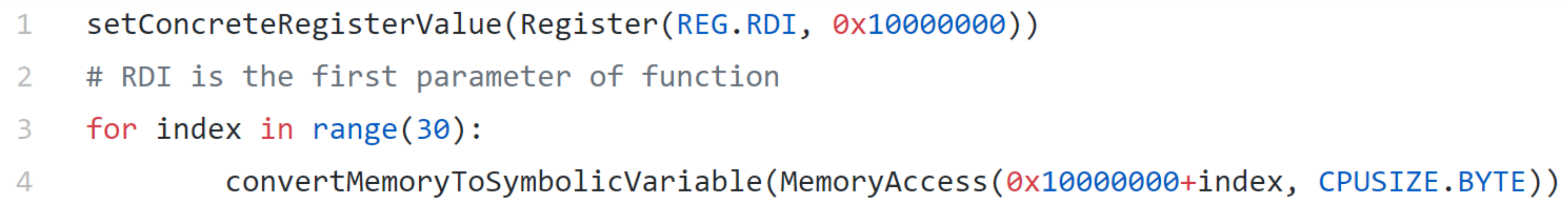
```python
def loadBinary(path):
    binary = Elf(path)
    raw    = binary.getRaw()
    phdrs  = binary.getProgramHeaders()
    for phdr in phdrs:
        offset = phdr.getOffset()
        size   = phdr.getFilesz()
        vaddr  = phdr.getVaddr()
        print '[+] Loading 0x%06x - 0x%06x' %(vaddr, vaddr+size)
        setConcreteMemoryAreaValue(vaddr, raw[offset:offset+size])
    return
```

Define fake stack (RBP and RSP)



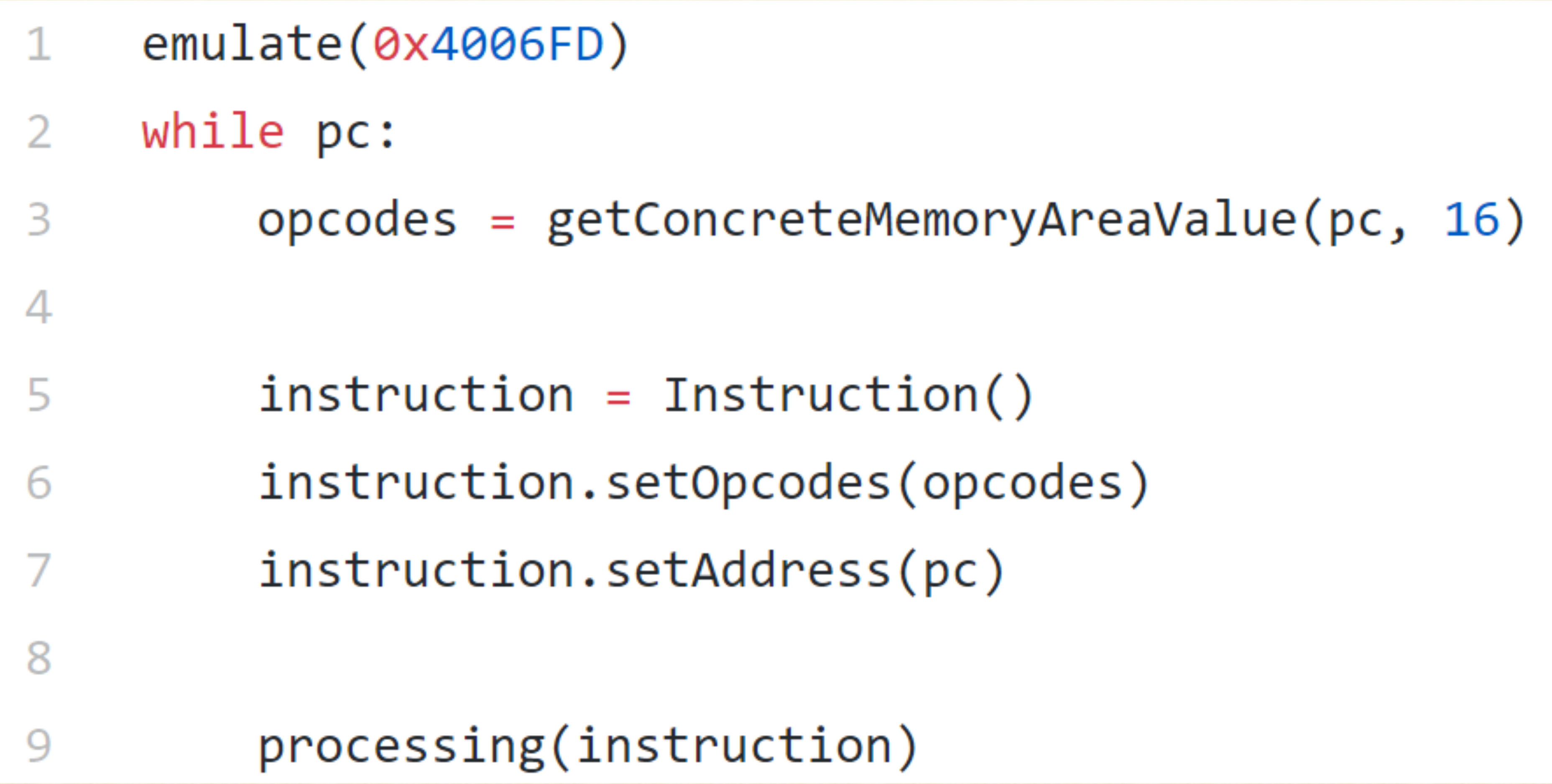
```
# Stack range from 0x6fffffff to 0x7fffffff
setConcreteRegisterValue(Register(REG.RBP, 0x7fffffff))
setConcreteRegisterValue(Register(REG.RSP, 0x6fffffff))
```

Symbolize user input



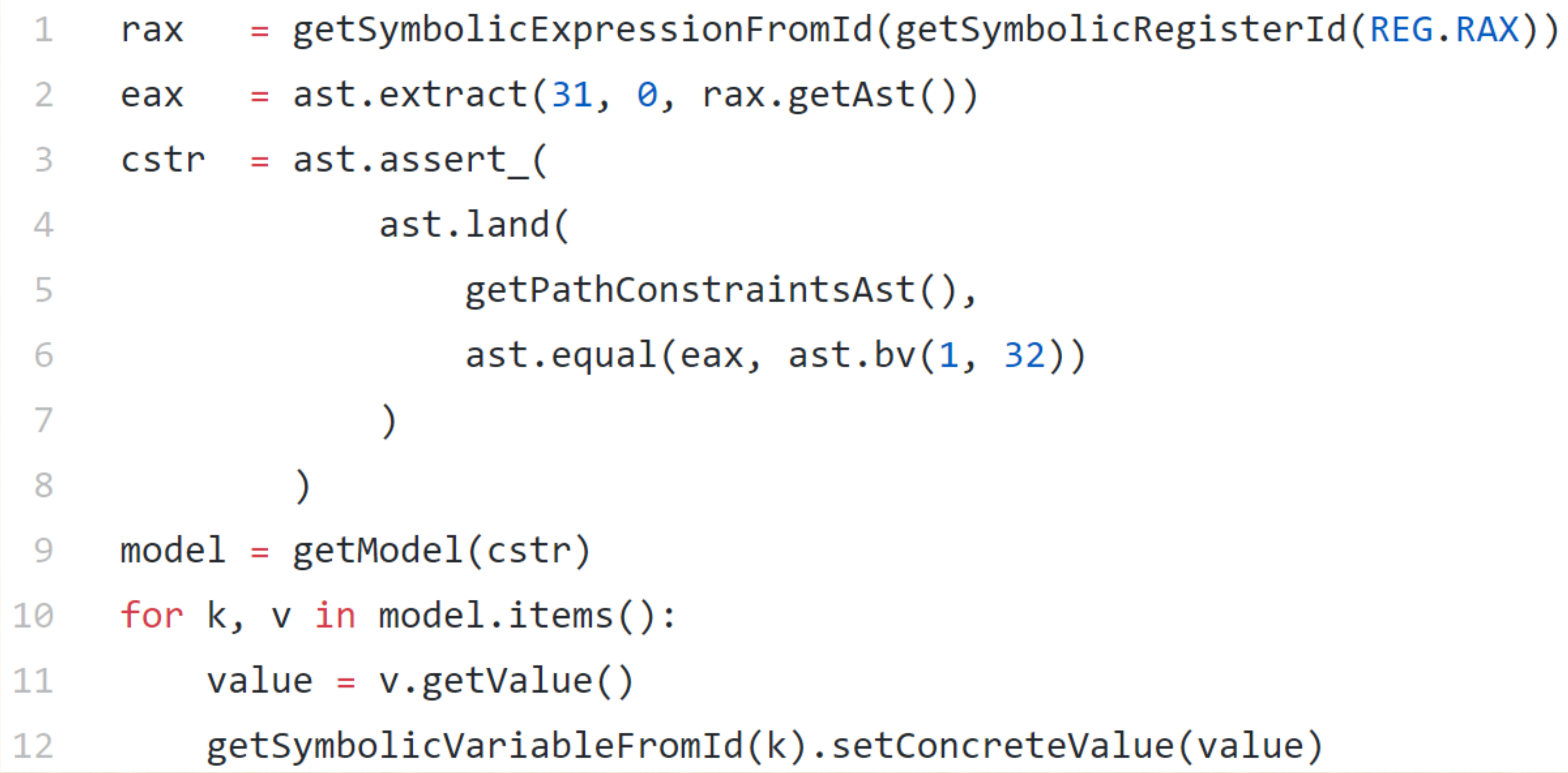
```
setConcreteRegisterValue(Register(REG.RDI, 0x10000000))
# RDI is the first parameter of function
for index in range(30):
        convertMemoryToSymbolicVariable(MemoryAccess(0x10000000+index, CPUSIZE.BYTE))
```

