My gcov

Version 2
8 December 2014

Epita systems/security laboratory 2014 <recrutement@lse.epita.fr>

Copyright

Contents

Obligations

- ▷ Read the *entire* subject
- ▷ Follow the rules
- ▷ Respect the submission

Submission

Project managers:	Jean-Loup BOGALHO < clippix@lse.epita.fr> Louis FEUVRIER < manny@lse.epita.fr> Jérémy LEFAURE < blatinox@lse.epita.fr>
Project markup:	**[GCOV]**
Developers:	1
Submission method:	Git ACU
Submission deadline:	21 December 2014 at 11h42
Project duration:	2 weeks
Newsgroup:	iit.labos.lse.recrutement
Architecture/OS :	x86_64 Linux
Language(s):	C++
Compiler:	**g++ version 4.9.1**
Compiler options:	**-Wall -Wextra -Werror**
Allowed includes:	All

Instructions

The following instructions are fundamental:

Respect them, otherwise your mark may be multiplied by 0.

They are clear, non-ambiguous and each of them has a precise objective.

They are non-negotiable and you must follow them carefully.

Do not dither to ask if you do not understand any of the rules.

- **Instruction 0:** A file **`AUTHORS`** must be present in the root directory of your submission. This file must contain the login of each member of your group, leader first, in the following format : a star **`*`**, a space , then your login (ex: login_x) followed by a newline.

  Example:

  ```
  42sh$ cat -e AUTHORS
  * login_x$
  42sh$
  ```

  If this file is not present you will get a non-negotiable mark of 0.

- **Instruction 1:** Respect carefully the output samples' format.
  Examples are indented: **`$`** symbolizes the prompt, use it as a landmark.

- **Instruction 2:** You have to **automagically** produce a source code using the coding style when it exists for the language you are using. In any case, your code must be clean, readable, never exceeding 80 columns.

- **Instruction 3:** File and directory permissions are mandatory and stand for the whole project: main directory, main directory's files, subdirectories, subdirectories' files, etc.

- **Instruction 4:** When an executable is requested, you must only provide its source code. It will be compiled by us.

- **Instruction 5:** If a file named **`configure`** is present in the root directory of your submission, it will be executed before processing the compilation. This file must have the execution permissions.

- **Instruction 6:** Your submission must respect the following directory tree, where **`login_x`** has to be replaced by the login of the leader of your group :

  ```
  login_x-my_gcov/AUTHORS *
  login_x-my_gcov/README *
  login_x-my_gcov/TODO *
  login_x-my_gcov/src/ *
  ```

  The following files are mandatory:

`TODO`	describes the tasks to complete. Need to be updated regularly.
`AUTHORS`	contains the authors of the project. Must be EPITA style compliant.
`README`	describes the project and the used algorithms in a proper english. Also explains how to use your project.

  Your test-suite will be executed during your oral examination (if any) by an assistant.

▷ **Instruction 7**

The root directory of your submission must contain a file named **`Makefile`** with the following **mandatory** rules:

`all`	compiles your project with the correct compiling options.
`clean`	removes all temporary and compiler-generated files.
`distclean`	follows the behavior of **clean** and also removes binaries and libraries.

Your project will be tested by executing **`( [ -x ./configure ] && ./configure ); make`**, then launching the resulting executable file.

▷ **Instruction 8** A *non-clean* tarball is a tarball containing forbidden files: **`*~`**, **`*.o`**, **`*.a`**, **`*.so`**, **`*#*`**, **`*core`**, **`*.log`**, binaries, etc.

A *non-clean* tarball is also a tarball containing files with inappropriate permissions.

A *non-clean* tarball will automatically remove two points from your final mark.

▷ **Instruction 9** Your work has to be submitted on time. Any late submission, even one second late, will result in a non-negotiable mark of 0 in the best case scenario.

▷ **Instruction 10:** Functions or commands not explicitly allowed are forbidden. Any abuse will result in a non-negotiable mark of -21.

▷ **Instruction 11:** Cheating, exchanges of source code, tests, test frameworks or coding style checking tools are severely punished with a non-negotiable mark of -42.

