My memcheck

Version 1
7 December 2015

Epita systems/security laboratory 2017 <recrutement@lse.epita.fr>

Copyright

Contents

Obligations

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

Submission

Project managers:	Vincent GATINE < `nurelin@lse.epita.fr`> Antoine FROGER < `antoine.froger@lse.epita.fr`> Ismaël FEZZAZ < `eliams@lse.epita.fr`>
Project markup:	**`[MMCK]`**
Developers:	1
Submission method:	Git ACU
Submission deadline:	20 December 2015 at 11h42
Project duration:	2 weeks
Newsgroup:	iit.labos.lse.recrutement
Architecture/OS :	x86_64 Linux
Language(s):	C++
Compiler:	**`g++ 5.2.0 - clang++ 3.7.0`**
Compiler options:	**`-std=c++11 -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:

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

  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_mmck/AUTHORS *

  login_x-my_mmck/README *

  login_x-my_mmck/TODO *

  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 , 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. 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.rezosup.fr`** or by email at **`recrutement@lse.epita.fr`**.

Have fun and impress us!

1 Level 1: memory strace

Synopsis

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

1.1 Objective

In this threshold, you will track system calls done by the traced program. This will give you an overview of the `ptrace(2)` uses and a code skeleton for the next parts of the project.

1.2 Your work: printing syscalls

You will have to trace the following syscall execution:

- execve
- fork, vfork, clone
- exit, exit_group
- mmap, mremap, mprotect, munmap
- brk

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

```
sh$ ./mem_strace /bin/ls
execve() = 0
brk() = 0xa46000
mmap() = 0x7f44f9663000
(...)
program exited with code 0
```

1.3 Information

1.3.1 x86_64 system calls

A system call, a.k.a. syscall, is a program request to the kernel. On x86_64, all syscall arguments are passed using registers. Here are the steps of a system call on this architecture:

- Set the syscall number into the `rax` register.
- Set arguments, from first to last into the following registers: `rdi`, `rsi`, `rdx`, `r10`, `r8`, `r9`.
- Execute the `syscall` or `int $0x80` instruction to notify the kernel to take control.
- The kernel executes the code corresponding to the system call and sets the return value into the `rax` register.

Beware that the `C` library wraps system calls so when you call `brk(2)` from a `C` program, it is actually the wrapper of the `C` library that is used which will itself invoke the system call. Besides, some syscalls prototypes change from the kernel and the `C` library point of view so look closely at the manpages.

See:

- `syscall(2)` manpage
- `/usr/include/asm/unistd_64.h` header
- System V application binary interface available here
- Stackoverflow thread: "What are the calling conventions for UNIX & Linux system calls on x86-64" available here

1.3.2 The `ptrace(2)` system call

Syscall that allows to take control over the execution of a traced program. It is primarily used to implement breakpoint debugging and system call tracing. A traced program first needs to be attached to the tracer by the use of the `PTRACE_ATTACH` or `PTRACE_TRACEME` request. Then, the tracer can analyse and modify the tracee memory, registers and control flow.

See:

- `ptrace(2)` manpage
- `/usr/include/sys/ptrace.h` header

1.4 Bonus

1.4.1 Follow child on fork

This bonus will impact each threshold of the project. The goal is to track child processes created by your traced program. A good start is to begin using the `PTRACE_SETOPTIONS` request to enable events. For example, `PTRACE_O_TRACECLONE` options will automatically start tracing the newly cloned process, which is exactly what we want here.

From now, you will add the process id at the beginning of each line of output. For example:

```
sh$ ./mem_strace /bin/ls
[pid 1240] execve() = 0
[pid 1240] brk() = 0xa46000
[pid 1240] mmap() = 0x7f44f9663000
(...)
[pid 1240] clone(Process 2415 attached)
[pid 2415] stat() = -1
(...)
program exited with code 0
```

1.4.2 Display syscall arguments

Display syscall arguments as `strace(1)` do. For example:

```
sh$ ./mem_strace /bin/ls
execve(filename = "/bin/ls", argv = ["/bin/ls"], [...]) = 0
brk(addr = 0) = 0xa46000
mmap(addr = NULL, length = 175717, prot = PROT_READ, flags = MAP_PRIVATE,
     fd = 3, offset = 0) = 0x7f44f9663000
(...)
program exited with code 0
```

Do not take too much time on this bonus, it is not the main objective of this subject. . .

2 Level 2: mem_strace_hook

Synopsis

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

2.1 Objective

When you handle a `SIGTRAP` triggered by a `PTRACE_SYSCALL` request, you cannot control correctly the traced program registers. We need to have control before entering in the syscall and this can be resolve using multiple techiques.