Start to processing opcodes



```
emulate(0x4006FD)
while pc:
    opcodes = getConcreteMemoryAreaValue(pc, 16)

    instruction = Instruction()
    instruction.setOpcodes(opcodes)
    instruction.setAddress(pc)

    processing(instruction)
```

Get symbolic expression and solve it



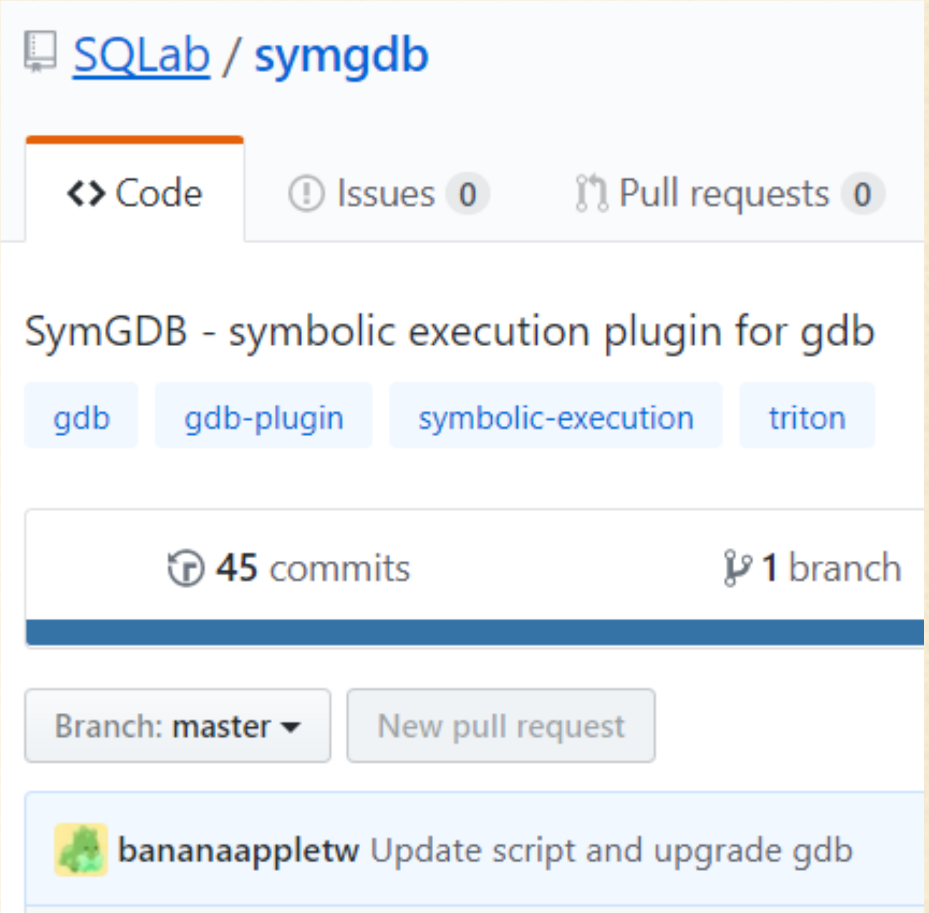
```python
rax   = getSymbolicExpressionFromId(getSymbolicRegisterId(REG.RAX))
eax   = ast.extract(31, 0, rax.getAst())
cstr  = ast.assert_(
            ast.land(
                getPathConstraintsAst(),
                ast.equal(eax, ast.bv(1, 32))
            )
        )
model = getModel(cstr)
for k, v in model.items():
    value = v.getValue()
    getSymbolicVariableFromId(k).setConcreteValue(value)
```

Some problems of Triton

- The whole procedure is too complicated
- High learning cost to use Triton
- With support of debugger, many steps could be simplified

SymGDB

- Repo: https://github.com/SQLab/symgdb
- Symbolic execution support for GDB
- Combined with:
  - Triton
  - GDB Python API
- Symbolic environment
  - symbolize argv



SQLab / symgdb

SymGDB - symbolic execution plugin for gdb

45 commits

1 branch

Branch: master

bananaappletw Update script and upgrade gdb

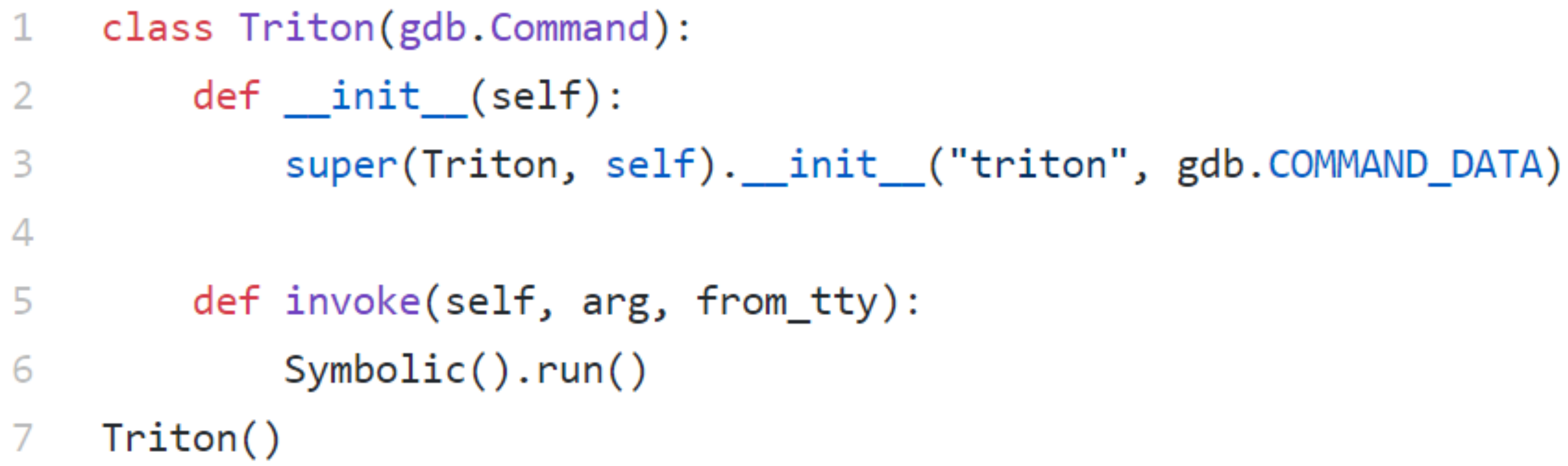
Design and Implementation

- GDB Python API
- Failed method
- Successful method
- Flow
- SymGDB System Structure
- Implementation of System Internals
- Relationship between SymGDB classes
- Supported Commands
- Symbolic Execution Process in GDB
- Symbolic Environment
    - symbolic argv
- Debug tips

