Votre dévermineur en Python

Pour toujours plus de bugs

switch @hitchhack, 17:06:2022

self

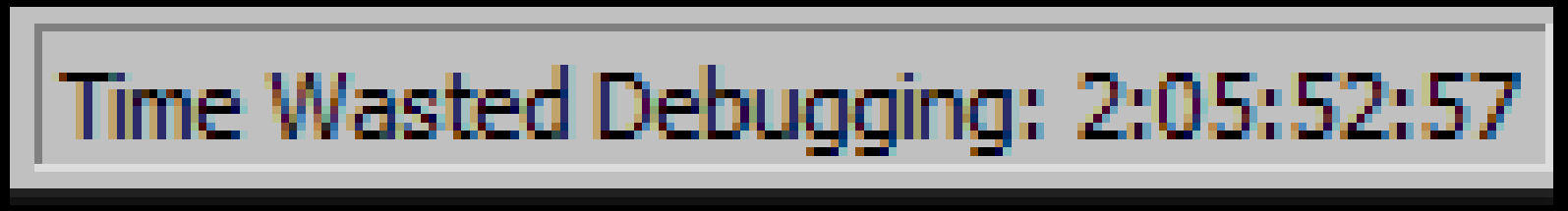
@swuitch
Team FR 2019
https://0xswitch.fr

acceis

{ pentest, TI interne / externe, IoT, RE, redteam, PASSI, CESTI, 8.6 }

on recrute

Pq ?



Scripter la récupération de valeurs

- grep coredump > api GDB
- outils overkill (angr, miasm, unicorn)
- emulation vs analyse dynamique

Identifier les rôles et fonctionnalités d’un dévermineur

- breakpoints
- Intéraction avec le processus
- gdb –q gdb

Qu’est ce qu’on veut

Lecture / écriture de la mémoire du process

Single Step

Breakpoints

Step backward

Renationaliser les autoroutes

Utiliser les tickets restaurant au bar

Qu’est ce qu’on veut vraiment

~~Lecture / écriture de la mémoire du process~~

~~Single Step~~

~~Breakpoints~~

~~Step backward~~

~~Renationaliser les autoroutes~~

Utiliser les tickets restaurant au bar

Qu’est ce qu’on veut vraiment ²

Lecture / écriture de la mémoire du process ⟶ API Kernel Linux

Single Step ⟶ API Kernel Linux

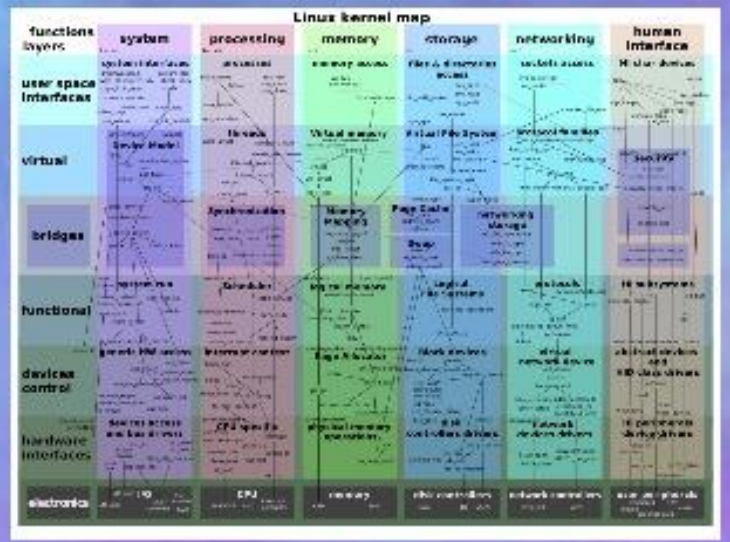
Breakpoints ⟶

~~Step backward~~

Renationaliser les autoroutes ⟶

Utiliser les tickets restaurant au bar

ah,yes.