2.2 Your work

2.2.1 Replace `syscall` instructions by breakpoints

Access the `r_debug` structure Step by step:

- Retrieve the traced process ELF program header from the auxiliary vector.
- Find the dynamic segment.
- Get the `r_debug` address from the `DT_DEBUG` tag.

Do not forget that part of the dynamic section is filled dynamically by `ld.so` so after a call to `execve(2)`, you will have to singlestep until the `r_debug` structure is filled again.

Track link map changes Comment for the `r_brk` address in the `r_debug` structure:

```
This is the address of a function internal to the run-time linker,
that will always be called when the linker begins to map in a
library or unmap it, and again when the mapping change is complete.
The debugger can set a breakpoint at this address if it wants
to notice shared object mapping changes.
```

Thus, by breaking and keeping trace of this address, you will be able to track all changes to the link map. The `r_state` field will then give you the type of modification applied to the link map.

Break on syscalls When the dynamic linker has finished to add or delete libraries from the link map, the `r_state` is set to `RT_CONSISTENT`. Therefore, if libraries have been added, you have to parse the loaded ELFs to analyse code and replace each occurrence of the `syscall` instruction by a breakpoint. Beware that the section table is actually not loaded by the dynamic linker so you will have to reopen the file from your filesystem.

Moreover, you have to keep track of each replaced syscall address in order to distinguish `SIGTRAP` signals caused by syscall breakpoints from others.

Obviously, you need to remove the breakpoint addresses from your list when the dynamic linker removes libraries.

2.2.2 Alternatives

Thread Create a thread in your traced process that will be waken up every time a syscall is hit.

Seccomp Use the `seccomp(2)` syscall with `ptrace(2)`.

Other Find ways to avoid the `PTRACE_SYSCALL` request side effects if there are any.

2.2.3 Expected output

To complete this threshold, we expect you to give the same output than the previous one but with a method that allows you to modify memory permissions before executing a syscall.

2.3 Information

2.3.1 Executable and Linking Format (ELF)

ELF is an executable format used on UNIX platforms. It contains a set of sections, interpreted by the linker, and a set of segments, interpreted by the program loader. In this project, we will mainly be interested into the:

PT_LOAD segments: Will be loaded into memory by the program loader.

SHT_DYNAMIC section: Holds information for dynamic linking.

See:
- `elf(5)` manpage
- `/usr/include/elf.h` header
- `readelf` program and its associated manpage `readelf(1)`

2.3.2 Dynamic Linker

Resolves shared object dependencies for dynamically linked programs. We are particularly interested in the `r_debug` structure that is filled at run-time to communicate details of shared object loading. It can be found in the dynamic section of the ELF program and contains a pointer to the link map, a list of all loaded shared objects.

See:
- `ld.so(8)` manpage
- `/usr/include/link.h` header

2.3.3 The `proc` pseudo-filesystem

From now on, you will need to use the process information pseudo-filesystem `/proc`. It provides an interface to kernel data structures that will be useful to gather information on the traced program that cannot be retrieved otherwise.

See:
- `proc(5)` manpage
- `Documentation/filesystems/proc.txt` in the Linux source tree

2.3.4 Auxiliary Vector (auxv)

Auxiliary vectors are a mechanism to transfer certain kernel level information to the user processes. There are multiple ways to retrieve data contained in auxiliary vectors, for example, the `getauxval(3)` function or the file `auxv` in the procfs.

See:
- `proc(5)` manpage
- `getauxval(3)` manpage
- `/usr/include/linux/auxvec.h` header
- `LD_SHOW_AUXV` environment variable

2.3.5 Virtual Dynamic Shared Object (vDSO)

Some system calls used intensely can dominate overall performances, due to the frequency of the call as well as the context-switch overhead that results from exiting user space and entering the kernel. This is why the vDSO exists, it is a small shared library that the kernel automatically maps into the address space of all program.

See:
- `vdso(7)` manpage
- `Documentation/vDSO/*` in the Linux source tree
- Adrien Schildknecht "vsyscall/vDSO" lightning talk available here

2.3.6 x86 breakpoints

A breakpoint corresponds to an `int3` or `int 0x03` instruction. In practice, only the `int3` instruction is used because of its opcode size. When a breakpoint is triggered, a `SIGTRAP` signal is sent so the tracer can get control back on the traced program.