GDB Python API

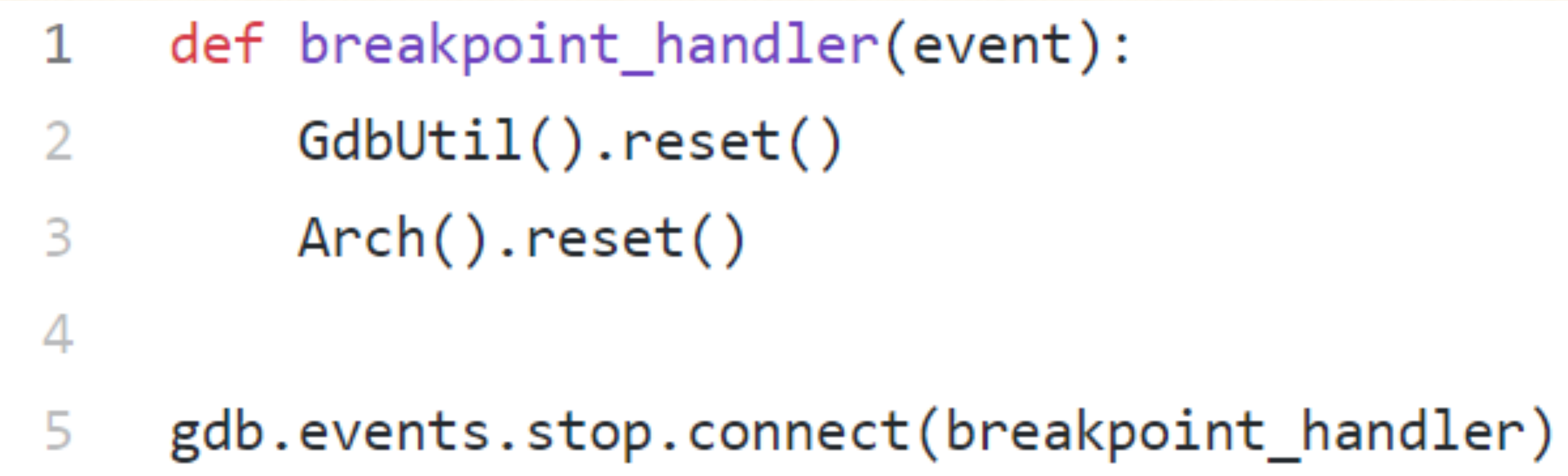
- API: https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html
- Source python script in .gdbinit
- Functionalities:
  - Register GDB command
  - Register event handler (ex: breakpoint)
  - Execute GDB command and get output
  - Read, write, search memory

Register GDB command



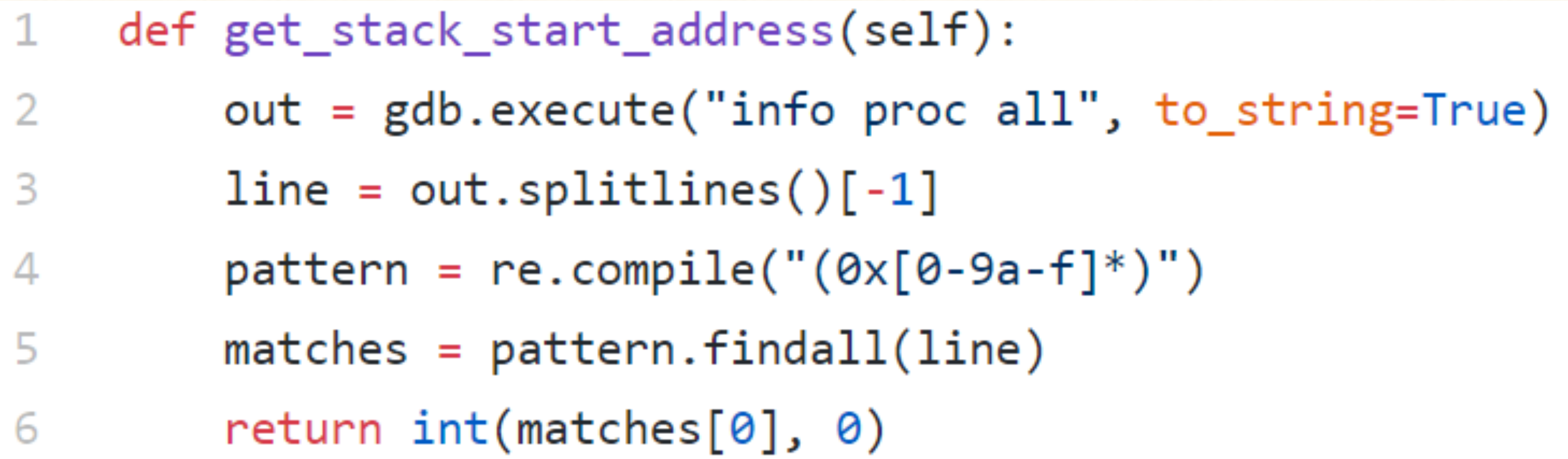
```python
class Triton(gdb.Command):
    def __init__(self):
        super(Triton, self).__init__("triton", gdb.COMMAND_DATA)

    def invoke(self, arg, from_tty):
        Symbolic().run()
Triton()
```

Register event handler



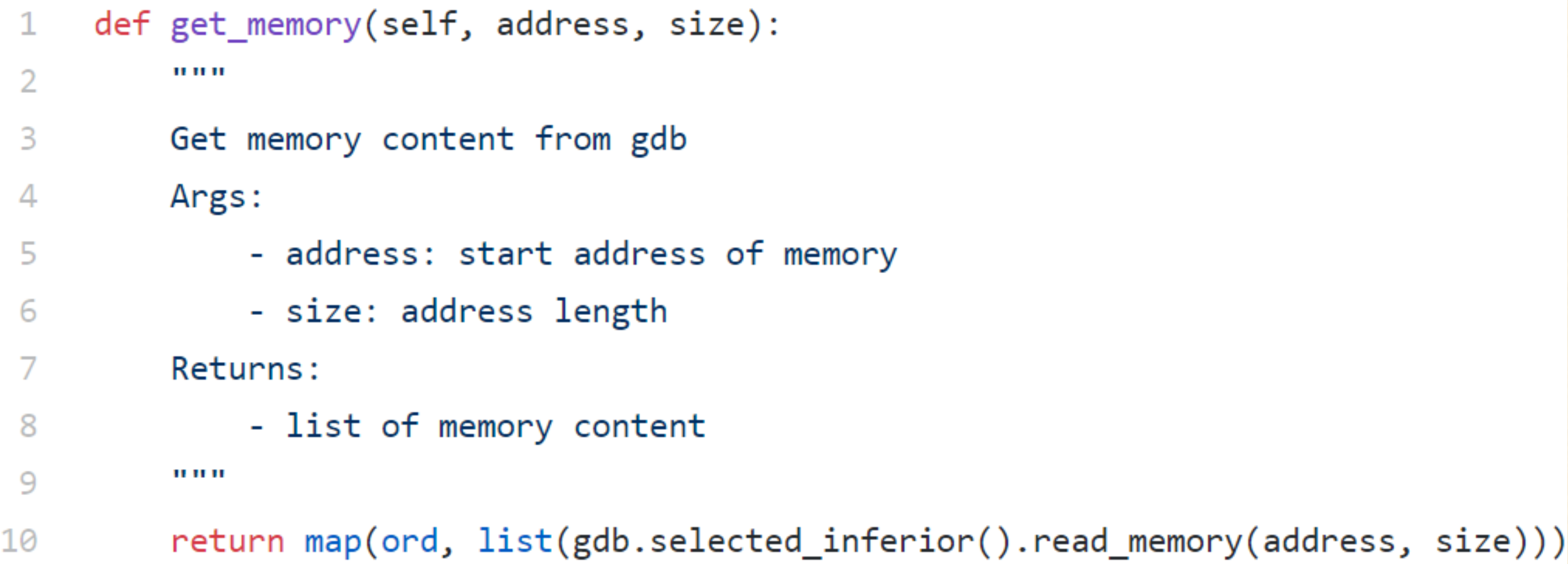
```python
def breakpoint_handler(event):
    GdbUtil().reset()
    Arch().reset()

gdb.events.stop.connect(breakpoint_handler)
```

