INSIGHT Interactive Inspector

A tutorial to symbolic debugging with `iii`

E. Fleury, G. Point, A. Vincent

Table of Contents

1 Introduction

This document describes the use of the `iii` tool for the analysis of an obfuscated executable file. This tutorial does not cover all features of the tool but presents the most usual ones. `iii` is essentially a simulator built on top of Insight framework ([Insight], page 27). Symbolic simulation can be an efficient tool for the debugging and the understanding of program behaviors.

Insight framework permits to interpret program semantics over different domains. Even if its design is quite independent of the interpretation domain, `iii` allows, for now, only two kinds of values: concrete and formula. In the first case, the tool behaves roughly like a classical debugger. However, `iii` is limited to the interpretation of the internal model of the program i.e., its microcode. Many low-level aspects are not yet captured by the framework: system calls, dynamic loading, multi-threading and so on. To be short, microcode model has threes kind of instructions:

1. **Assignments** *lv* := *E* where *lv* is a *l-value* i.e., a register or a memory cell and *E* is a bitvector expression.
2. **Guarded static jumps** change the program counter to an address known *a priori*, if the guard is satisfied.
3. **Dynamic jumps** that change the program counter to an address computed on-the-fly by the program.

By default, values are interpreted as formulas. In this case `iii` behaves like a symbolic simulator ([JK76], page 27). Since the arising of efficient solvers, symbolic simulation has became an effective tool for the analysis of programs. For more details on symbolic simulation we refer the reader to the literature, this document focus only on main ideas.

Regardless of the interpretation domain, the simulator maintains a state of the simulation that models the content of the memory and registers, and the value of the program counter. In the context of symbolic simulation this state associates a formula to each byte of the memory and to each register that have been accessed during the symbolic execution; the program counter is a concrete address that points somewhere into the loaded memory. In addition to these three components, a *symbolic state* possesses a fourth one: a formula called the *path-condition* which is initially set to `true`.

Each time an instruction of the microcode is interpreted, the symbolic state is changed as follows:

- If the instruction is an assignment $lv = E$ where *lv* is an *l-value* (i.e., either a register or a memory cell) and *E* is an expression then, each register and memory cell used by *E* are replaced by their assigned value in the current state. If some register or memory cell, says *x*, used by *E* is not assigned then a fresh variable *fv* is generated and assigned to *x* in the new state and *x* is replaced by *fv* in *E*. Finally the formula obtained from *E* when all substitutions have been done is assigned to *lv* in the new state.
- If the instruction is a static jump to the address *tgt* and if the jump is guarded by some condition *G*, then the simulator check first if *G* is satisfiable in the current state. To this aim, as for assignments, any occurrence of a register or memory cell is replaced by its current value or a fresh variable in *G*; this gives a formula *G'*. The solver is then used to verify the satiafiability of (*G'* `and` *pc*) where *pc* is the current path-condition. If the

formula is satisfiable then the new path-condition becomes (G' `and` pc) and program counter is updated; else the simulator tries the next microcode instruction at the same address or stops.

- If the instruction is a dynamic jump to an address obtained from an expression E, then, as for others instructions, registers and memory cells are replaced by their value in E. And, the solver is used to compute a valid value `addr` for E under the constraint of the path condition. If this is the case, the constraint (E = `addr`) is added to the new path condition and the program counter is set to `addr`.

2 Analyzed program

The program we are studying using `iii` is a *crackme* challenge, meaning that the program implements a few security mechanisms to protect some critical parts of the software and we have to bypass these to access the protected data.

The binary code has been obtained by compiling a small `x86` assembly program using `fasm` assembler ([FASM], page 27). The source code of this challenge is given in Appendix A [Crackme source code], page 28.

As shown below, the behavior of this program is simple: It displays a prompt, reads on the standard input a password and spawns a `/bin/sh` shell program if the password is correct or exits in the other case.

```
$ ./crackme
Enter password:
toto
Wrong password
$ ./crackme
Enter password:
Iv6oCb2U
sh-4.2$
```

Our challenge is to discover, from the binary code, the password that permits the execution of the shell program. In order to counter analysis of the binary file, the program implements fences to protect itself from reverse-engineering:

1. The password is not stored as-is in the binary data but is hashed.
2. The algorithm that computes the hash-value of the input is cyphered and this part of the binary is uncyphered on-the-fly at execution time.

The last point imposes to change the read/write/execute flags of the `.text` section of the executable to allow self-modifying code (see `elf (5)`).

3 Basic Features

3.1 pynsight interpreter

pynsight is Python interpreter extended with Insight bindings. iii is a debugger built on the top of pynsight. As shown below, when iii is started a banner is displayed and the tool presents a prompt that permits to interact with pynsight:

```
$ iii
iii
Insight Interactive Inspector
Try 'help(insight.debugger)' to get information on debugger commands.
Type 'aliases()' to display list of defined aliases.

No module named iiirc

iii>
```

The interpreter indicates that iiirc module has not been found. This point will be clarified in Section 4.2 [Initialization file], page 14.

After the prompt, iii>, any Python script can be executed:

```
iii> for i in range(5):
...     print 2*i
...
0
2
4
6
8
iii>
```

In the context of iii, some modules are pre-loaded. The most interesting one, insight.debugger, contains all functions proposed by iii. Since we are in a Python interpreter, documentation related to Insight modules can be displayed using the help function; as suggested by the banner try 'help(insight.debugger)' to discover all functions implemented in iii.

Since iii is an interactive tool, several shorcuts have been defined for most frequently used commands e.g., run, step, ... The function aliases lists these shortcuts:

```
iii> aliases()
ms          -> microstep
P           -> prog
cond        -> cond
ep          -> entrypoint
...
```

Each function should be documented. For instance help(microstep), or equivalently help(ms), describes the behavior of microstep function.

3.2 Getting started

The tool can be started without any argument or with the path to a binary file to analyse. iii accepts several options; the usual --help option lists all others. Let's start iii with our crackme program.

```
$ iii crackme
```

`crackme` has been successfully loaded. Another way to load a binary file is to use the `binfile` function. We can get informations related to what kind of program is currently loaded; to this purpose we use the function `info()`:

```
iii> info()
address_size         : 0x20(32)
memory_min_address   : 0x8048000(134512640)
memory_max_address   : 0x80ec4b3(135185587)
format               : elf32-i386
inputname            : crackme
registers            : 'ac': 1, 'gs': 16, 'af': 1, 'zf': 1, 'edi': 32,
'iopl': 2, 'cf': 1, 'vip': 1, 'ebp': 32, 'cs': 16, 'vif': 1, 'edx': 32,
'ebx': 32, 'id': 1, 'es': 16, 'if': 1, 'esp': 32, 'rf': 1, 'pf': 1,
'tf': 1, 'nt': 1, 'esi': 32, 'fs': 16, 'df': 1, 'vm': 1, 'eax': 32,
'ds': 16, 'ecx': 32, 'ss': 16, 'of': 1, 'sf': 1
entrypoint           : 0x8048c18(134515736)
word_size            : 0x20(32)
endianness           : little
cpu                  : x86-32
iii>
```

Among other informations, `info()` gives the list of registers with their respective size in number of bits.

`iii` is a debugger i.e., it simulates behaviors of analyzed programs according to some domain used to evaluate values. By default, the *symbolic* domain is used; this means that values are formulas. The domain is specified as an argument to the `binfile` function. Currently only two domains are supported formulas and concrete values (see `help(binfile)`).

In `iii`, most of functions are related to simulation. Among them, one cannot be avoided: `run()`. This function starts the simulation of the loaded program. A simulation-related function should fail if it has not been preceded by a call to `run()`. For instance, if we request the execution of one assembler instruction using the `step()` function while the simulation is not started we obtain:

```
iii> step()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "/home/point/LaBRI/Projects/dev/install/linux-x86_64/share/insight/pynsight/in
sight/debugger.py", line 272, in step
    __record(pc(), step, a)
  File "/home/point/LaBRI/Projects/dev/install/linux-x86_64/share/insight/pynsight/in
sight/debugger.py", line 583, in pc
    return mcpc()[0]
  File "/home/point/LaBRI/Projects/dev/install/linux-x86_64/share/insight/pynsight/in
sight/debugger.py", line 594, in mcpc
    return simulator.get_pc()
SimulationNotStartedException
iii>
```

The reader should have noticed that `pynsight` interpreter prints out its call stack. Remember that we are interacting with a `Python` interpreter; `step()` is a simple call to a function which could have been invoked from a script.