▷ **Advice:** Do not wait until the last minute to start the project.

Introduction

This project is the opportunity to show us that you are able to work on LSE's projects. You will approach some notions linked to binaries manipulation and you will discover different ways to work on running programs.

A concise README file is welcomed to explain what you have done or what you tried to do. Of course, your code will be clear and properly commented.

This project is divided into 4 exercises. The last 3 exercises are not totally independent, you are free to choose to have 3 different programs for this exercises or only one program which takes the level as first argument. Both will be in exercises synopsis.

If you have any question, do not hesitate to contact us either on the newsgroup **`iit.labos.lse.recrutement`**, on the IRC channel **`#lse-recrut@irc.epita.fr`** or by email at **`recrutement@lse.epita.fr`**.

Have fun and impress us!

1 Level 1: my_strace

Program name:	**my_strace**
Directory:	**login_x-my_gcov/src/my_strace/**
Recommended includes:	sys/ptrace.h, sys/user.h

Synopsis

```
./my_strace /path/to/traced/binary [args]
```

1.1 Objective

In this exercise, you will produce a program similar to **`strace(1)`**. It will print the syscalls called by another process.

1.2 Principle

For this purpose you will use **`ptrace(2)`**. You will have to run the binary given to your **`strace(1)`**. It should be traced by its parent (see the command **`PTRACE_TRACEME`**. Then use the command **`PTRACE_SYSCALL`** of **`ptrace(2)`** to notify the parent before and after each syscall. So the parent can read its child's registers to determine which syscall is called.

1.3 Syscalls calling convention

On x86_64 architecture, all arguments are passed using registers. To help you to understand how it works, here are the steps of a system call:

- Put the syscall number in **`rax`**
- Put arguments from left to right in the following registers: **`rdi`**, **`rsi`**, **`rdx`**, **`r10`**, **`r8`**, **`r9`**
- Instruction to trap to the kernel (**`int 0x80`** by example)
- The kernel does its stuff
- The return value of the syscall can be found in **`rax`** register

Knowing the calling convention, you can get useful information from the registers, before and after system calls. See also the **`syscall(2)`** man page.

1.4 Your work: printing syscalls

You will have to display the list of syscalls called by a program, followed by their return value. Have a look to the file **`/usr/include/asm/unistd_64.h`** to determine which number corresponds to which syscall.

You are free to choose an appropriate output format. You can take your inspiration from this example:

```
42sh$ ./my_strace /bin/ls
brk() = 6557696
mmap() = 140077684056064
access() = -2
open() = 3
```

```
...
close() = 0
munmap() = 0
close() = 0
program exited with code 0
42sh$
```

1.5 Bonus

The main goal of this exercise is to discover **`ptrace(2)`**. As it is not the main part of this project, *do not lose too much time on this bonus part*.

1.5.1 Bonus 1: display arguments

A useful feature for your my_strace would be to display syscalls arguments. Since it would be tiresome to handle all syscalls, we advise you to handle only the main syscalls. Handling **`open(2)`**, **`close(2)`**, **`read(2)`**, **`write(2)`**, **`mmap(2)`**, **`munmap(2)`** and **`fstat(2)`** is a good base. For example:

```
42sh$ ./my_strace /bin/ls
brk() = 6557696
mmap(addr = 0x0, length = 4096, prot = 3, flags = -1, fd = -1, offset = 0) =
140077684056064
access() = -2
open(pathame = "/etc/ld.so.cache", flags = 1) = 3
...
close(fd = 1) = 0
munmap(addr = 0x7f6660995000, length = 4096) = 0
close(fd = 2) = 0
program exited with code 0
42sh$
```

1.5.2 Bonus 2: Syscall filtering

For this last step, you will implement some features similar to those proposed by **`systrace(1)`**. You will have a kind of sandboxing to test the behavior of your programs. For instance, what is the behavior of **`ls`** if you forbid **`open(2)`** ?

Firstly, you will modify the return value of syscalls to simulate a failure. The only thing to do is to modify a register after a syscall.