Execute GDB command and get output



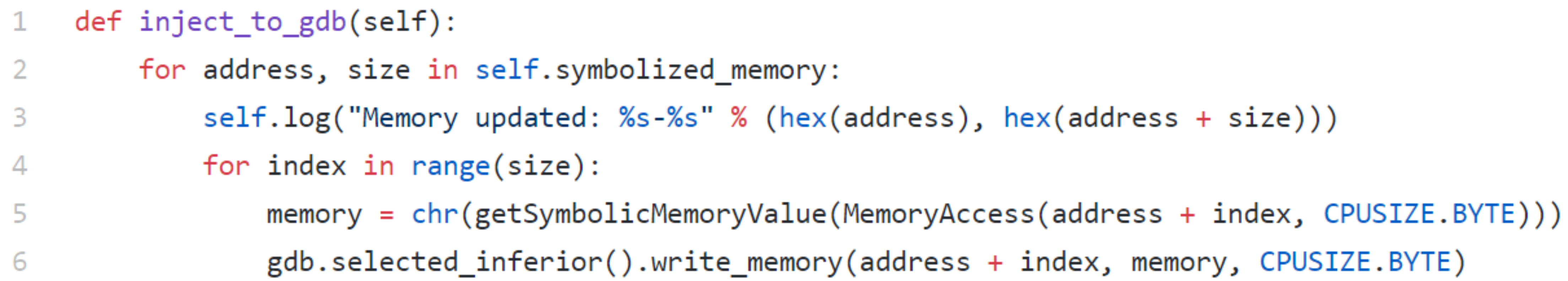
```
def get_stack_start_address(self):
    out = gdb.execute("info proc all", to_string=True)
    line = out.splitlines()[-1]
    pattern = re.compile("(0x[0-9a-f]*)")
    matches = pattern.findall(line)
    return int(matches[0], 0)
```

Read memory



```python
def get_memory(self, address, size):
    """
    Get memory content from gdb
    Args:
        - address: start address of memory
        - size: address length
    Returns:
        - list of memory content
    """
    return map(ord, list(gdb.selected_inferior().read_memory(address, size)))
```

Write memory

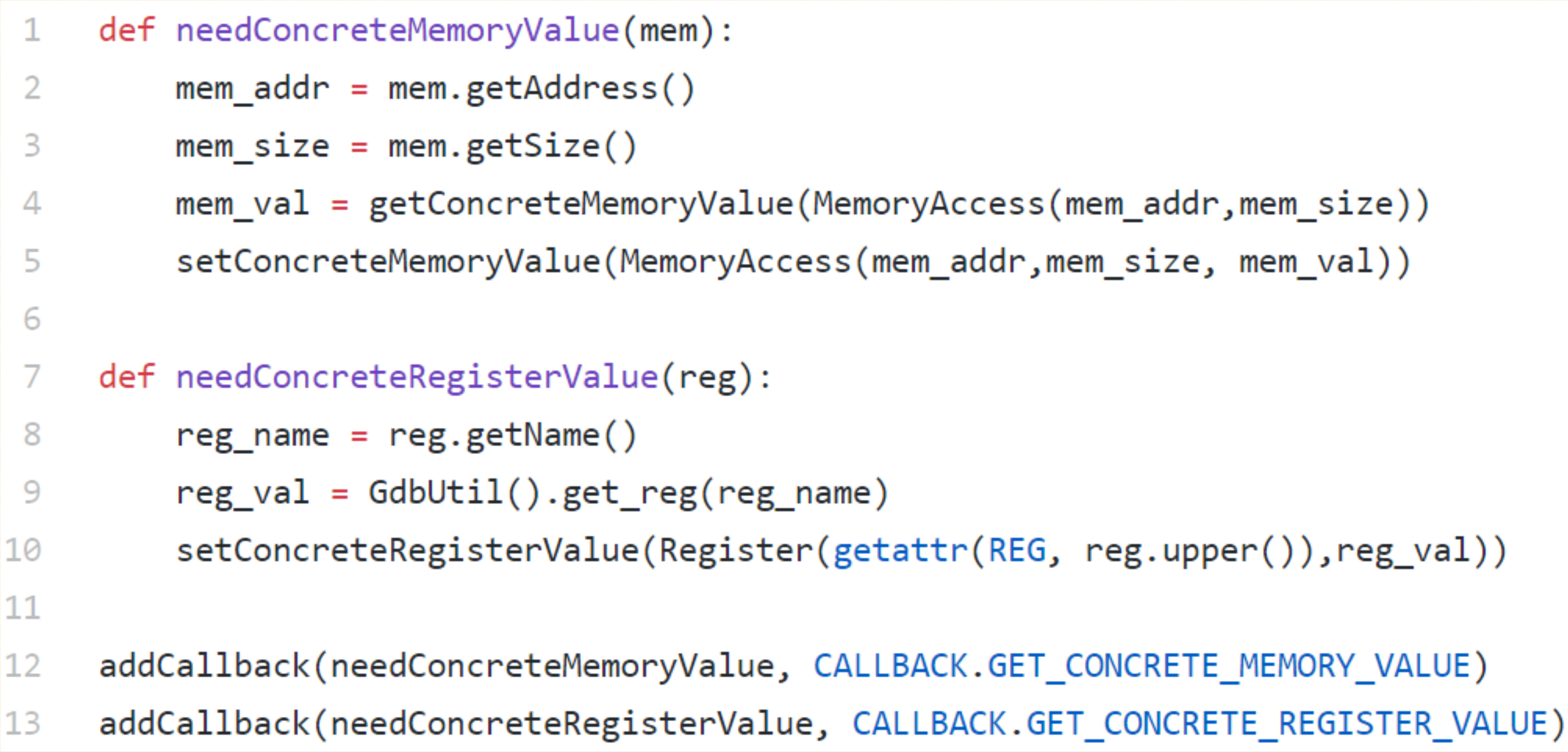


```python
def inject_to_gdb(self):
    for address, size in self.symbolized_memory:
        self.log("Memory updated: %s-%s" % (hex(address), hex(address + size)))
        for index in range(size):
            memory = chr(getSymbolicMemoryValue(MemoryAccess(address + index, CPUSIZE.BYTE)))
            gdb.selected_inferior().write_memory(address + index, memory, CPUSIZE.BYTE)
```

Failed method

- At first, I try to use Triton callback to get memory and register values
- Register callbacks:
  - needConcreteMemoryValue
  - needConcreteRegisterValue
- Process the following sequence of code
  - mov eax, 5
  - mov ebx,eax (**Trigger needConcreteRegisterValue**)
- We need to set Triton context of eax

Triton callbacks



```python
def needConcreteMemoryValue(mem):
    mem_addr = mem.getAddress()
    mem_size = mem.getSize()
    mem_val = getConcreteMemoryValue(MemoryAccess(mem_addr,mem_size))
    setConcreteMemoryValue(MemoryAccess(mem_addr,mem_size, mem_val))

def needConcreteRegisterValue(reg):
    reg_name = reg.getName()
    reg_val = GdbUtil().get_reg(reg_name)
    setConcreteRegisterValue(Register(getattr(REG, reg.upper()),reg_val))

addCallback(needConcreteMemoryValue, CALLBACK.GET_CONCRETE_MEMORY_VALUE)
addCallback(needConcreteRegisterValue, CALLBACK.GET_CONCRETE_REGISTER_VALUE)
```

Problems

- Values from GDB are out of date
- Consider the following sequence of code
- mov eax, 5
- We set breakpoint here, and call Triton's processing()
- mov ebx,eax (trigger callback to get eax value, eax = 5)
- mov eax, 10
- mov ecx, eax (Trigger again, get eax = 5)
- Because context state not up to date

Tried solutions

- Before needed value derived from GDB, check if it is not in the Triton's context yet

Not working!