3.3 Step-by-step simulation

So, let's start the simulation with `run()`. At this point, if `run()` is used without any argument, the execution will start at the entrypoint of the program (see `ep()`). The function can accept a different start address which is reused later when `run` is invoked a new time without arguments.

After the invocation of `run` the simulator displays the list of enabled microcode arrows:

```
iii> run()
Arrows from (0x8048c18,0):
0 : (0x8048c18,0) %ebp{0;32} := (XOR %ebp{0;32} %ebp{0;32}){0;32} --> (0x8048c18,1)
iii>
```

At any moment one can display currently enabled arrows using the function `arrows`. Above, only one arrow is enabled. This arrow goes from microcode address `(0x8048c18,0)` to `(0x8048c18,1)`. A microcode address has two components. The first one is a *global* address that corresponds to an actual address in the concrete memory of the process (aka. virtual memory). The second one is *local* address used to implement the semantics of instructions. To know what is the current instruction pointed by the current microcode address, use the function `instr()` and `mcpc()` to know what is the current value of the program pointer:

```
iii> instr()
xor    %ebp,%ebp
iii> map(hex, mcpc())
['0x8048c18', '0x0']
iii>
```

This instruction, `xor %ebp,%ebp`, can not be implemented by only one microcode statement; this is why the destination of this first arrow is a *local* microcode address. When the arrow is triggered the register `ebp` is set to 0; actually it receives the exclusive-or of its current value with itself.

The execution of an arrow is requested using `microstep()` (or `ms()`) function:

```
iii> ms()
Arrows from (0x8048c18,1):
0 : (0x8048c18,1) %sf{0;1} := %ebp{31;1} --> (0x8048c18,2)
iii>
```

`iii` displays a new microcode arrow between two local addresses. Actually, following Intel specifications, `xor` instruction computes the exclusive-or of its operands and then, assigns several flags according to the resulting value; here the *sign* flag (`sf`) is computed. The reader should have noticed that flags are implemented using an *ad-hoc* register instead of a window into the actual `eflags` register. This is essentially due to performance reasons.

Let's continue until the end of the `xor` assembly instruction:

```
iii> ms()
Arrows from (0x8048c18,2):
0 : (0x8048c18,2) %zf{0;1} := (EQ %ebp{0;32} 0x0{0;32}){0;1} --> (0x8048c18,3)
iii> ms()
Arrows from (0x8048c18,3):
0 : (0x8048c18,3) %pf{0;1} := (XOR (XOR (XOR (XOR (XOR (XOR (XOR (XOR 0x1{0;1} ...
--> (0x8048c18,4)
iii> ms()
Arrows from (0x8048c18,4):
0 : (0x8048c18,4) %cf{0;1} := 0x0{0;1} --> (0x8048c18,5)
iii> ms()
```

```
Arrows from (0x8048c18,5):
0 : (0x8048c18,5) %of{0;1} := 0x0{0;1} --> (0x8048c1a,0)
iii> ms()
Arrows from (0x8048c1a,0):
0 : (0x8048c1a,0) %esi{0;32} := [%esp{0;32}]{0;32} --> (0x8048c1a,1)
iii> instr()
pop    %esi
iii>
```

For clarity reasons, the expression at microde address `(0x8048c18,3)` has been cutted off because of its length (value of the parity flag `pf`). After a few microsteps, the simulation reach the address `(0x8048c1a,0)` which points to the assembly instruction `pop %esi`.

The simulation of the program at the microde-level is not so interesting. As usual with a debugger, it is preferable to step forward at instruction-level. In this case the function `step()` (or `s()`) must be used. Let's restart the program and execute the first instruction in a single step (i.e., intermediate microcode steps will be hidden):

```
iii> r()
Arrows from (0x8048c18,0):
0 : (0x8048c18,0) %ebp{0;32} := (XOR %ebp{0;32} %ebp{0;32}){0;32} --> (0x8048c18,1)
iii> instr()
xor    %ebp,%ebp
iii> s()
Arrows from (0x8048c1a,0):
0 : (0x8048c1a,0) %esi{0;32} := [%esp{0;32}]{0;32} --> (0x8048c1a,1)
iii> instr()
pop    %esi
iii>
```

`run()`, `microstep()` and `step()` are functions that drive a simulator that interprets the semantics of instructions according to some state of the program. This state is given by:

1. The value of the program counter (i.e. the current microcode address);
2. and some *context* that represents the values stored into the memory and the assignment of registers. This context depends on the domain used to represent values.

In the sequel, we will only use the symbolic domain. The context component of the states is an assignment of memory cells and registers with formulas, and an additional formula, called the *path condition*. This condition is actually a constraint on all variables used in the formulas of the state.

The function `print_state()` can be used to display the current state. Actually it must be used with care because it can print out an huge amount of data. The following example continues the simulation for two more steps and then invokes `print_state()`:

```
iii> s(); s()
Arrows from (0x8048c1b,0):
0 : (0x8048c1b,0) %ecx{0;32} := %esp{0;32} --> (0x8048c1d,0)
Arrows from (0x8048c1d,0):
0 : (0x8048c1d,0) %esp{0;32} := (AND %esp{0;32} 0xfffffff0{0;32}){0;32}
--> (0x8048c1d,1)
iii> print_state()
<(0x8048c1d,0), MemoryDump:
Registers:
[sf{0;1} = 0x0{0;1}]
[of{0;1} = 0x0{0;1}]
[esp{0;32} = (ADD uv_3_0x8048c1a_32b{0;32} 0x4{0;32}){0;32}]
[ecx{0;32} = (ADD uv_3_0x8048c1a_32b{0;32} 0x4{0;32}){0;32}]
```

```
[esi{0;32} = [uv_3_0x8048c1a_32b{0;32}]{0;32}]
[zf{0;1} = 0x1{0;1}]
[cf{0;1} = 0x0{0;1}]
[ebp{0;32} = 0x0{0;32}]
[pf{0;1} = 0x1{0;1}]
condition = 0x1{0;1}
>
iii>
```

The displayed state indicates that the simulator is currently stopped at microcode address `(0x8048c1d,0)`. It also shows that no memory cell has been yet assigned by the program; nothing is displayed after `MemoryDump` message[1]. Up to now, only registers have been assigned. Some have received constant values; for instance, `zf` flag has been set to `0x1{0;1}`. Others registers are assigned with formulas e.g. `ecx` is assigned with the value `(ADD uv_3_0x8048c1a_32b{0;32} 0x4{0;32}){0;32}`. `uv_3_0x8048c1a_32b` is the identifier of a fresh variable created by the simulator. The identifier gives us some informations on its creation context:

- `uv`: This variable has been allocated when an **u***nknown* **v***alue* had to be assigned to a register or a memory cell.
- `3`: This is the third fresh variable created so far.
- `0x8048c1a`: This variable has been created by the instruction at the address `0x8048c1a`
- `32b`: This variable is a bitvector of size 32 bits.

If we have a look to instruction at `0x8048c1a` we obtain:

```
iii> instr(0x8048c1a)
pop    %esi
iii>
```

What is the connection with `ecx` ? Actually, this instruction pops the top of the stack and stores the value into the register `esi`. The top of the stack is pointed out by the register `esp`. When this instruction has been triggered, we has the following context:

- `esp` was not assigned. In order to continue, the simulator assigned to `esp` an unknown value abstracted with a fresh variable: `uv_3_0x8048c1a_32b`.
- Then, the top of the stack can be assigned to `esi`; it is the memory cell located at the address pointed by `esp` i.e. `uv_3_0x8048c1a_32b`. `esi` receives the value/formula: `[uv_3_0x8048c1a_32b{0;32}]{0;32}]`.
- The top of the stack must be removed thus, according to Intel specifications, `esp` is increased to point 4 bytes forward: it is assigned the value `(ADD uv_3_0x8048c1a_32b{0;32} 0x4{0;32}){0;32}`.
- Finally the value of `ecx` comes from the next instruction located at address `(0x8048c1b,0)`: `mov %esp,%ecx`.