"API Kernel linux"

PTRACE everything

Syscall qui fait le café

- lecture / écriture mémoire : `PTRACE_(PEEK|POKE)TEXT`
- lecture / écriture registres : `PTRACE_(GET|SET)REGS`
- single step : `PTRACE_SINGLESTEP`
- continuer : `PTRACE_CONT`
- s'attacher : `PTRACE_TRACEME + PTRACE_ATTACH`

```
PTRACE(2)                    Linux Programmer's Manual                    PTRACE(2)

NAME
       ptrace - process trace

SYNOPSIS
       #include <sys/ptrace.h>

       long ptrace(enum __ptrace_request request, pid_t pid,
                   void *addr, void *data);
```

Attachement, scission et exécution d'enfants

/proc/sys/kernel/yama/ptrace_scope = 1 ; par défaut
Tldr: on attache que ses propres enfants

fork() : création d'un processus identique que l'on controle

[parent]

1. waitpid(): attend que l'enfant change d'état (STOPPED)
4. ptrace(PTRACE_CONT) : demande à l'enfant de continuer son exécution

[enfant]

2. ptrace(PTRACE_TRACEME): passe en STOPPED s'il reçoit un signal
3. execv("/bin/ls"): charge le binaire et l'execute puis envoie un SIGTRAP après le syscall

```python
libc = ctypes.CDLL('libc.so.6')
libc.ptrace.argtypes = [ctypes.c_uint64, ctypes.c_uint64, ctypes.c_void_p, ctypes.c_void_p]
libc.ptrace.restype = ctypes.c_uint64

pipe_stdin, pipe_stdout = os.pipe(), os.pipe()

pid = os.fork()

if pid == 0: # enfant (pid == 0)
    os.close(pipe_stdin[1])
    os.dup2(pipe_stdin[0], 0)

    libc.ptrace(PTRACE_TRACEME, 0, 0, 0)           # process state will be STOPPED after the first signal received
    os.execvp(file_to_dbg, [file_to_dbg, "1", "2"]) # send a SIGTRAP
else: # parent (pid == pid + 1)
    os.waitpid(pid, 0)                             # wait state to be STOPPED
```

```
switch    182064  146056  3 00:28 pts/11   00:00:00 python dbg.py /home/switch/projets/workshop/debug/code/hello
switch    182065  182064  0 00:28 pts/11   00:00:00 /home/switch/projets/workshop/debug/code/hello 1 2
```

Lecture / écriture des registres

Syscall qui fait le café²

Tldr: on créer une structure qui se remplit avec PTRACE_GETREGS

```python
def read_regs(libc, pid):
    regs_struct = user_regs_struct_x64()
    libc.ptrace(PTRACE_GETREGS, pid, None, ctypes.byref(regs_struct))
    return regs_struct

saved_ins = bp(libc, pid, bp_addr)
print(f"RIP: {hex(regs.rip)}")
```

```python
_fields_ = [
    ("r15", ctypes.c_ulonglong),
    ("r14", ctypes.c_ulonglong),
    ("r13", ctypes.c_ulonglong),
    ("r12", ctypes.c_ulonglong),
    ("rbp", ctypes.c_ulonglong),
    ("rbx", ctypes.c_ulonglong),
    ("r11", ctypes.c_ulonglong),
    ("r10", ctypes.c_ulonglong),
    ("r9", ctypes.c_ulonglong),
    ("r8", ctypes.c_ulonglong),
    ("rax", ctypes.c_ulonglong),
    ("rcx", ctypes.c_ulonglong),
    ("rdx", ctypes.c_ulonglong),
    ("rsi", ctypes.c_ulonglong),
    ("rdi", ctypes.c_ulonglong),
    ("orig_rax", ctypes.c_ulonglong),
    ("rip", ctypes.c_ulonglong),
    ("cs", ctypes.c_ulonglong),
    ("eflags", ctypes.c_ulonglong),
    ("rsp", ctypes.c_ulonglong),
    ("ss", ctypes.c_ulonglong),
    ("fs_base", ctypes.c_ulonglong),
    ("gs_base", ctypes.c_ulonglong),
    ("ds", ctypes.c_ulonglong),
    ("es", ctypes.c_ulonglong),
    ("fs", ctypes.c_ulonglong),
    ("gs", ctypes.c_ulonglong),
]
```

Breakpoints

Pas de café ⟶ s/instruction/0xcc/

Tldr: on remplace l'instruction avec un sigtrap et le debugger nous rend la main quand il passe dessus

```python
def bp(libc, pid, addr):
    backup = libc.ptrace(PTRACE_PEEKDATA, pid, ctypes.c_void_p(addr), None)
    trap = (backup & 0xFFFFFFFFFFFFFF00) + 0xCC
    # writing the 0xcc
    libc.ptrace(PTRACE_POKEDATA, pid, ctypes.c_void_p(addr), trap)
    return backup
```