Triton will fall into infinite loop

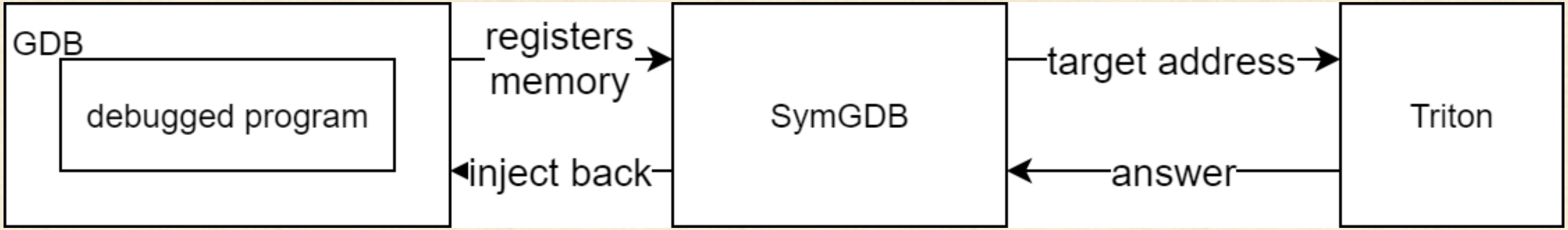
Successful method

- Copy GDB context into Triton
- Load all the segments into Triton context
- Symbolic execution won't affect original GDB state
- User could restart symbolic execution from breakpoint

Flow

- Get debugged program state by calling GDB Python API
- Get the current program state and yield to triton
- Set symbolic variable
- Set the target address
- Run symbolic execution and get output
- Inject back to debugged program state

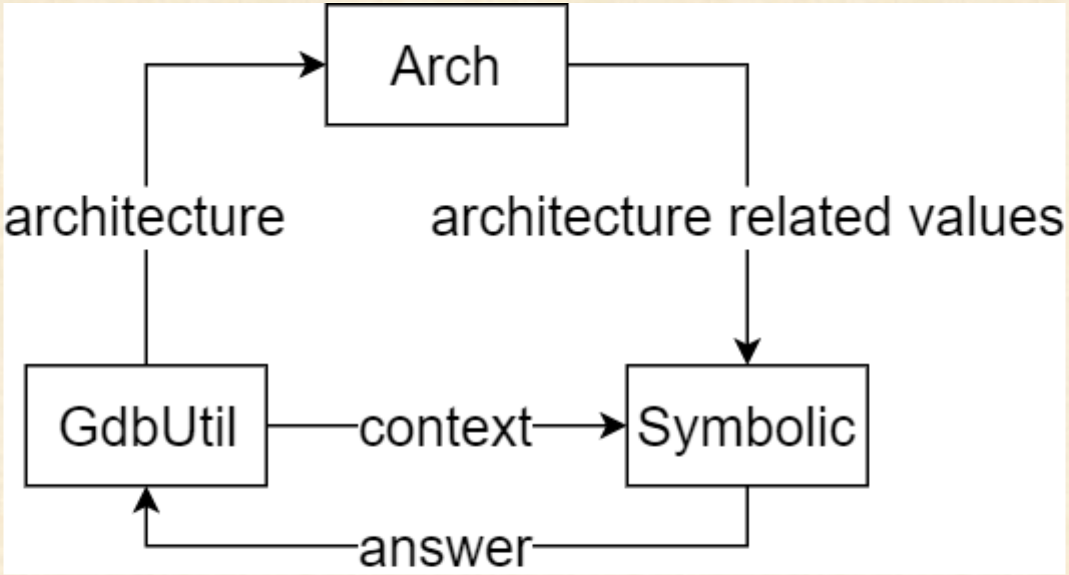
SymGDB System Structure



Implementation of System Internals

- Three classes in the symGDB
  - Arch(), GdbUtil(), Symbolic()
- Arch()
  - Provide different pointer size 、 register name
- GdbUtil()
  - Read write memory 、 read write register
  - Get memory mapping of program
  - Get filename and detect architecture
  - Get argument list
- Symbolic()
  - Set constraint on pc register
  - Run symbolic execution

Relationship between SymGDB classes



Supported Commands

Command	Option	Functionality
symbolize	argv memory [address][size]	Make symbolic
target	address	Set target address
triton	None	Run symbolic execution
answer	None	Print symbolic variables
debug	symbolic gdb	Show debug messages

Symbolic Execution Process in GDB

- gdb.execute("info registers", to_string=True) to get registers
- gdb.selected_inferior().read_memory(address, length) to get memory
- setConcreteMemoryAreaValue and setConcreteRegisterValue to set triton state
- In each instruction, use isRegisterSymbolized to check if pc register is symbolized or not
- Set target address as constraint
- Call getModel to get answer
- gdb.selected_inferior().write_memory(address, buf, length) to inject back to debugged program state

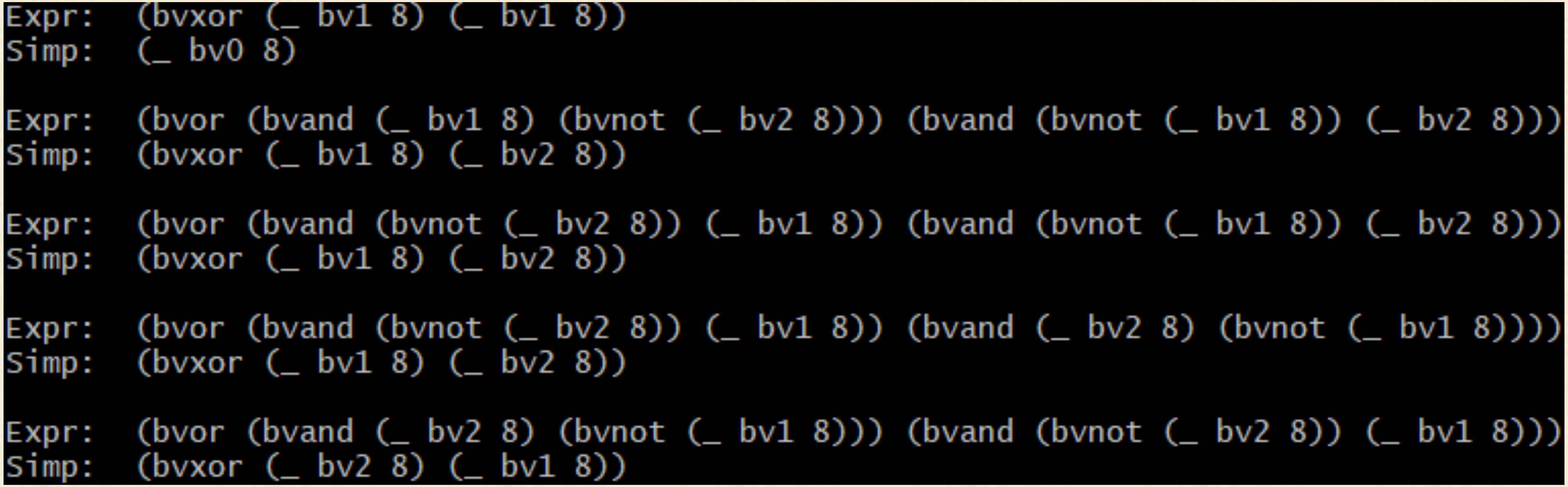
Symbolic Environment: symbolic argv

- Using "info proc all" to get stack start address
- Examining memory content from stack start address
  - argc
  - argv[0]
  - argv[1]
  - ……
  - null
  - env[0]
  - env[1]
  - ……
  - null

argc	argument counter(integer)
argv[0]	program name (pointer)
argv[1]	program args (pointers)
…	
argv[argc-1]	
null	end of args (integer)
env[0]	environment variables (pointers)
env[1]	
…	
env[n]	
null	end of environment (integer)