Like other debuggers, we can let `iii` run the simulation until we interrupt it or something enforces it to stop. `iii`'s *continue* function, is called `cont()` (or `c()`). The following example restarts the simulation and execute `cont()` just after the `run()` call.

```
iii> run()
Arrows from (0x8048c18,0):
```

[1] This does not mean that the memory is empty! Actually some parts of the memory is already occupied by the loaded sections of the program itself.

```
0 : (0x8048c18,0) %ebp{0;32} := (XOR %ebp{0;32} %ebp{0;32}){0;32} --> (0x8048c18,1)
iii> cont()
execution interrupted:
Undefined value (SUB (AND (ADD uv_13_0x8048c1a_32b{0;32} 0x4{0;32}){0;32}
0xfffffff0{0;32}){0;32} 0x4{0;32}){0;32}
Arrows from (0x8048c20,1):
0 : (0x8048c20,1) [%esp{0;32}]{0;32} := %eax{0;32} --> (0x8048c21,0)
iii>
```

We are back to `iii` prompt with a message from `cont()` indicating that the simulation has been interrupted due to an undefined value. The enabled arrow indicates that we are stopped at microcode address `(0x8048c20,1)`. At this location the program tries to assign the value of register `eax` into the memory cell pointed by register `esp` i.e., it tries to put `eax` on the stack[2]. Here, the problem is that we cannot concretize the value of `esp` because it is an unknown value. We use the function `register()` to get it:

```
iii> register("esp")
'(SUB (AND (ADD uv_13_0x8048c1a_32b{0;32} 0x4{0;32}){0;32} 0xfffffff0{0;32}){0;32}
0x4{0;32}){0;32}'
iii>
```

`esp` is assigned a formula that depends on the variable `uv_13_0x8048c1a_32b`. Thanks to the embedded SMT solver of Insight, the simulator guesses that `uv_13_0x8048c1a_32b` can take many values, but concretization requires that value to be unique in order to be translated into a memory address, thus `iii` can not determine the memory cell to assign.

The behavior of the simulator is not surprising since `esp` is not initialized[3]. In order to assign explicitly a value to a register or a memory cell we use the function `set()`. Let's try to assign `0x12345678` to `esp`:

```
iii> set("esp", 0x12345678)
try to assign an inconsistent value to esp
iii>
```

`iii` replies that `0x12345678` is an inconsistent value in the current context. Meaning that `0x12345678` is not compatible with the current formula assigned to `esp`, `(SUB (AND (ADD uv_13_0x8048c1a_32b{0;32} 0x4{0;32}){0;32} 0xfffffff0{0;32}){0;32} 0x4{0;32}){0;32}`, under the constraint of the current path condition (which is true here). A consistent value in this case is `0xfffffffc`.

```
iii> set("esp", 0xfffffffc)
iii> register("esp")
'0xfffffffc0;32'
iii>
```

3.4 Hooks

In order to prevent the problem with an unknown `esp`, we should assign it just after the call to `run()`. When the simulator is started, the memory and registers are not initialized; thus, any value can be assigned to `esp`. However, it is preferable to choose a value that have a sense for the program. Usually `0xFFFFFFF0` is a good candidate.

When debugging a program, `run()` is called quite often (for the entrypoint or elsewhere) and initializing `esp` each time becomes a tedious task. Fortunately `iii` possesses a mean to get rid of such repeative work.

[2] Which is confirmed by a call to `instr()`.

[3] This assignment is usually done by the OS which is not described in our model.

`iii` permits to attach callbacks to simulation functions (i.e., `run()`, `microstep()`, `step()` and `cont()`). Such callbacks are called *hooks*.

A *hook* is a `Python` function invoked with no argument; a function with default values assigned to all its parameters can be used as a hook. In order to attach a hook h to a function F, one can used two functions:

1. `add_hook(F, h)`
2. `add_F_hook(h)`

The first version attach a hook to any function used for the simulation (and only them). The list of currently attached hooks can be obtained using `show("hooks")` and a hook can be removed using `del_hook()`.

We can now set the initialization of `esp` as a hook[4]:

```
iii> add_run_hook(lambda: set("esp", 0xFFFFFFF0))
iii> run()
Arrows from (0x8048c18,0):
0 : (0x8048c18,0) %ebp{0;32} := (XOR %ebp{0;32} %ebp{0;32}){0;32} --> (0x8048c18,1)
iii> cont()
stop in a configuration with several output arrows
Arrows from (0x8048ed7,0):
0 : (0x8048ed7,0) << (NOT %zf{0;1}){0;1} >> Skip --> (0x8048ed0,0)
1 : (0x8048ed7,0) << %zf{0;1} >> Skip --> (0x8048ed9,0)
iii> instr()
jne    0x8048ed0
iii>
```

The call to `cont()` led the simulator farther. Now the simulation display a choice between two enabled arrows. Actually, as shown by `instr()`, we face a conditional jump instruction.

`iii` displays both arrows of the conditional jumps because both guards `(NOT %zf{0;1}){0;1}` and `%zf{0;1}` can be satisfied (obviously not by the same assignment of variables):

```
iii> register("zf")
'(EQ [(ADD (ADD 0xffffffff4{0;32} (MUL_U uv_8_0x8048c1a_32b{0;32} 0x4{0;32}){0;32}){0;
32} 0x4{0;32}){0;32}]{0;32} 0x0{0;32}){0;1}'
iii>
```

In order to continue the simulation, we have to follow one of the two arrows. Indeed, `microstep()`, `step()` and `cont()` accept a parameter. By default this parameter is set to `0`. But, in fact, this parameter is the index of the arrow to trigger. For instance, if we want to continue the simulation assuming `zf` set to `1` we call `cont(1)`. For the moment we just make a single step and have a look to the simulation state:

```
iii> s(1)
Arrows from (0x8048ed9,0):
0 : (0x8048ed9,0) %esp{0;32} := (SUB %esp{0;32} 0x4{0;32}){0;32} --> (0x8048ed9,1)
iii> print_state()
<(0x8048ed9,0), MemoryDump:
...
...
condition = (EQ [(ADD (ADD 0xffffffff4{0;32} (MUL_U uv_4_0x8048c1a_32b{0;32} 0x4{0;32})
{0;32}){0;32} 0x4{0;32}){0;32}]{0;32} 0x0{0;32}){0;1}
>
iii>
```

[4] Note that the '`lambda:`' construction is part of the standard `Python` language.

The reader can notice that the value of `zf` has been assigned to the path condition of the state. Each time we enforce the simulator to follow an arrow with a guard, this guard is conjuncted to the path condition of the current state.

Now, we continue a little bit more with `cont()`.

```
iii> c()
stop in a configuration with several output arrows
Arrows from (0x8057c89,0):
0 : (0x8057c89,0) << %zf{0;1} >> Skip --> (0x8057dce,0)
1 : (0x8057c89,0) << (NOT %zf{0;1}){0;1} >> Skip --> (0x8057c8f,0)
iii>
```

Since we are yet stopped by a conditional jump, we could take a while to have a look to the code of the program. This can be done using the `disas()` function. This function accepts several parameters. The first one is a start address from which the function have to display instructions; if it is omitted the whole program is displayed. A second one is *l*, the number of instructions to be displayed (set by default to 20). Let's have a look to the ten first instructions from the entrypoint:

```
iii> disas(entrypoint(),l=10)
08048c18 <_start>:
 8048c18:  xor    %ebp,%ebp
 8048c1a:  pop    %esi
 8048c1b:  mov    %esp,%ecx
 8048c1d:  and    $0xfffffff0,%esp
 8048c20:  push   %eax
 8048c21:  push   %esp
 8048c22:  push   %edx
 8048c23:  push   $0x80494c0
 8048c28:  push   $0x8049420
 8048c2d:  push   %ecx
 8048c2e:
iii>
```

Note that these instructions are those collected from the microcode built during the simulation and not the output of a direct linear-sweep on the binary code.

3.5 Graphical views

A graphical view of the CFG of assembly instructions can be diplayed by the `view_asm()` function which is based on the `GraphViz` tool ([DOT], page 27). When invoked, the function `view_asm()` opens an `XDot` ([XDot], page 27) window while `iii` remains active:

```
iii> view_asm()
iii> arrows()
Arrows from (0x8057c89,0):
0 : (0x8057c89,0) << %zf0;1 >> Skip --> (0x8057dce,0)
1 : (0x8057c89,0) << (NOT %zf0;1)0;1 >> Skip --> (0x8057c8f,0)
iii>
```

Figure 3.1 shows an example of the graphical representation of the CFG. Each block has a distinct color. If Insight succeeds to extract symbols, `view_asm` displays them on the graph (as `_dl_aux_init` on the figure). The current program point is marked with a double-surrounding red line. Some nodes appear with an oval shape; they correspond to pending arrows for unexplored program points. Edges can be labeled with a number that is the index of the arrow printed out by `iii` (by using `arrows()` function for instance).