See:
- Intel Instruction Set Reference (Volume 2) available here
- `signal(7)` manpage

2.3.7 Accessing process memory

There are multiple ways to access a process memory space:

▷ **The file `mem` in the procfs:**

pros Straightforward: accessible through `open(2)`, `read(2)` and `lseek(2)`

cons Cannot write

▷ **The `ptrace` syscall using the `PTRACE_PEEKTEXT` and `PTRACE_POKETEXT` requests:**

pros Overwrite page protections

cons A pain to use when large chunk to read or write

▷ **The `process_vm_readv(2)` and `process_vm_writev(2)` syscalls:**

pros Flexible and easy to use

cons Take care of the traced program page protections

See:
- `proc(5)` manpage
- `ptrace(2)` manpage
- `process_vm_read(2)` and `process_vm_write(2)` manpages

2.3.8 Disassembling

Disassembling is the process to translates machine language into assembly language. Some `C` libraries can disassemble the code for you. `Capstone`[1] is one of them and is really easy to use.

See:

– `objdump` program and its associated manpage `objdump(1)`
– `/usr/include/capstone/{capstone,x86}.h` headers

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

3 Level 3: memory tracker

Synopsis

```
./mem_tracker [--preload lib.so] /path/to/traced/binary [args]
```

3.1 Objective

Track memory allocation, deallocation and protection. We need to have the exact representation of the underlying mappings of the traced program.

3.2 Your work

3.2.1 Tracking memory changes

You will maintain a list of the mappings done by the traced program that will be updated when the following syscalls are executed:

mmap Store the range and the protection of the mapping. Consider only non-shared and anonymous mappings.

mremap, mprotect, munmap First of all you must find the mappings that will be affected in your list. Then, you have to update or remove these mappings accordingly to the syscall. Be aware that these syscalls can affect fewer pages than the corresponding call to `mmap(2)` has allocated. In this case, you potentially have to split the original mapping in your list.

brk You also need to register allocated memory and protection of pages mapped by this syscall. First you will have to get the start address of the program break. Fortunately, the program will always do a `brk(0)` syscall at the very beginning so you can retrieve it. After that you just have to track the incrementation and decrementation of the program break and change the mapping data accordingly.

3.2.2 Preloading the memory allocator

You have to preload the memory allocator functions: `malloc(3)`,`calloc(3)`,`realloc(3)` and `free(3)` to track sub-mappings. There are multiple ways to do so, we will let you think about a solution here but here are some hints:

- Consider internal memory allocator mappings separately.
- Remember that `dlsym(3)` uses `calloc(3)`.
- There is some information in the link map that can help you to avoid `dlsym(3)`.
- Think about a way to retrieve information in your tracer from your preloader.

Your goal is to maintain a list of sub-mappings done by the `C` memory allocator. You can take inspiration from this output:

```
sh$ ./mem_tracker tests/mem_tracker1
(...)
brk { addr = 0xc2d000, len = (nil), prot = 3 }
 to { addr = 0xc2d000, len = 0x32000, prot = 3 }
mmap { addr = 0x7f4dc679e000, len = 0x1, prot = 0 }
```

```
mremap { addr = 0x7f4dc679e000, len = 0x1, prot = 0 }
    to { addr = 0x7f4dc5543000, len = 0x400063, prot = 0 }
mprotect { addr = 0x7f4dc5543000, len = 0x400063, prot = 0 }
      to { addr = 0x7f4dc5543000, len = 0x400063, prot = 3 }
munmap { addr = 0x7f4dc5543000, len = 0x400063, prot = 3 }
(...)
malloc { addr = 0x12d0040, len = 0x38 }
calloc { addr = 0x177d120, len = 0x250 }
realloc { addr = 0x12d0040, len = 0x38 }
     to { addr = 0x12d0100, len = 0x9600 }
free { addr = 0x12d0100, len = 0x9600 }
free { addr = 0x177d120, len = 0x250 }
(...)
```

3.3 Bonus: Handling split mappings

`mremap(2)`,`mprotect(2)` and `munmap(2)` syscalls can affect fewer pages the corresponding call to `mmap(2)` has allocated. In this case, you may have to split the original mapping in your list.

Once again, here is an output example:

```
sh$ ./mem_tracker tests/mem_tracker2
(...)
brk { addr = 0x249e000, len = (nil), prot = 3 }
 to { addr = 0x249e000, len = 0x32000, prot = 3 }
mmap { addr = 0x7f503ed12000, len = 0x1000, prot = 3 }
mremap { addr = 0x7f503ed12000, len = 0x1000, prot = 3 }
    to { addr = 0x7f503dab8000, len = 0x400000, prot = 3 }
mprotect split { addr = 0x7f503dab8000, len = 0x165a000, prot = 3 } into
             * { addr = 0x7f503dab9000, len = 0x1000, prot = 0 }
             * { addr = 0x7f503dab8000, len = 0x1000, prot = 3 }
             * { addr = 0x7f503daba000, len = 0x1658000, prot = 3 }
munmap split { addr = 0x7f503daba000, len = 0x1658000, prot = 3 } into
           * { addr = 0x7f503daba000, len = 0x2000, prot = 3 }
           * { addr = 0x7f503dabd000, len = 0x1655000, prot = 3 }
(...)
```