Debug tips

- Simplify: https://github.com/JonathanSalwan/Triton/blob/master/src/examples/python/simplification.py



```
Expr:   (bvxor (_ bv1 8) (_ bv1 8))
Simp:   (_ bv0 8)

Expr:   (bvor (bvand (_ bv1 8) (bvnot (_ bv2 8))) (bvand (bvnot (_ bv1 8)) (_ bv2 8)))
Simp:   (bvxor (_ bv1 8) (_ bv2 8))

Expr:   (bvor (bvand (bvnot (_ bv2 8)) (_ bv1 8)) (bvand (bvnot (_ bv1 8)) (_ bv2 8)))
Simp:   (bvxor (_ bv1 8) (_ bv2 8))

Expr:   (bvor (bvand (bvnot (_ bv2 8)) (_ bv1 8)) (bvand (_ bv2 8) (bvnot (_ bv1 8))))
Simp:   (bvxor (_ bv1 8) (_ bv2 8))

Expr:   (bvor (bvand (_ bv2 8) (bvnot (_ bv1 8))) (bvand (bvnot (_ bv2 8)) (_ bv1 8)))
Simp:   (bvxor (_ bv2 8) (_ bv1 8))
```

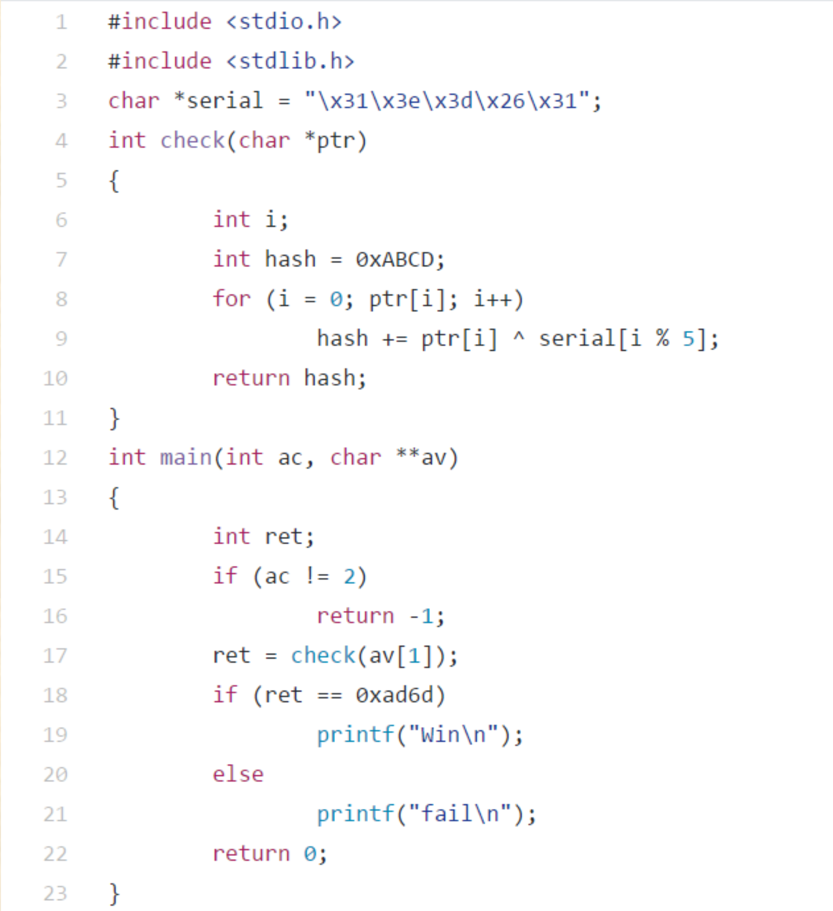
Demo

- Examples
  - crackme hash
  - crackme xor
- GDB commands
- Combined with Peda

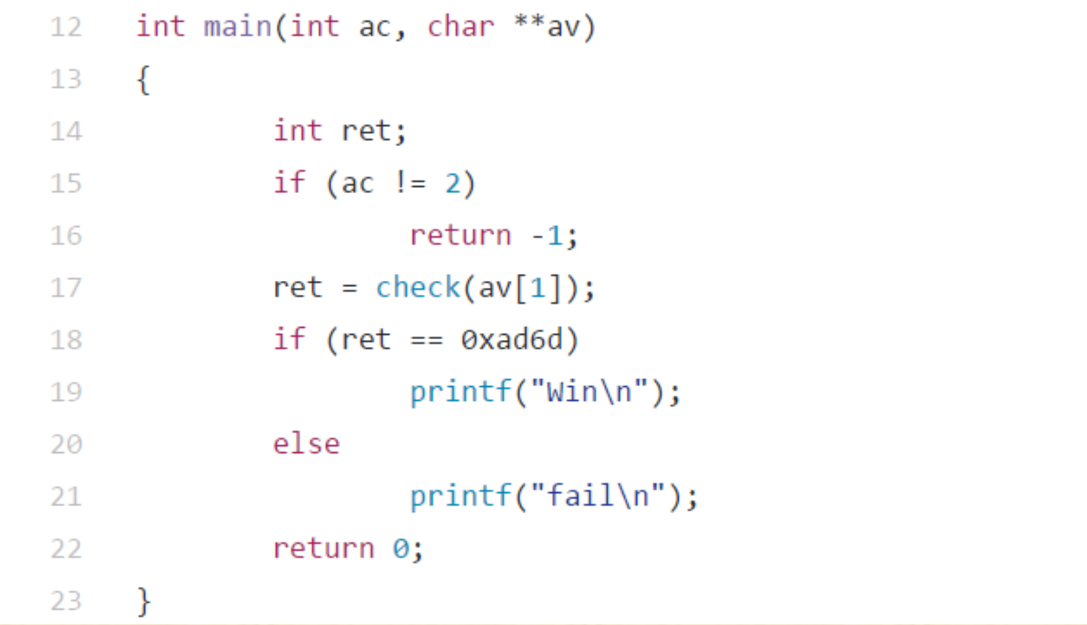
crackme hash

- Source: https://github.com/illera88/Ponce/blob/master/examples/crackme_hash.cpp
- Program will pass argv[1] to check function
- In check function, argv[1] xor with serial(fixed string)
- If sum of xored result equals to 0xABCD
  - print "Win"
- else
  - print "fail"

crackme hash

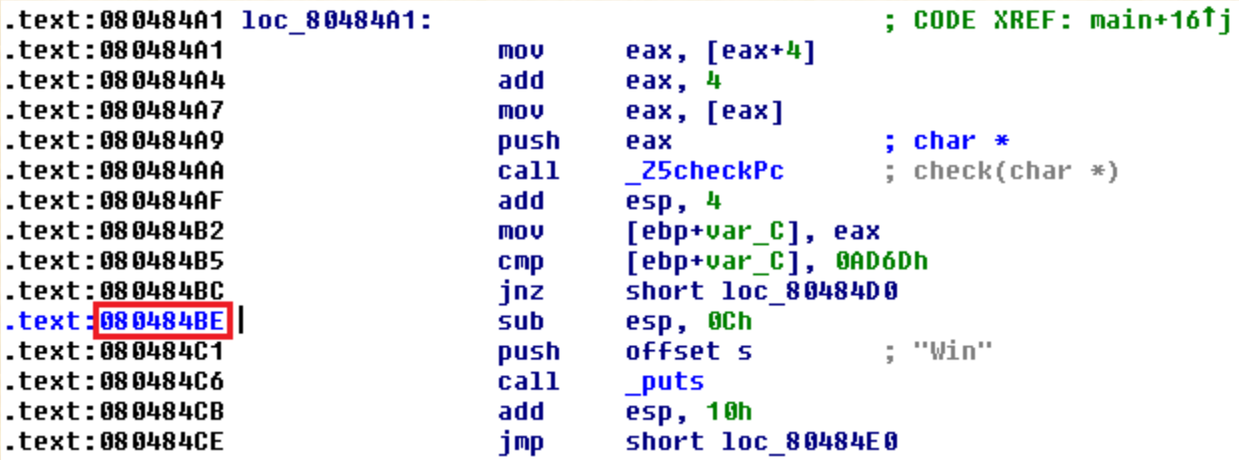


```c
#include <stdio.h>
#include <stdlib.h>
char *serial = "\x31\x3e\x3d\x26\x31";
int check(char *ptr)
{
        int i;
        int hash = 0xABCD;
        for (i = 0; ptr[i]; i++)
                hash += ptr[i] ^ serial[i % 5];
        return hash;
}
int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;
        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```