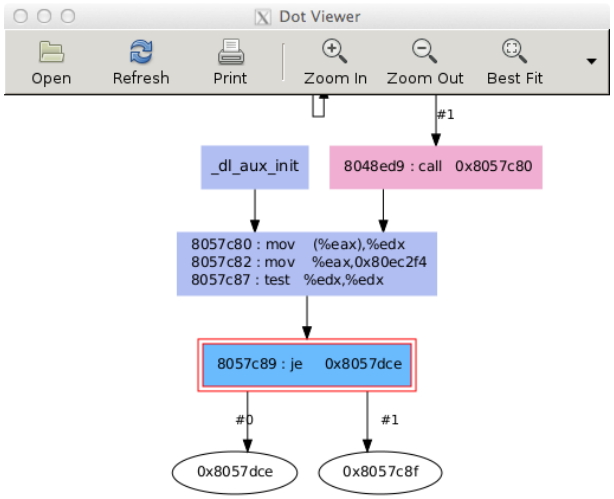


Figure 3.1: Graphical view of the CFG.

The `view_mc()` function displays also a graphical view of the CFG but at a microcode level. For example, Figure 3.2 shows the exact same program as in Figure 3.1 but from a microcode perspective.

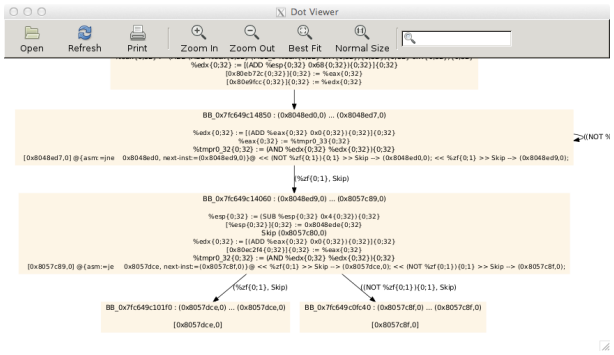


Figure 3.2: Graphical view of the Microcode.

`view_asm`, as well as `view_mc`, can be used as hooks. The following example attaches `view_asm` to all simulation functions. If the simulation is pursued then the graphical view of the CFG is automatically updated.

```
iii> for f in [ cont, step, run]: add_hook(f, view_asm)
iii> cont()
...
```

4 Advanced features

4.1 Using stubs

Still on our running example, we can notice (using the `view_asm()` or `disas()` functions) that several symbols are discovered: `__libc_start_main`, `_dl_aux_init`, `_dl_discover_osversion`, ... It seems that simulation has entered inside initialization procedure done by the `libc` library. This part of the program should not be actually connected with the behaviors of the `crackme` challenge.

`iii` offers a mechanism to bypass parts of the program. It is possible to preload a microcode model at a specified address. These preloaded models are called *stubs*.

In order to skip the initialization work done by `__libc_start_main`, we have to:

- Abstract the behaviors of `__libc_start_main` into a microcode model;
- Create a loadable (i.e., `xml`) file for this model;
- Attach this model to the address of `__libc_start_main`.

The disassembly of program from the entrypoint (`_start`) shows that the address `0x8048da9` is pushed onto the stack just before the call to `__libc_start_main`. This address points to `main` function[1]. When `__libc_start_main` has finished its jobs, it jumps to the address pushed by `_start` onto the stack.

```
iii> disas (entrypoint())
08048c18 <_start>:
 8048c18:  xor    %ebp,%ebp
 8048c1a:  pop    %esi
 8048c1b:  mov    %esp,%ecx
 8048c1d:  and    $0xfffffff0,%esp
 8048c20:  push   %eax
 8048c21:  push   %esp
 8048c22:  push   %edx
 8048c23:  push   $0x80494c0
 8048c28:  push   $0x8049420
 8048c2d:  push   %ecx
 8048c2e:  push   %esi
 8048c2f:  push   $0x8048da9
 8048c34:  call   0x8048e80 # jump to : __libc_start_main
08048e80 <__libc_start_main>:
 8048e80:  push   %edi
 8048e81:  mov    $0x80ea5c0,%eax
 8048e86:  push   %esi
 8048e87:  push   %ebx
 8048e88:  sub    $0x40,%esp
 8048e8b:  test   %eax,%eax
 8048e8d:  mov    0x5c(%esp),%esi
 8048e91:
iii>
```

Since initialization done by `__libc_start_main` does not matter for our analysis, we will abstract its behaviors as a direct jump to the `main` function. For this purpose we could

[1] This information can be obtained (if available) using the table of symbols attached to the program; try `help(prog().sym)`.

create a microcode model that jump to the address `0x8048da9` but this is not re-usable at all; we should prefer to jump to the address stored in the stack.

Microcode files are XML files. Even if the model for `__libc_start_main` is quite simple, writing it by hands is a tedious task. The simplest way to proceed is to:

1. Write the abstraction into a small `x86` assembly program:
   ```
   $ cat stub_libc_start_main.s
           jmp *4(%esp)
   $
   ```
2. Compile it using `gcc` for instance;
   ```
   $ gcc -m32 -c stub_libc_start_main.s -o stub_libc_start_main.o
   ```
3. Generate the microcode file using the `cfgrecovery` tool. A simple linear sweep disassembly is sufficient to generate the exact microcode for this small program
   ```
   $ cfgrecovery -f mc-xml -d linear stub_libc_start_main.o -o \
     stub_libc_start_main.mc.xml
   ```

Now we have the abstraction for `__libc_start_main` stored into the file `stub_libc_start_main.mc.xml`; it remains to load it at the address pointed by the `__libc_start_main` symbol. The function `load_stub()` is used for this purpose. All stubs should be loaded before the first call to `run()`. Actually stubs are merged into the microcode on demand and not as a replacement of existing microcode. `load_stub()` takes three arguments: a filename, an address and a Boolean that indicates whether the microcode must be relocated at the same microcode address or not.

4.2 Initialization file

Since we already have recovered the first instruction of `__libc_start_main`, the loading of the stub is useless; we must restart `iii` to make the stub effective.

If we restart `iii`, we have to redo all the work we have done so far; and it will be the case each time we will restart the tool. Fortunately, `iii` permits to specify an initialization module using the `-c` option. When no module is specified, the interpreter looks for a module called `iiirc`. This file must be a `Python` script that contain calls to `iii` functions.

The initialization file must first import Insight/`iii` functions into the script (`insight.debugger` and `insight.iii`). Then, comes a line that indicates the binary file to load (see `help(binfile)` for details). And, then, it follows all our previous work:

1. First the stub for `__libc_start_main` is loaded;
2. Then we define a function, `init_registers`, in charge of the initialization of registers according to a global table *valregs*.
3. Hooks are then attached: `init_registers` is attached to `run` and `view_asm` is attached to simulation functions.
4. Finally the simulation is started.
   ```
   $ cat iiirc.py
   # mandatory import to be able to use Insight functions
   from insight.debugger import *
   from insight.iii import *
   ```

```
# we load the binary file
binfile ("crackme", target="elf32-i386", domain="symbolic")

# load the stub replacing __libc_start_main
load_stub ("stub_libc_start_main.mc.xml", P ().sym ("__libc_start_main"))

# initialization of register
valregs = {
    'esp' :  0xFFFFFFF0,
    'df' :  0  # mandatory for string operations
}

# useful hooks
def init_registers ():
    global valregs
    for r in P().info()['registers']:
        if r in valregs:
          val = valregs[r]
          set(r, val)

add_hook (run, init_registers)

add_hook (cont, view_asm)
add_hook (run, view_asm)
add_hook (step, view_asm)

# start simulation from entrypoint
run ()
cont()
```

Starting `iii` with the above initialization module will produce the following output:

```
$ iii
Insight Interactive Inspector
Try 'help(insight.debugger)' to get information on debugger commands.
Type 'aliases()' to display list of defined aliases.

Arrows from (0x8048c18,0):
0 : (0x8048c18,0) %ebp{0;32} := (XOR %ebp{0;32} %ebp{0;32}){0;32} --> (0x8048c18,1)
stop in a configuration with several output arrows
Arrows from (0x805ad7a,0):
0 : (0x805ad7a,0) << %zf{0;1} >> Skip --> (0x805ad9b,0)
1 : (0x805ad7a,0) << (NOT %zf{0;1}){0;1} >> Skip --> (0x805ad7c,0)

iii>
```

And, thanks to the `view_asm` window, we can see on Figure 4.1 that the simulation has gone yet farther. Indeed, the actual code of `__lib_start_main` has been skipped thanks to its attached stub. However, `iii` is now simulating another function of the standard library: `printf`. This function will be replaced by the following stub:

```
mov $0x0, %eax
ret
```

We assume that it has no interesting side effect and enforce its return value to 0 which is stored to `eax`. Another standard function should appear later, `read`, which is called to read the password. The stub for this function is the following:

```
mov 12(%esp), %eax
mov %eax, %ecx
```

```
          mov 8(%esp), %ebx
label:    movb $0x33, (%ebx)
          inc %ebx
          dec %ecx
          jnz label
          ret
```

This stub simulates the read of n characters stored into a memory area at address p. n and p are themself stored onto the stack at `12(%esp)` and `8(%esp)`, respectively. Then the stub assigns the n memory cells with the value `0x33`. This special value is used to locate the assignment into the microcode file. This value is replaced by hand (or a `sed` script) by a `RND` expression that will be interpreted as an unknown value by the symbolic simulator.