Secondly, you will forbid some syscalls. Before a syscall, check if the number in **`rax`** is authorized. If it is not the case, block the execution and return -1.

2 Level 2: my_sscov

Program name:	**my_sscov**
Directory:	**login_x-my_gcov/src/my_sscov/**
Recommended includes	sys/ptrace.h, sys/user.h, elf.h

Synopsis

```
./my_sscov path/to/output path/to/binary [args]
```

or

```
./my_gcov --level2 path/to/output path/to/binary [args]
```

2.1 Objective

With this exercise, you will have a first draft of a code coverage tool. It will be able to follow the execution of a program and get some information.

2.2 Principle

You will use what you learned in the previous exercise to know which instructions are executed by the tracee. Moreover, you will have to read the ELF file that you're executing.

2.2.1 ELF

An ELF is an executable format used on UNIX platforms. It contains a lot of information and can provide most things you could dream of (but no coffee, sorry). It is divided into sections. A section is no more than a chunk of contiguous data. One part of your work will be to parse the ELF file in order to retrieve the sections that interest you. Most of the information you will need in order to do that is available via **`man 5 elf`** on any respectable system.

In this exercise, we will be interested in the `.text` section.

2.3 Your work: print execution trace

You will have to write in a file (given as an argument) the addresses of executed instructions, sometimes followed by some information. Instructions executed in the program libraries (not coming from the `.text` section) have to be discarded.

Information to print (on the same line as the executed address):

- If the executed instruction is a call (opcode `0xe8`), write `CALL`.
- If the executed instruction is a return (opcode `0xc3` or `0xc2`), write `RET` followed by the value of the **`rax`** register (return value of non-void functions).
- Finally, if the executed instruction is a jump (opcodes `0xe9` to `0xeb`, `0x70` to `0x7f` and `0x0f80` to `0x0f8f`), write `JMP` followed by the values of the CPU flags.

2.3.1 Output format

You are free to choose an appropriate output format. You can take your inspiration from this example:

```
42sh$ ./my_sscov out ./my_super_factorial && cat out
...
0x40063a:
0x40063f:
0x400640: RET  0
0x400580:
0x400587: JMP  0x246
0x400589:
0x40058a:
0x40058d: CALL
0x400500:
0x400505:
0x400506:
0x40050c:
0x400510:
0x400513: JMP  0x297
0x400530:
0x400531: RET  7
0x400592:
0x400593:
0x40059a:
child exited with status 0
42sh$
```

2.4 Bonus

The main goal of this exercise is to discover ELF files. As it is not the longest part of the project, *do not lose too much time on this bonus part*. Bonuses are independent.

2.4.1 Bonus 1: Skip library execution

As you have probably noticed, the execution of a traced process is really slow. It is because of huge amount of syscalls. One way to reduce the number of syscalls is to avoid single-stepping into library code.

To do this, you will have to enhance a bit your instruction parser (the one you wrote to recognize routine calls), and put a breakpoint (opcode `0xcc`) on the instruction just after a call to a library, using **`ptrace(2)`**, and continue normal execution. The breakpoint will trigger when the execution returns from the library and you will be able to control execution again. Don't forget to remove the breakpoint and to execute the previously alienated instruction.

There's another way to do this if you think about `call` as the result of two other instructions combined.

2.4.2 Bonus 2: Pretty print

Addresses of executed instructions? Who's able to read that?! Using a library (let's say capstone [1]) that disassemble executed instructions, you could also be able to write the assembly associated!
Wait, who's able to read that?!

[1] http://www.capstone-engine.org/

3 Level 3: DWARF

Program name:	**my_addr2line**
Directory:	**login_x-my_gcov/src/my_addr2line/**
Recommended includes:	sys/ptrace.h, sys/user.h, elf.h, dwarf.h

Synopsis

```
./my_addr2line /path/to/binary [args]
```

or

```
./my_gcov --level3 /path/to/binary [args]
```

3.1 Objective

Right now, your program is aware of the instructions executed by the tracee. However, this information is tough to read, and is not necessarily interesting to developers working at a higher level than you are right now. Our objective at this threshold is to get you to make a real code coverage, meaning actually print the source code that is being executed.