```c
int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;
        ret = check(av[1]);
        if (ret == 0xad6d)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

crackme hash



```
.text:080484A1 loc_80484A1:                              ; CODE XREF: main+16↑j
.text:080484A1                 mov     eax, [eax+4]
.text:080484A4                 add     eax, 4
.text:080484A7                 mov     eax, [eax]
.text:080484A9                 push    eax             ; char *
.text:080484AA                 call    _Z5checkPc      ; check(char *)
.text:080484AF                 add     esp, 4
.text:080484B2                 mov     [ebp+var_C], eax
.text:080484B5                 cmp     [ebp+var_C], 0AD6Dh
.text:080484BC                 jnz     short loc_80484D0
.text:080484BE                 sub     esp, 0Ch
.text:080484C1                 push    offset s        ; "Win"
.text:080484C6                 call    _puts
.text:080484CB                 add     esp, 10h
.text:080484CE                 jmp     short loc_80484E0
```

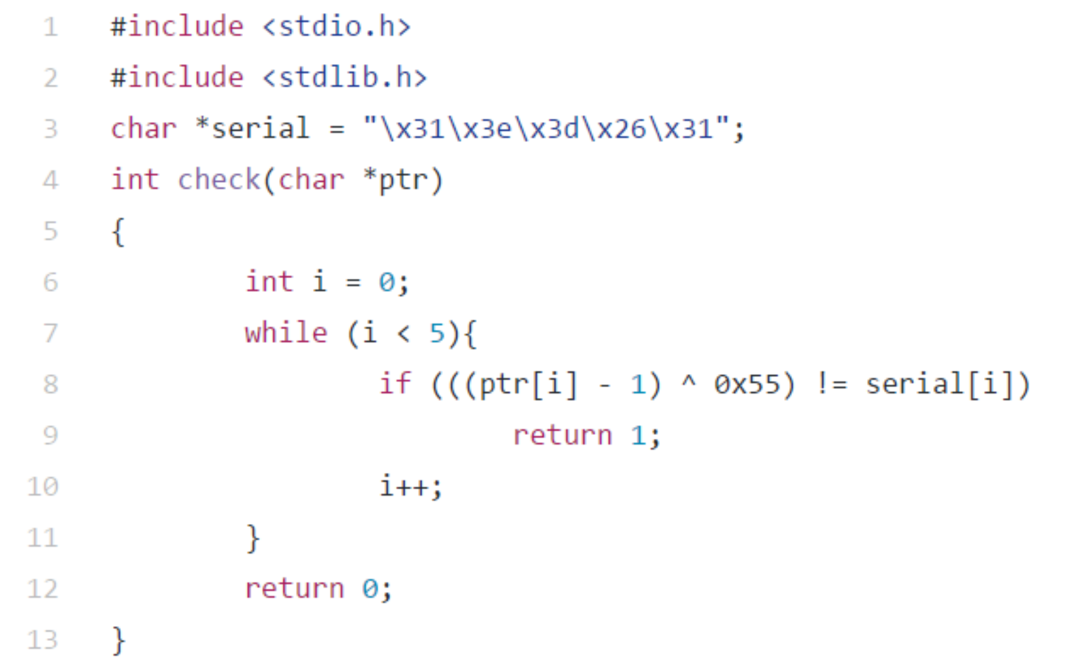
crackme hash

```
apple@apple-All-Series:~/gdb-symbolic/examples$
```

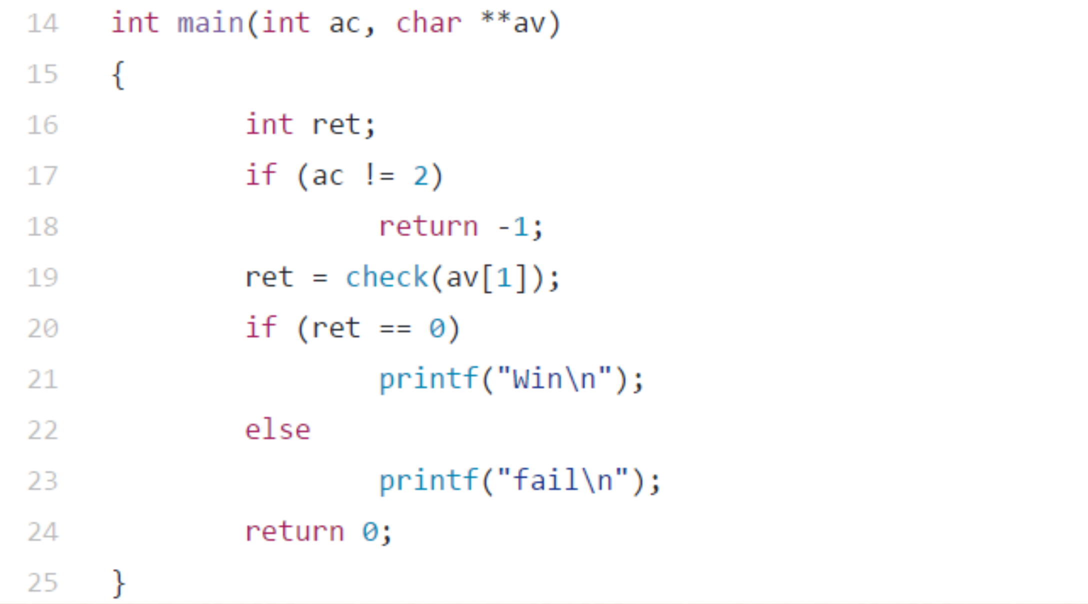
crackme xor

- Source: https://github.com/illera88/Ponce/blob/master/examples/crackme_xor.cpp
- Program will pass argv[1] to check function
- In check function, argv[1] xor with 0x55
- If xored result not equals to serial(fixed string)
  - return 1
  - print "fail"
- else
  - go to next loop
- If program go through all the loop
  - return 0
  - print "Win"