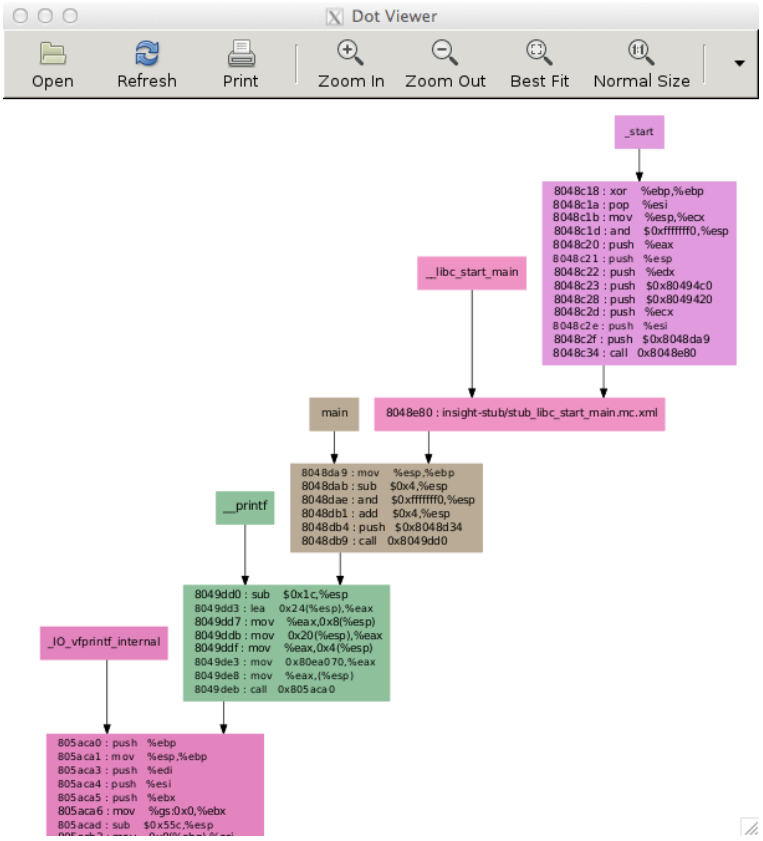
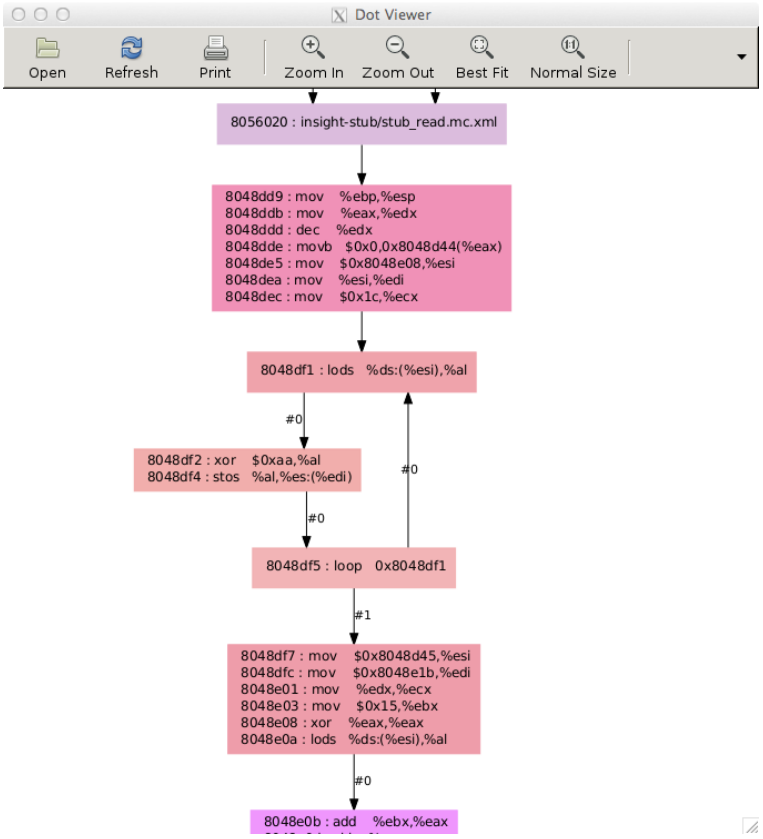


Figure 4.1: Graphical view of the Microcode. `__lib_start_main` has been replaced by its stub but, `__printf` is now decoded.

We add the new `load_stub` calls into the initialization file and restart the tool. The simulation stops on a `jne` instruction. The CFG is the following:



After the call to `read` i.e., the user has entered its password, the program enters into a loop. The small piece of code that precedes the loop does the following things:

- At address `0x8048ddb`, the return value of `read` is copied from `eax` into `edx`. `edx` is then decremented and a null character is stored at the address `0x8048d44+eax`. Indeed, the last byte read from the standard input is a carriage-return character. At address `0x8048dde`, it is replaced by `0` in order to form a valid null-terminated C string. `edx` store the length of the password and must not take into account the last null byte.
- From addresses `0x8048de5` to `0x8048dec`, registers `esi`, `edi` and `ecx` are prepared for some string operation. `esi` and `edi` are the source and destination pointers of the operation and `ecx` the number of iterations. We can see that, `esi` and `edi` have the same starting value (`0x8048e08`).

According to the value assigned to `ecx`, the loop is iterated 28 times. At each iteration of the loop, the byte pointed by `esi` is XOR-ed with `0xaa`. The loop covers addresses between `0x8048e08` and `0x8048e24` which are located few bytes after the loop itsef. This means that this loop is used to modify instructions just after the loop; it is a known trick to obfuscate programs and prevent static-analysis of it.

We can compare on-the-fly decoded instructions by the simulator and the ones that were loaded at start-up using functions `disas` (left) and `insight.utils.pretty_disas_memory`[2] (right). The latter uses a *linear sweep* algorithm on the original binary file raw bytes.

[2] `prog()` returns an opaque object that contains the loaded binary file.

```
iii> disas (0x8048e08, l=8)
 8048e08:  xor   %eax,%eax
 8048e0a:  lods  %ds:(%esi),%al
 8048e0b:  add   %ebx,%eax
 8048e0d:  shl   %eax
 8048e0f:  xor   $0x12,%eax
 8048e12:  mov   %al,%bl
 8048e14:  scas  %es:(%edi),%al
 8048e15:  jne   0x8048e3f # jump to L_1
 8048e17:
```

```
iii> insight.utils.pretty_disas_memory
... (prog(), 0x8048e08, l=8)
 8048e08 :  fwait
 8048e09 :  push  $0x6
 8048e0b :  stos  %eax,%es:(%edi)
 8048e0c :  jb    0x8048e89
 8048e0e :  dec   %edx
 8048e0f :  sub   %ebx,-0x48(%edx)
 8048e12 :  and   0x4(%ecx),%ch
 8048e15 :  fild  -0x5cbebab8(%edx)
```

4.3 Examining memory

We are still stuck at address `0x8048e15`. `arrows()` tells us what are the enabled arrows and we choose to follow the arrow `0`.

```
iii> arrows()
Arrows from (0x8048e15,0):
0 : (0x8048e15,0) << (NOT %zf{0;1}){0;1} >> Skip --> (0x8048e3f,0)
1 : (0x8048e15,0) << %zf{0;1} >> Skip --> (0x8048e17,0)

iii> cont(0)
sink node reached after(0x8048e59, 2)
Arrows from (0x8048c39,0):
iii>
```