3.2 Principle

You will have to combine the tremendous power of your second threshold with some kind of **`addr2line(1)`**-like library of your own making. This utility is located in the binutils package in case you want to have a look. It uses the DWARF format to associate a given address with a tuple line number/file matching the statement the instruction at the given address is part of.

3.3 Example

```
42sh$ ./my_addr2line ./my_addr2line
0x402bd1: /tmp/my_addr2line/main.c:15 | {
0x402be9: /tmp/my_addr2line/main.c:18 |     if (argc < 2)
0x402bf2: /tmp/my_addr2line/main.c:19 |         errx(2, "missing bin");
my_addr2line: missing bin
42sh$
```

3.4 DWARF

Many versions of DWARF exist. All of them are compatible with each other up to a point, except for the first version (DWARF1.) You can get to know about the differences between each version by looking at the changelogs at the beginning of each specification. We will only ask you to handle the latest version, DWARF4. In order for your gcc or g++ to generate DWARF4 metadata, you will have to use the **`-gdwarf-4`** option.

Be careful however, as the different headers in the DWARF metadata might give you different information regarding their version...

3.4.1 Line State Machine

The most trivial way of storing the information that we so desire (the match between a source code line and the instructions it entails, and vice versa) would have been to create a giant matrix. However, this is inapplicable due to size concerns, as you would have guessed.

Instead, DWARF stores a very compact bytecode. This bytecode, when interpreted, allows us to generate the previously mentioned matrix.

3.4.2 .debug_everything

You will soon realise that there is quite of a lot of information available in the DWARF debug format. Most of this information is scattered across the various sections we talked about earlier. This allows DWARF to save lots of space by never storing the same information twice.

You can find below a rapid list of the sections that are most interesting to us, and that you will have to parse, as well as the content you are interested in. Don't lose sight of the fact that each of these sections also contains information that are useless to you.

- .debug_str: file names, paths...
- .debug_line: bytecode generating the all-powerful matrices
- .debug_aranges: matrix matching an address and a compilation unit
- .debug_info: relation between compilation units and chunks of bytecode
- .debug_abbrev: headers of the .debug_info section

3.5 A bit of advice...

3.5.1 More info on DWARF

The first thing you will have to do right now is to fetch the DWARF4 specification and start from there.

In case of doubts regarding the way to handle a given attribute, don't think twice and go take a look at the open-source projects that parse DWARF. This will both enable you to be successful in your project, as well as learn to navigate big and intimidating source trees. *This is greatly encouraged*. Of course, no need to remind you that we already wrote a reference for this project, and that we will be able to recognize **`readelf(1)`**'s source code...

3.5.2 Addresses size

The format itself contains the size of the target machine addresses for more modularity. Given the fact that you will be evaluated with x86_64 binaries, you do not have to consider those fields and can just assume an 8 bytes address size all-throughout the project. However, handling this might be beneficial if you are interested in the laboratory itself...

4 Level 4: my_gcov

Program name:	**my_gcov**
Directory:	**login_x-my_gcov/src/my_gcov/**
Recommended includes	sys/ptrace.h, sys/user.h, elf.h

Synopsis

```
./my_gcov path/to/binary [args]
```

or

```
./my_gcov --level4 path/to/binary [args]
```

4.1 Objective

Following program execution with single-step works but can be very very slow! We will see in this exercise that you don't need to stop the tracee at each instruction. Quite the opposite, the tracee will be stopped only where you put some breakpoints. You will learn how to put breakpoints in a program and where it can be smart to put them.

In this exercise, you will follow the execution of a program. It will probably call some subroutine from external libraries. You do not need to step inside these functions, we only care about the code of the program you are tracing.

4.2 Principle

4.2.1 Basic blocks

According to Wikipedia, "a basic block is a portion of the code within a program with only one entry point and only one exit point". In other words, if you execute the first instruction of a basic block, you know that all instructions to the lase one of the basic block will be executed.

To avoid stopping the tracee at each instruction, you will put some breakpoints at smart locations. Thanks to these breakpoints, you will be able to find the start and the end of basic blocks. So the tracee will be stopped only twice regardless the size of the block.