Breakpoints bis

```python
def pass_bp(libc, pid, addr, value):
    libc.ptrace(
        PTRACE_POKEDATA, pid, ctypes.c_void_p(addr), value
    )
    # as the 0xcc int has been executed we have to go 1 instruction backward
    # or simply use the breakpoint address as RIP value
    regs = read_regs(libc, pid)
    regs.rip = addr
    libc.ptrace(PTRACE_SETREGS, pid, None, ctypes.byref(regs))
    # going through the instruction
    libc.ptrace(PTRACE_SINGLESTEP, pid, None, None)
    os.waitpid(pid, 0)
```

```
λ ackira ~/projets/workshop/debug/code » python dbg.py /home/switch/projets/workshop/debug/code/hello
[parent] RIP: 0x7f5f757ab930
[parent] bp set @ 0x4011b8
[child] before meh
[parent] RIP before BP: 0x4011b8
[parent] meh arg1 (rdi): 0x0
[parent] meh arg2 (rsi): 0x25
[parent] <CsInsn 0x4011b8 [e879ffffff]: call 0x401136>
[parent] <CsInsn 0x4011bd [8945fc]: mov dword ptr [rbp - 4], eax>
[parent] <CsInsn 0x4011c0 [488d056e0e0000]: lea rax, [rip + 0xe6e]>
[parent] RIP after BP: 0x401136
continue >>
[child] 0
[child] 1
[child] 3
[child] 6
[child] 10
[child] 15
[child] 21
[child] 28
[child] 36
[child] 45
[child] 55
[child] 66
[child] 78
[child] 91
[child] 105
[child] 120
[child] 136
[child] 153
[child] 171
[child] 190
[child] 210
[child] 231
[child] 253
[child] 276
[child] 300
[child] 325
[child] 351
[child] 378
[child] 406
[child] 435
[child] 465
[child] 496
[child] 528
[child] 561
[child] 595
[child] 630
[child] 666
[child] after meh
[parent] quitting
```

```python
if pid == 0:
   os.close(pipe_stdin[1])
   os.dup2(pipe_stdin[0], 0)
   libc.ptrace(PTRACE_TRACEME, 0, 0, 0) # process state will be STOPPED after the first signal received
   os.execvp(file_to_dbg, [file_to_dbg, "1", "2"]) # send a SIGTRAP
else:
   x = os.waitpid(pid, 0) # wait state to be STOPPED
   regs = read_regs(libc, pid) # read registers
   saved_ins = bp(libc, pid, bp_addr)
   rprint(f"RIP: {hex(regs.rip)}")
   rprint("bp set @ 0x{:x}".format(bp_addr))

   cont(libc, pid) # continue until sigtrap (bkreapoint)

   # breakpoint reached
   regs = read_regs(libc, pid)
   rprint(f"RIP before BP: {hex(regs.rip - 1)}")
   rprint(f"meh arg1 (rdi): {hex(regs.rdi)}")
   rprint(f"meh arg2 (rsi): {hex(regs.rsi)}")

   # disass next instruction
   ins = dis.disasm(text.data[(regs.rip - 1)  - text.sh_addr:], (regs.rip - 1))
   for i in range(3):
       next_ins = next(ins)
       rprint(green(f"{next_ins}"))

    # s/0xcc/backup/
   pass_bp(libc, pid, bp_addr, saved_ins)

   regs = read_regs(libc, pid)
   rprint(f"RIP after BP: {hex(regs.rip)}")

   input("continue >>")
   # continue until la mort
   cont(libc, pid)
   rprint("quitting")
   libc.ptrace(PTRACE_KILL, pid, 0, 0)
```

```python
import ctypes
import os
import sys
from hellf import ELF
from capstone import *
from capstone.x86 import *
from pwnyy import red, green

PTRACE_TRACEME = 0
PTRACE_PEEKTEXT = 1
PTRACE_PEEKDATA = 2
PTRACE_PEEKUSER = 3
PTRACE_POKEDATA = 5
PTRACE_CONT = 7
PTRACE_KILL = 8
PTRACE_SINGLESTEP = 9
PTRACE_GETREGS = 12
PTRACE_SETREGS = 13
PTRACE_ATTACH = 16