The tool indicates that the simulator has reached a sink node i.e., no successor state can be visited. If we have a look at the code visited after the conditional jump at address `0x8048e15` we obtain:

```
iii> disas(0x8048e3f,10)
08048e3f <L_2>:
 8048e3f:  mov    %esp,%ebp
 8048e41:  sub    $0x4,%esp
 8048e44:  and    $0xfffffff0,%esp
 8048e47:  add    $0x4,%esp
 8048e4a:  push   $0x8048e62
 8048e4f:  call   0x8049dd0 # jump to : printf, __printf, _IO_printf
08048e54 <L_4>:
 8048e54:  mov    %ebp,%esp
 8048e56:  xor    %eax,%eax
 8048e58:  inc    %eax
 8048e59:  ret     # jump to : L_0
 8048e5a:
```

The program prints out (using `printf`) a string stored at address `0x8048e62` and then returns (to the termination instruction `hlt`). `dump` function is used to display values stored in the memory for the current state. Let's try to dump it directly from the memory:

```
iii> dump(0x8048e62, l = 10)
0x57{0;8}
0x72{0;8}
0x6f{0;8}
0x6e{0;8}
0x67{0;8}
0x20{0;8}
0x70{0;8}
0x61{0;8}
0x73{0;8}
```

```
0x73{0;8}
iii>
```

We get 10 values. Actually these are *symbolic* values; even if they are concrete. Here we were lucky to get constants and no abstract values. `dump()` accepts an additional parameter that is a callback called to transform the value returned by the simulator. The following `Python` script defines a function that we will use later to translate abstract constants into printable characters. This code is added to the configuration file.

```
import re

def filter_abstract_byte (val):
    """translate a concrete "abstract" value into a character"""
    p = re.compile('^(0x[0-9A-Fa-f]1,2)\.*\$')
    m = p.match (val)
    if m is not None:
        return chr(int(m.group(1),16))
    else:
        return val
```

If we call `dump` with `filter_abstract_byte` we get:

```
iii> dump(0x8048e62, l = 10, filter=filter_abstract_byte)
W
r
o
n
g

p
a
s
s
iii>
```

We can deduce from the content of this string that we have followed the branch where the user enters a wrong password.

4.4 Assignment of abstract values

We restart the simulation and continue until coming back to the conditional jump at address `0x8048e15`. This time we will follow the second branch:

```
iii> r(); c(); c(1)
Arrows from (0x8048c18,0):
0 : (0x8048c18,0) %ebp{0;32} := (XOR %ebp{0;32} %ebp{0;32}){0;32} --> (0x8048c18,1)
stop in a configuration with several output arrows
Arrows from (0x8048e15,0):
0 : (0x8048e15,0) << (NOT %zf{0;1}){0;1} >> Skip --> (0x8048e3f,0)
1 : (0x8048e15,0) << %zf{0;1} >> Skip --> (0x8048e17,0)
stop in a configuration with several output arrows
Arrows from (0x8048e15,0):
0 : (0x8048e15,0) << (NOT %zf{0;1}){0;1} >> Skip --> (0x8048e3f,0)
1 : (0x8048e15,0) << %zf{0;1} >> Skip --> (0x8048e17,0)
iii>
```

We are back to the instruction at `0x8048e15` but this time we have visited a new loop as show on the following CFG:

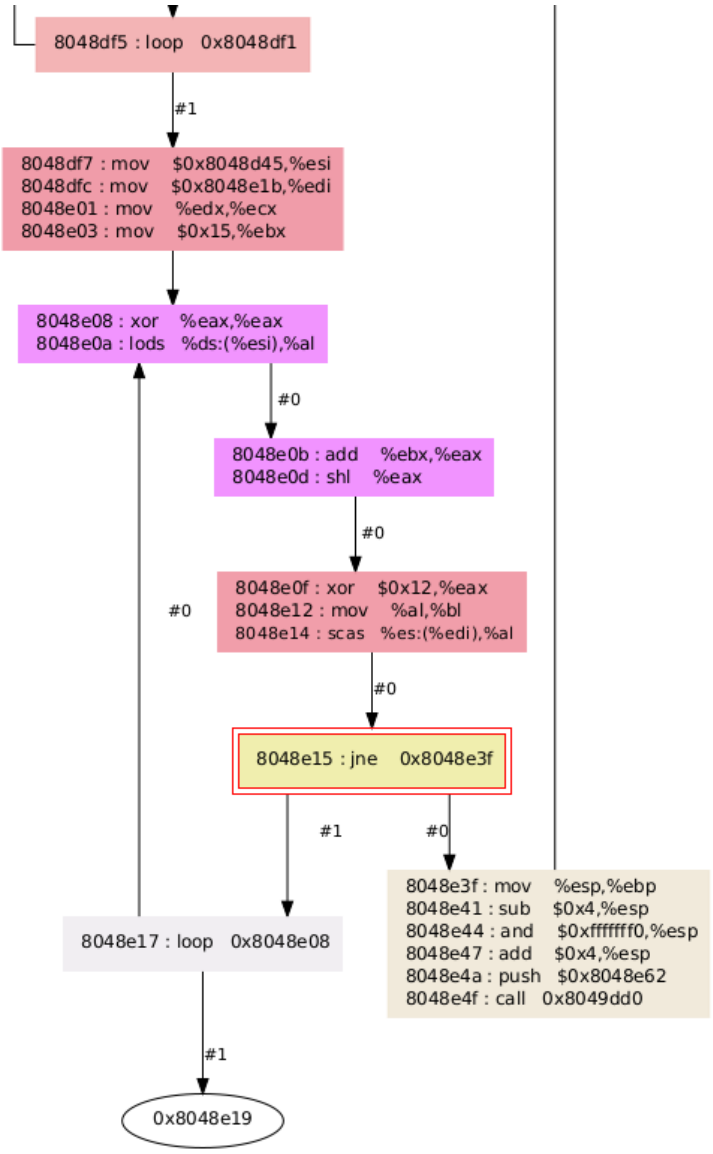


Figure 4.2

The reader should wonder why the simulation did not stop at instruction `0x8048e17` while two outgoing arrows exist there. Indeed, the simulator follows `#0` arrow because it is the only one that is enabled ! Let's see why. We make a single step from `0x8048e15` to `0x8048e17`.

```
iii> arrows()
Arrows from (0x8048e15,0):
0 : (0x8048e15,0) << (NOT %zf{0;1}){0;1} >> Skip --> (0x8048e3f,0)
1 : (0x8048e15,0) << %zf{0;1} >> Skip --> (0x8048e17,0)
iii> s(1)
Arrows from (0x8048e17,0):
```

```
0 : (0x8048e17,0) %ecx{0;32} := (SUB %ecx{0;32} 0x1{0;32}){0;32} --> (0x8048e17,1)
iii> ms()
Arrows from (0x8048e17,1):
0 : (0x8048e17,1) << (NEQ %ecx{0;32} 0x0{0;32}){0;1} >> Skip --> (0x8048e08,0)
iii> register("ecx")
'0x6{0;32}'
iii>
```

The `loop` assembly instruction jumps to `0x8048e08` until `ecx` falls to `0`. However, in the current state, register `ecx` is the constant `6`; thus, the second arrow going out of the `loop` instruction can not be enabled.

We will be back on the loop later. For the moment we want to know what happens when the program follows the second branch. To enable the arrow that moves to address `0x8048e19` we request the simulator to forget the current value of `ecx`. This is done using the function `unset()` (see `help(unset)`).

`unset()` permits to replace the current value of a register or a memory cell with an *unknown value*. In the context of symbolic simulation, this means that a fresh variable is used in place of the current value.