crackme xor

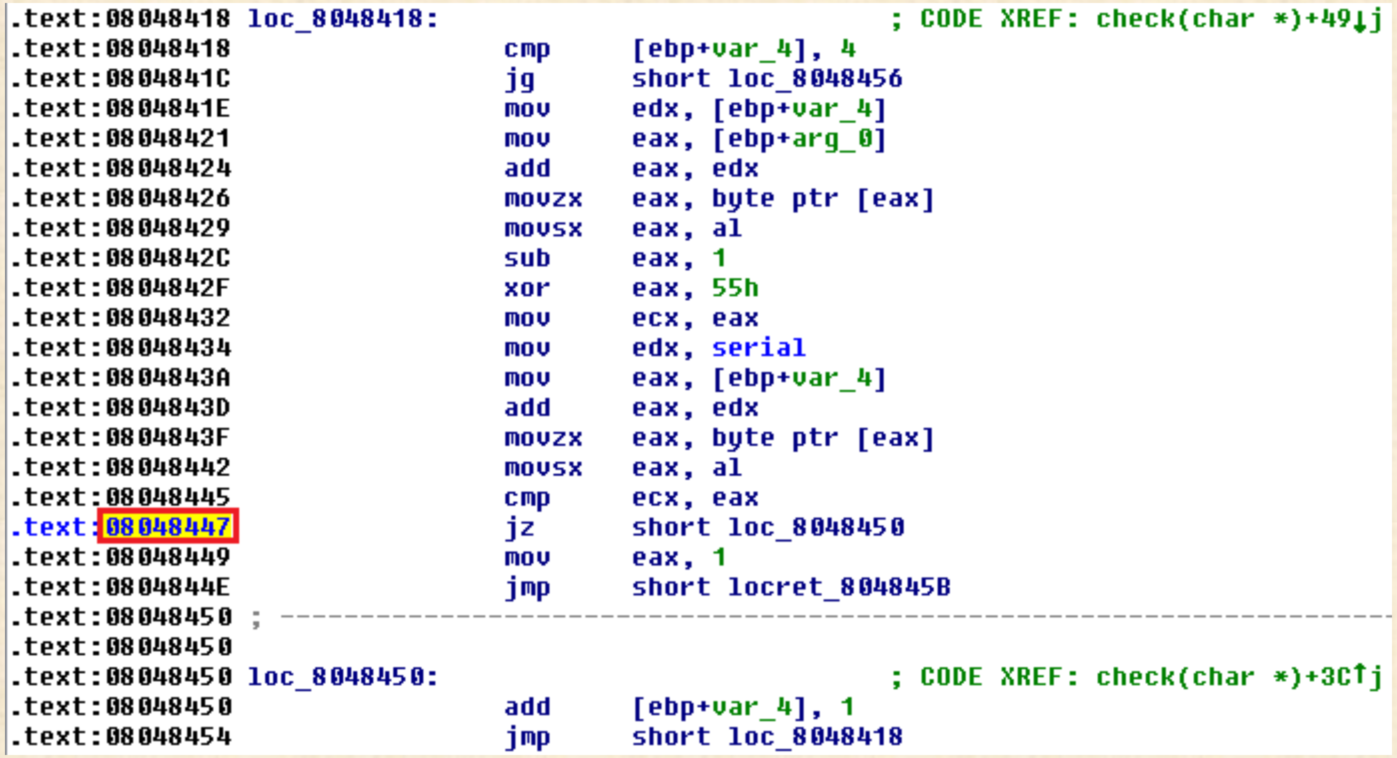


```c
#include <stdio.h>
#include <stdlib.h>
char *serial = "\x31\x3e\x3d\x26\x31";
int check(char *ptr)
{
        int i = 0;
        while (i < 5){
                if (((ptr[i] - 1) ^ 0x55) != serial[i])
                        return 1;
                i++;
        }
        return 0;
}
```



```c
int main(int ac, char **av)
{
        int ret;
        if (ac != 2)
                return -1;
        ret = check(av[1]);
        if (ret == 0)
                printf("Win\n");
        else
                printf("fail\n");
        return 0;
}
```

crackme xor

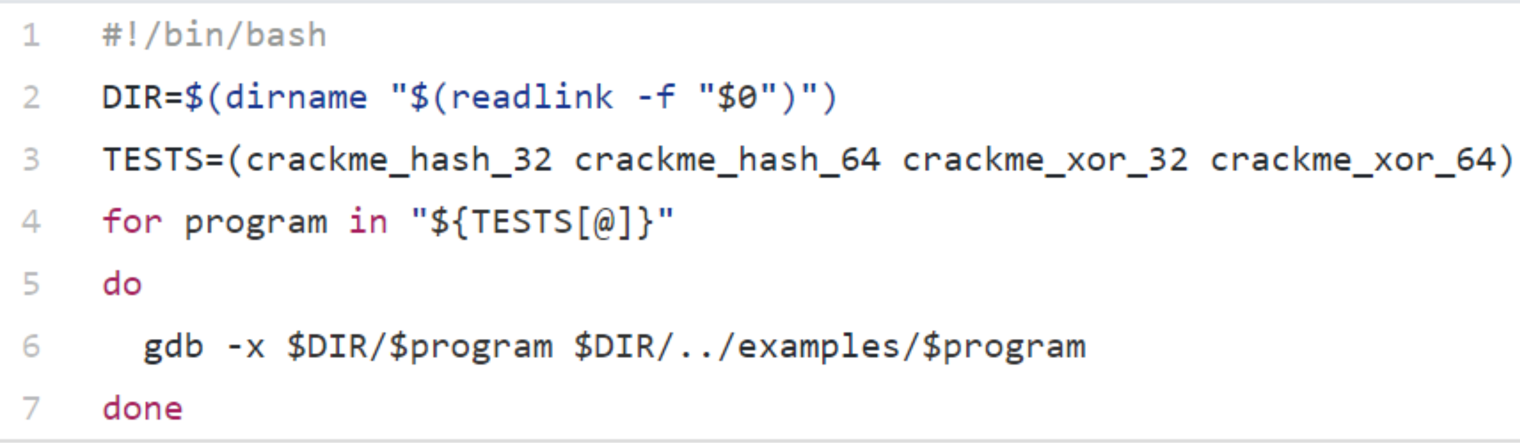


```
.text:08048418 loc_8048418:                                    ; CODE XREF: check(char *)+49↓j
.text:08048418                 cmp     [ebp+var_4], 4
.text:0804841C                 jg      short loc_8048456
.text:0804841E                 mov     edx, [ebp+var_4]
.text:08048421                 mov     eax, [ebp+arg_0]
.text:08048424                 add     eax, edx
.text:08048426                 movzx   eax, byte ptr [eax]
.text:08048429                 movsx   eax, al
.text:0804842C                 sub     eax, 1
.text:0804842F                 xor     eax, 55h
.text:08048432                 mov     ecx, eax
.text:08048434                 mov     edx, serial
.text:0804843A                 mov     eax, [ebp+var_4]
.text:0804843D                 add     eax, edx
.text:0804843F                 movzx   eax, byte ptr [eax]
.text:08048442                 movsx   eax, al
.text:08048445                 cmp     ecx, eax
.text:08048447                 jz      short loc_8048450
.text:08048449                 mov     eax, 1
.text:0804844E                 jmp     short locret_804845B
.text:08048450 ; ---------------------------------------------------------------------------
.text:08048450
.text:08048450 loc_8048450:                                    ; CODE XREF: check(char *)+3C↑j
.text:08048450                 add     [ebp+var_4], 1
.text:08048454                 jmp     short loc_8048418
```

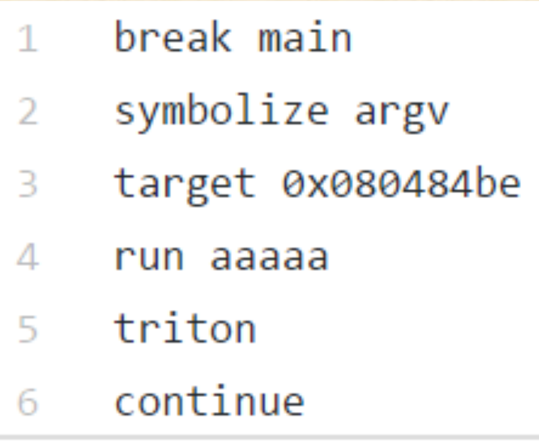
crackme xor

```
apple@apple-All-Series:~/gdb-symbolic/examples$
```

GDB commands



```bash
#!/bin/bash
DIR=$(dirname "$(readlink -f "$0")")
TESTS=(crackme_hash_32 crackme_hash_64 crackme_xor_32 crackme_xor_64)
for program in "${TESTS[@]}"
do
  gdb -x $DIR/$program $DIR/../examples/$program
done
```



```
break main
symbolize argv
target 0x080484be
run aaaaa
triton
continue
```

GDB commands

```
apple@apple-All-Series:~/gdb-symbolic/tests$
```

Combined with Peda

- Same demo video of crackme hash
- Using find(peda command) to find argv[1] address
- Using symbolize memory argv[1]_address argv[1]_length to symbolic argv[1] memory

Combined with Peda

```
apple@apple-All-Series:~/gdb-symbolic/examples$
```

Drawbacks

- Triton doesn't support GNU c library
- Why?
- SMT Semantics Supported: https://triton.quarkslab.com/documentation/doxygen/SMT_Semantics_Supported_page.html
- Triton has to implement system call interface to support GNU c library a.k.a. support "int 0x80"

Triton versus Angr

Difference	Triton	Angr
Architecture support	x86 amd64	x86 amd64 arm ……
GNU c library support	No	Yes
Path explore	No	Yes

References

- Wiki: https://en.wikipedia.org/wiki/Symbolic_execution
- Triton: https://triton.quarkslab.com/
- GDB Python API: https://sourceware.org/gdb/onlinedocs/gdb/Python-API.html
- Peda: https://github.com/longld/peda
- Ponce: https://github.com/illera88/Ponce
- Angr: http://angr.io/

Bamboofox

Q & A

Thank you