4 Level 4: memory checker

Synopsis

```
./mem_checker [--preload lib.so] /path/to/traced/binary [args]
```

4.1 Objective

You have: a breakpoint on each `syscall` instruction and the exact representation of your traced program memory space. You can detect memory leaks and setup a mechanism that will catch invalid memory accesses from the traced program. To do so, each time a new mapping via `mmap(2)` is done you have to remove the pages protection to trigger `SIGSEGV` signals when the program try to access memory. Then you can analyse the fault address, singlestep and recover the pages protection. This will allow you to verify every memory access done by your traced program.

4.2 Your work

4.2.1 Removing page protection

Modify your `mmap(2)` handler to be able to overwrite the protection on allocated mappings. Remember that you only have to consider non-shared and anonymous mappings. Besides, for executable mappings, let the `PROT_EXEC` flag to avoid traps on instruction fetch. In order to remove mappings protection, you need to invoke the `mprotect(2)` syscall in the traced program. To do so, you have multiple options:

▷ **Inject code at current `rip` and restore the overwritten instructions afterward**

pros simplest implementation

cons not elegant
will not work with multithreaded programs

▷ **Map a page into the traced process memory space at the very beginning of the traced process execution and write the `syscall` instruction**

pros allows optimizations: you write the code once and can use it anytime everywhere

cons will not work with multithreaded programs

▷ **Return-Oriented Programming to find a `syscall; ret` sequence**

pros when linked dynamically this sequence exists in `ld.so`
fun!

cons when linked statically this sequence may not exist

▷ **Use the link map from the threshold 2 to find the address of the `mprotect(2)` function of the traced process `C` library**

pros "clean" way to do it
fun!

cons more complicated than other techniques

4.2.2 Restoring page protection

Now that you have removed page protection on every mapping, each time the traced program will read or write into the memory, it will receive a `SIGSEGV` signal. This is exactly what we want, and you will use this to detect invalid memory access in the next step. For now, each time you trigger a `SIGSEGV` signal you have to restore the original pages protection, singlestep the instruction that caused the fault and remove again the protections.

That is not all! If you only take care of the `SIGSEGV` signals, every syscall that will be executed and access traced memory will fail. Remember what we have done in the threshold 2? It is time to use it! Each time you get a `SIGTRAP` signal caused by a syscall breakpoint, restore original page protections, singlestep and remove them again. This will prevent the tracer to break the traced program execution.

4.2.3 Invalid memory access

At this point, your traced program can normally be executed normally from the user point of view. But we currently have not detected any memory violation...

- Analyse your `rip` register to get the address of the instruction that has generated the `SIGSEGV` signal.
- If the address is not part of one of the mappings in your list: this is an invalid memory access.
- Get the fault address from the signal information retrieved with the `PTRACE_GETSIGINFO` request.
- Display the instruction that generated the signal and the fault address.
- Restore the corresponding mapping protection, singlestep and remove it again.

Here is an example of output:

```
sh$ ./mem_checker tests/invalid-read
Invalid memory access of size 4 at address: 0x7f567cc0d001
0x400066d: movzbl (%rax), %eax
(...)
```

4.2.4 Displaying memory leaks

If your mapping list is not empty at the traced program exit, it means that there are memory leaks. Actually, you will always have false positives because some libraries as `ld.so` does not unmap allocated pages and let the kernel do it at exit as they stay during the whole program execution. An example of output:

```
sh$ ./mem_checker tests/leak1
(...)

Memory leaks: total of 0x21723 bytes not liberated at exit
    * address = 0x7f567cc0d000 - length = 0x1000
    * address = 0x7f0a82e03000 - length = 0x1789
(...)
```

4.3 Bonus

4.3.1 Display more invalid memory access information

Now that you have the address of fault and of your instruction, by playing with `Capstone`, you can retrieve lot more information, such as:

- Invalid read/write
- If multiple invalid accesses from the same type are done consecutively, display only one line with the total size of the invalid memory access and the base address with their corresponding instructions.
- Display the entire call stack
- Use DWARF debugging format to display `C` line triggering the invalid access.

4.3.2 Remove memory leaks false positives

To remove memory leaks false positives, you have to get the complete call stack of the traced program. So when you execute the `mmap(2)` syscall, you can know in which code portion the allocation was requested and store the call stack to display it at exit on each memory leaks.

4.3.3 Non exhaustive list

From easiest to hardest:

- Memory leaks
- Support for other architectures (x86, ARM, PPC, . . .)
- Conditional jumps/move that depend on uninitialized values
- Portable Executable (PE) format support