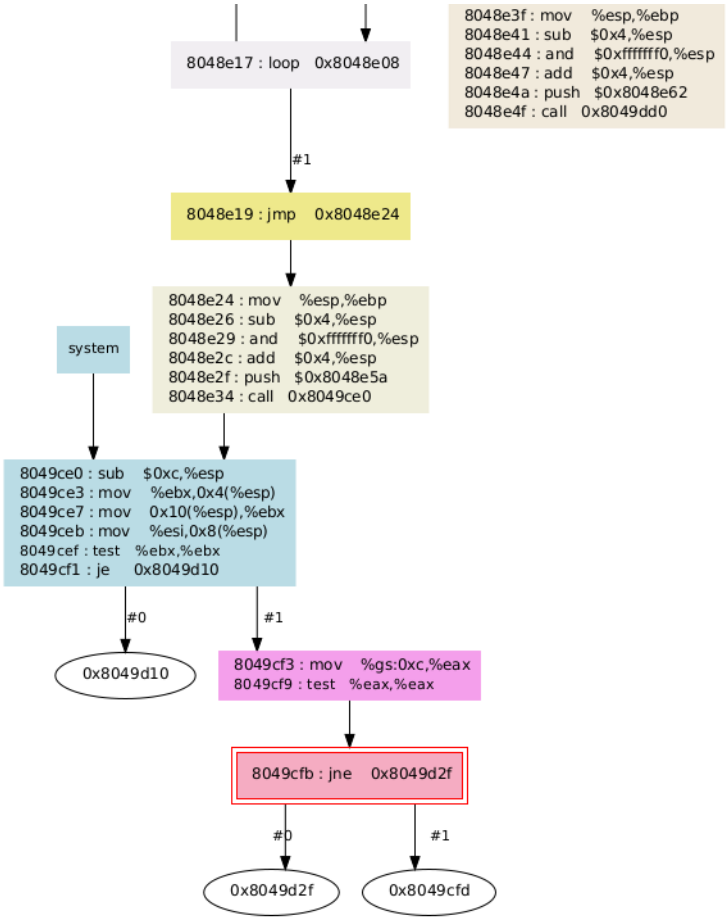
`unset()` takes one, two or three parameters. Only the first one, *loc* is mandatory; it is a register or an address. The second parameter, *len*, is the size of the memory area that is asbtracted; by default *len* is set to 1. The third parameter, *keep*, is a Boolean value that indicates whether or not the new value must be kept consistent with the old one. This notion of consistency varies with the domain used for the simulation. In the case of symbolic simulation this means that:

- if *keep* is `true`, the new state generated by `unset` enforces the fresh variable, say `fv`, to be equal to the current value `val` of the register or memory cell i.e., the constraint `fv=val` is added to the path-condition.
- if *keep* is `false`, the new state generated by `unset` forgets the current value and any value can be assigned to the fresh variable.

keep=`true` is used to maintain the simulation into a real execution path. Using *keep*=`false` means that the simulation is authorized to follow a spurious trace; this is exactly what we want to do there: we want to enforce the trace to follow an arrow that is not possible in the current state.

```
iii> arrows()
Arrows from (0x8048e17,1):
0 : (0x8048e17,1) << (NEQ %ecx{0;32} 0x0{0;32}){0;1} >> Skip --> (0x8048e08,0)
iii> unset("ecx",keep=False)
iii> register("ecx")
'abs_50_0x8048e17_1_32b{0;32}'
iii> arrows()
Arrows from (0x8048e17,1):
0 : (0x8048e17,1) << (NEQ %ecx{0;32} 0x0{0;32}){0;1} >> Skip --> (0x8048e08,0)
1 : (0x8048e17,1) << (NOT (NEQ %ecx{0;32} 0x0{0;32}){0;1}){0;1} >> Skip --> (0x8048e19,0)
iii> cont(1)
```

Now, the value assigned to `ecx` is a fresh symbol. The prefix `abs` is used to indicate that the variable comes from an abstraction required by the user and the address locates the program point where the abstraction has been invoked. The two arrows becomes enabled and we can now follow the arrow to `0x8048e19`.



On the CFG, we can see that, after the loop, the program reaches a call to the `libc` function `system()`. It is invoked with the string located at address `0x8048e5a` as parameter (see instruction at `0x8048e2f`).

```
iii> dump(0x8048e5a,l=10,filter=filter_abstract_byte)
/
b
i
n
/
s
h

W
r
iii>
```

Good news! The call to `system()` spawns a shell `/bin/sh`; this is where we have to go. We, now, get back to the analysis of the loop.

4.5 Breakpoints

In order to study the loop between addresses `0x8048e08` and `0x8048e17` (see Figure 4.2), we execute `r(); c()`. As suggested by instructions at `0x8048df7` and `0x8048dfc`, we are apparently in a loop that compares strings located at addresses `0x8048d45` and `0x8048e1b`. Since the instruction `loop` is used to iterate the comparisons, it means that the length of

the strings is stored into `ecx` by instruction at `0x8048e01` which assigns `edx` to `ecx`. Now remember that `edx` is the length of the input (see remarks related to `read` in Section 4.2 [Initialization file], page 14).

Let's have a look to these strings:

```
iii> register("edx")
0x80;32
iii> dump(0x8048d45, l = 8, filter = filter_abstract_byte)
uv_58_0x8056020_3_8b{0;8}
uv_60_0x8056020_3_8b{0;8}
uv_62_0x8056020_3_8b{0;8}
uv_64_0x8056020_3_8b{0;8}
uv_66_0x8056020_3_8b{0;8}
uv_68_0x8056020_3_8b{0;8}
uv_70_0x8056020_3_8b{0;8}
uv_72_0x8056020_3_8b{0;8}
iii> dump(0x8048e1b, l = 8, filter = filter_abstract_byte)
?
Z
2
P
4
>
?
?
iii>
```

We can deduce that string at `0x8048d45` is the input string (fresh variable has been generated by `__read`). String at `0x8048e1b` is quite cryptic; it should be a hashed value of the password.

Interpreting the code of the loop could be a complicated task. In order to understand it we can try to wait the termination of the loop and have a look to the content of the state.

To this aim we could iterate the loop by hand until instruction `0x8048e17` permits us to go to `0x8048e19` but this a tedious work. A more generic way is the following:

1. We create a conditional breakpoint at `0x8048e17` that will be enabled when `ecx` is equal to `1` (`loop` first decreases `ecx` before checking its value).
2. We add a constraint at `0x8048e15` that enforces the simulator to stay in the loop.

The function `breakpoint()` requests the simulator to stop at a given address. Used without argument, it sets a breakpoint at the current program point. Otherwise, it takes a microcode address (a global and a local address); and the latter is by default set to `0`. The function returns the identifier of the breakpoint; it can be used later by a client script.

```
iii> breakpoint(0x8048e17)
breakpoint set at (0x8048e17,0) with id=1.
1
iii>
```

To make a breakpoint conditional we use the function `cond()`. It can be invoked with one or two arguments. The first one is always an identifier of a breakpoint. The second one is a string that contains the condition to enable the breakpoint. The syntax of these expressions is given elsewhere in Insight documentation. If the second argument is omitted the condition is removed from the breakpoint.

```
iii> cond (1, "(EQ %ecx 1)")
making breakpoint 1  conditional
```

```
1  :  breakpoint: (0x8048e17,0) cond = (EQ %ecx{0;32} 0x1{0;32}){0;1}
iii>
```

The reader should notice that, in expressions, registers are prefixed by a '%' character as in AT&T syntax. The list of breakpoints can be obtained using `show ("breakpoints")`:

```
iii> show ("breakpoints")
1  : hits=0 breakpoint: (0x8048e17,0) cond = (EQ %ecx{0;32} 0x1{0;32}){0;1}
iii>
```

`show("breakpoints")` gives the number of times the simulation did hit the breakpoint.

In order to enforce the simulator to continue inside the loop at instruction `0x8048e15`, we use the function `assume()`. This latter can take up to 3 arguments; the third one is the local component of a microcode address and is, by default, set to `0`. The first argument is the location where the enforcement takes place i.e., an address. The second argument is a string that contains a Boolean expression (same syntax as for conditional breakpoints).

In the context of symbolic simulation, the `assume()` adds constraints on the path condition with its second argument. The list of assumptions can be displayed using the function `show("assumptions")`:

```
iii> assume(0x8048e15, "%zf")
iii> show("assumptions")
0x8048e15 : %zf{0;1}
iii> cont()
```

After the assumption at `0x8048e15`, we let the simulator does its work and wait for the termination of the loop.

4.6 Concretization

After a while we get:

```
iii> cont()
stop condition 1 reached: breakpoint: (0x8048e17,0) cond = (EQ %ecx{0;32} 0x1{0;32})
{0;1}
Arrows from (0x8048e17,0):
0 : (0x8048e17,0) %ecx{0;32} := (SUB %ecx{0;32} 0x1{0;32}){0;32} --> (0x8048e17,1)
iii> s()
Arrows from (0x8048e19,0):
0 : (0x8048e19,0) Skip --> (0x8048e24,0)
iii>
```

The simulator says that it stops because it encounters the conditional breakpoint at address `0x8048e17` while its condition enables it. Contrary to previous sections, this time, if we make a step forward the simulator exits the loop. Now, we have to look at what has been computed.