PTRACE_DETACH = 17
PTRACE_SETOPTIONS = 0x4202

rprint = lambda x: print(red("[parent]") + f" {x}")

class user_regs_struct_x64(ctypes.Structure):

    wanted = ["rax", "rbx", "rcx", "rdx", "rsi", "rdi", "rbp", "rsp", "rip",] + [
        "r{}".format(i) for i in range(8, 16)
    ]

    _fields_ = [...]

    def __repr__(self):
        return "\n".join([reg + " : " + hex(getattr(self, reg)) for reg in self.wanted])


def read_regs(libc, pid):
    regs_struct = user_regs_struct_x64()
    libc.ptrace(PTRACE_GETREGS, pid, None, ctypes.byref(regs_struct))
    return regs_struct

def bp(libc, pid, addr):
    backup = libc.ptrace(PTRACE_PEEKDATA, pid, ctypes.c_void_p(addr), None)
    # trap = (backup & 0xffffffffffffff00) + 0xcc
    trap = (backup & 0xFFFFFFFFFFFFFF00) + 0xCC
    # writing the 0xcc
    libc.ptrace(PTRACE_POKEDATA, pid, ctypes.c_void_p(addr), trap)
    return backup

def pass_bp(libc, pid, addr, value):
    libc.ptrace(
        PTRACE_POKEDATA, pid, ctypes.c_void_p(addr), value
    )
    # as the 0xcc int has been executed we have to go 1 instruction backward
    # or simply use the breakpoint address as RIP value
    regs = read_regs(libc, pid)
    regs.rip = addr
    libc.ptrace(PTRACE_SETREGS, pid, None, ctypes.byref(regs))
    # going through the instruction
    libc.ptrace(PTRACE_SINGLESTEP, pid, None, None)
    os.waitpid(pid, 0)

def cont(libc, pid):
    libc.ptrace(PTRACE_CONT, pid, None, None)
    os.waitpid(pid, 0)


if __name__ == "__main__":
    dis = Cs(CS_ARCH_X86, CS_MODE_64)

    libc = ctypes.CDLL('libc.so.6')
    libc.ptrace.argtypes = [ctypes.c_uint64, ctypes.c_uint64, ctypes.c_void_p, ctypes.c_void_p]
    libc.ptrace.restype = ctypes.c_uint64

    file_to_dbg = sys.argv[1]
    elf = ELF(file_to_dbg)
    text = elf.get_section_by_name('.text')
    bp_addr = 0x0000000000401156

    pipe_stdin, pipe_stdout = os.pipe()
    pid = os.fork()

    if pid == 0:
        os.close(pipe_stdin[1])
        os.dup2(pipe_stdin[0], 0)
        libc.ptrace(PTRACE_TRACEME, 0, 0, 0)
        os.execvp(file_to_dbg, [file_to_dbg, "a", "a"])
    else:
        os.waitpid(pid, 0)

        regs = read_regs(libc, pid)
        saved_ins = bp(libc, pid, bp_addr)
        rprint("RIP: {hex(regs.rip)}")
        rprint("bp set @ 0x{:x}".format(bp_addr))

        cont(libc, pid)

        regs = read_regs(libc, pid)
        rprint(f"RIP before BP: {hex(regs.rip - 1)}")
        rprint(f"meh arg1 (rdi): {hex(regs.rdi)}")
        rprint(f"meh arg2 (rsi): {hex(regs.rsi)}")

        ins = dis.disasm(text.data[(regs.rip - 1)  - text.sh_addr:], (regs.rip - 1))
        for i in range(1):
            next_ins = next(ins)
            rprint(green(f"{next_ins}"))

        pass_bp(libc, pid, bp_addr, saved_ins)

        regs = read_regs(libc, pid)
        rprint(f"RIP after BP: {hex(regs.rip)}")

        input("continue >>")
        cont(libc, pid)
        rprint("quitting")
        libc.ptrace(PTRACE_KILL, pid, 0, 0)
```

- https://0xswitch.fr/CTF/fcsc-2021-shuffleme
- FCSC 2022 – Souk
- Librairie Python : Pyning w/ Hellf

KTHXBYE!