4.2.2 Breakpoints

It is quite easy to put breakpoints in a traced process. A breakpoint is a **`int 3`** instruction. The corresponding opcode is **`0xCC`**. You only have to put this value in the executed code where you want to stop the program. When a breakpoint is triggered, the tracer gets execution control back. At this time, you will have to find which breakpoint has been triggered. Do not forget to restore the old value, execute the instruction (via **`PTRACE_SINGLESTEP`**) and put the breakpoint back.

4.2.3 Disassembling

In order to find all instructions that modify **`rip`**, you will probably want to disassemble the **`.text`** section. There are some libraries which disassemble the code for you. We recommend to use one of them, like Capstone [2]. Capstone is very easy to use. Of course you are free to do what you want, you are not forced to use an external library.

[2] http://www.capstone-engine.org/

4.3 Step 1: Display executed instructions

For this first step, you will display all executed instructions. Do not trace the binary with single steps. Instead, put a breakpoint on every instruction which modify **rip** (**call**, **ret**, **jmp**, ...).

When a breakpoint is triggered, go to the next instruction with a single step (which is the target of your jump): it is the beginning of a new basic block. Continue the execution until the next breakpoint (which is the end of your basic block). Since you saved breakpoints locations, you know which breakpoints have been triggered so which instructions have been executed.

Tip: the entry point of your program is also the first instruction of a basic block.

You are free to choose your output format (printing opcodes or not, adding a separator between basic blocks,...). Here is an example of output:

```
42sh$ ./my_gcov ./a.out
0x400410:       xor     ebp, ebp
0x400412:       mov     r9, rdx
0x400415:       pop     rsi
0x400416:       mov     rdx, rsp
0x400419:       and     rsp, 0xfffffffffffffff0
0x40041d:       push    rax
0x40041e:       push    rsp
0x40041f:       mov     r8, 0x400590
0x400426:       mov     rcx, 0x400520
0x40042d:       mov     rdi, 0x400506
0x400434:       call    0x4003f0
0x400520:       push    r15
0x400522:       push    r14
0x400524:       mov     r15d, edi
0x400527:       push    r13
0x400529:       push    r12
0x40052b:       lea     r12, qword ptr [rip + 0x2001ae]
0x400532:       push    rbp
0x400533:       lea     rbp, qword ptr [rip + 0x2001ae]
...
```

4.4 Step 2: Source code coverage

This step concerns only programs compiled with debug options. If a traced program was not compiled with debug options, your **my_gcov** will behave as the previous step.

You will now combine the first step with your **my_addr2line**. Instead of printing executed instructions, you will associate them with the corresponding file/line of source code. So you will be able to produce an annotated version of source files. Your output format will be inspired from **my_gcov(1)**.
Example:

```
42sh$ ./my_gcov /tmp/a.out > /dev/null && cat /tmp/main.c.cov
-:    1:#include <stdio.h>
-:    2:
2:    3:void fun(void)
```

```
-:    4:{
2:    5:    printf("You are in a function !\n");
2:    6:}
-:    7:
1:    8:int main(int argc, char **argv)
-:    9:{
1:   10:    if (argc > 1)
-:   11:        printf("This program does not need any argument\n");
-:   12:
1:   13:    fun();
1:   14:    fun();
1:   15:    return 0;
-:   16:}
42sh$
```

4.5 Bonus

4.5.1 Conditional jumps

You know how many times a conditional jump instruction is executed. It would be great to know how many times the condition was true and how many times it was false.

4.5.2 Other architectures

Try to adapt your program to deal with other architectures than x86_64 (like MIPS, ARM, Sparc,...).

4.5.3 Dead code

You could detect the lines of code (not the blank ones) which were not executed during this execution and mark them in the output.

4.5.4 PE

You worked with elf file format on Linux. But what about PE format and Windows ?

4.5.5 Free porn

Do what you want to impress us!

5 CTF

Do not forget the CTF[3] ! ;)

[3] http://ctf.lse.epita.fr/