In Section 4.5 [Breakpoints], page 22, we have discovered that the string at address `0x8048d45` is the input given by the user. If we look at the content of the string we can notice that it does not change.

```
iii> dump(0x8048d45, l = 8, filter = filter_abstract_byte)
uv_58_0x8056020_3_8b{0;8}
uv_60_0x8056020_3_8b{0;8}
uv_62_0x8056020_3_8b{0;8}
uv_64_0x8056020_3_8b{0;8}
uv_66_0x8056020_3_8b{0;8}
uv_68_0x8056020_3_8b{0;8}
uv_70_0x8056020_3_8b{0;8}
```

```
uv_72_0x8056020_3_8b{0;8}
```

A quick look at the iterated code shows that it just compute a value that is then compared at the corresponding offset in the hashed password located at `0x8048e1b`.

```
iii> disas(0x8048e08,l=10)
08048e08 <L_3>:
 8048e08:  xor    %eax,%eax
 8048e0a:  lods   %ds:(%esi),%al
 8048e0b:  add    %ebx,%eax
 8048e0d:  shl    %eax
 8048e0f:  xor    $0x12,%eax
 8048e12:  mov    %al,%bl
 8048e14:  scas   %es:(%edi),%al
 8048e15:  jne    0x8048e3f # jump to : L_2
 8048e17:  loop   0x8048e08 # jump to : L_3
 8048e19:  jmp    0x8048e24 # jump to : L_4
 8048e1b:
iii>
```

Actually the data related to computed values is stored in the path condition accumulated each time we enforced the simulator to follow the arrow `#1` to stay in the loop.

Now, we are outside the loop. This means that the current state of the simulation encodes all traces that can reach the current program point (i.e., `0x8048e19`). Thanks to the SMT-solver integrated to Insight, it is possible to compute an assignment of fresh variables (i.e., a concrete input) that satisfies the path-condition of the current state.

Here, we reuse a function already encountered: `set()`. In Section 3.3 [Step-by-step simulation], page 6, this section was used to assign a value to register `esp`; the function was called with two arguments. This time, we use `set` without specifying a value. In this case, `iii` requests the SMT solver to pick-up a value that is consistent with the current state.

The following script calls `set` for each byte of the input string. Then `iii` displays values computed by the solver.

```
iii> for i in range(8): set (0x8048d45 + i)
...
iii> dump(0x8048d45, l = 8, filter = filter_abstract_byte)
I
v
6
o
C
b
2
U
iii>
```

It is the password; see Chapter 2 [Analyzed program], page 3.

5 Acknowledgements

The crackme has been originally written by Renaud Tabary, a former member of the Insight team and improved by Gérald Point. This tutorial has been first presented at Dagstuhl seminar 14241, *Challenges in Analysing Executables: Scalability, Self-Modifying Code and Synergy* in June 2014, at Dagstuhl Schloss, Germany. We would like to thanks all the attenders for their precious feedback and comments on this tutorial, but also all the people that sent us their feedback.

6 References

[DOT] Graphviz - Graph Visualization Software. http://www.graphviz.org/, 2009.

[FASM] Tomasz Grysztar. Flat Assembler. http://flatassembler.net/, 2014.

[XDot] Jose Fonseca. https://github.com/jrfonseca/xdot.py, 2014.

[JK76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, volume 19, number 7, 385–394, 1976.

[Insight] Insight Framework. http://insight.labri.fr/, 2014.

Appendix A Crackme source code

Here is the source code of the `crackme` program. It is written using `fasm` ([FASM], page 27) assembly language.

```
format ELF

;====================================== DATA ================================

include 'ccall.inc'


macro crypt dstart,dsize {
          local a
          repeat dsize
                    load a from dstart+%-1
                    a = a xor $AA
                    store a at dstart+%-1
          end repeat
}



;====================================== CODE ================================
section '.text'  executable writeable


public main
extrn printf
extrn system
extrn read
extrn strcmp

msg db    "Enter password:",0xA,0
buffer db 100 dup(0)


main:
          ;pwd = Iv6oCb2U


    ccall printf, msg
          ccall read,0,buffer,9
          mov edx,eax
          dec edx
          mov [buffer+eax-1],byte 0

          mov esi,debut_crypt
          mov edi,esi
          mov ecx,to_crypt
decrypt:
          lodsb               ; obfuscation par chiffrement de code
          xor al,0xAA
          stosb
          loop decrypt


          mov esi,buffer
```

```
          mov edi,pwd
          mov ecx,edx
          mov ebx, 0x0015
debut_crypt:
teste:
          xor eax, eax
          lodsb
          add eax, ebx
          shl eax, 1
          xor eax, 0x12
          mov bl, al
          scasb
          jnz ko
          loop teste
          jmp ok
pwd db 174, 90, 50, 80, 52, 62, 242, 156, 0
to_crypt=$-debut_crypt
crypt debut_crypt,$-debut_crypt

ok:
          ccall system,shell
          xor eax,eax
          jmp fin
ko:
          ccall printf,msg2
          xor eax,eax
          inc eax
fin:
          ret


shell db "/bin/sh",0
msg2 db "Wrong password",0xA,0
```

The example includes the following file containing the `ccall` macro:

```
macro ccall proc,[arg]
  { common
     local size
     size = 0
     mov ebp,esp
     if ~ arg eq
    forward
     size = size + 4
    common
     sub esp,size
     end if
     and esp,-16
     if ~ arg eq
     add esp,size
    reverse
     pushd arg
    common
     end if
     call proc
     mov esp,ebp }
```

Appendix B Stubs

Stubs have to be compiled into an object file (using `gcc` for instance) and then their microcode is generated using `cfgrecovery`.

B.1 __libc_start_main

```
jmp *4(%esp)
```

B.2 __printf

```
mov $0x0, %eax
ret
```

B.3 __read

```
mov 12(%esp), %eax
mov %eax, %ecx
mov 8(%esp), %ebx
label:   movb $0x33, (%ebx)
inc %ebx
dec %ecx
jnz label
ret
```

For this stub, the constant `0x33` (`51`) has to be replaced by a random expression into the generated XML file. To this aim you can use the following `sed` commands:

```
sed -e 's+<const size="8" offset="0">51</const>+<random size="8" \
offset="0"></random>+g' stub_read.mc.xml
```

Appendix C Script for automatic password recovery

The following `Python` script can be used as an initialization file (see Section 4.2 [Initialization file], page 14 to automatically recover the password hidden in the `crackme` file.

```
# mandatory import to be able to use Insight functions
from insight.debugger import *
from insight.iii import *

# we load the binary file
binfile ("crackme", target="elf32-i386", domain="symbolic")

# load the stub replacing __libc_start_main
load_stub ("stub_libc_start_main.mc.xml", P ().sym ("__libc_start_main"), True)
load_stub ("stub_printf.mc.xml", P ().sym ("__printf"), True)
load_stub ("stub_read.mc.xml", P ().sym ("__read"), True)

# useful hooks
def init_registers ():
    valregs = {
        'esp' :  0xFFFFFFF0,
        'df' :  0  # mandatory for string operations
    }
    for r in P().info()['registers']:
        if r in valregs:
          val = valregs[r]
          set(r, val)

# filter functions
import re

def filter_abstract_byte (val):
    """translate a concrete "abstract" value into a character"""
    p = re.compile('^(0x[0-9A-Fa-f]{1,2})\{.*\}$')
    m = p.match (val)
    if m is not None:
        return chr(int(m.group(1),16))
    else:
        return val

# setting hooks
add_hook (run, init_registers)
for f in [cont, run, step]:  add_hook(f, view_asm)

# conditional breakpoint to stop just after the loop that checks
# if the password given by the user is correct.
breakpoint(0x8048e17)
cond(1, "(EQ %ecx 1)")

# start the simulator
run()

# assumption must be introduced after the simulation is started
assume(0x8048e15, "%zf")

# start the computation until the breakpoint is reached.
cont()
# go out of the loop
```

```
step()
# let the SMT solver gives us valid input character
for i in range(8):  set(0x8048d45+i)
# display the password
dump(0x8048d45, l = 8, filter = filter_abstract_byte)
```