



CODE INJECTION

Software Project

January - April 2020

Dupré Nail
First year CSE Student

Introduction

This report aim to showcase various things about our Software security Project:

- Some generals facts about our code, its compilation, and so on.
- How did we structure our code, and achieve the various step composing this project.
- the various “tricks” that we used all along the development of our program.

Execution and organisation of code:

The archive/directory given to you is organized in the following way: there is one sub-directory for every step, the content of each directory was copied to the next one once a step was done and validated. The directory from step 1 to step 6 were left alone, as per our given instruction. The directory named step 7 was “cleaned” , the code re-organized and checked. This directory can execute every step successfully and this reports refer to the code within the directory step 7.

Generals comment about our code

Makefile:

We have severals compilation methods in our makefile; we may compile with gcc and the “traditional” flags -Wall and -Wextra, or with clang and the -Weveryting flag. We also have some additional compilation options, like switching from debug to default Mode, we will explain in more details this mode later. We may also compile our tracer with clang and its -fsanitize=undefined flag. For all of these compilation methods , our program works as expected, and produce no warnings during compilation.

Some attribute where given to functions in order for our code to function properly with certain flag. For example, the attribute no_sanitize(“undefined”) was given to our virus inside of our tracer, the code injection would otherwise crash, since our sanitizer could inject code inside our function that could potentially not be linked properly.

To execute our code, go into the step7 sub-directory and input the command “make”. For further options with compilation, look inside the makefile.

Debug and default Mode:

We have implemented a debug mode for our tracer, this mode display additional information, mainly about return values and attributes that were deemed “sensitive” (I.E prone to causing crash). This mode is enabled through a macro defined during compilation through the -D flags, which is used by the preprocessor. As you may see in our file headers.h, our macro are either defined to a fprintf to stderr, or nothing. This means that if we are to execute our tracer when our debug mode is activated, then those print will be executed.

Other general comments:

There is severals other things that we can point out about our code: we are using no VLA, we instead use pointer to allocated memory. We have no leaks nor memory errors when we execute our tracer, we have used valgrind extensively in order to achieve this. We have also tried to generalized part of our as much as possible, so that the same function could be used in order to call Mprotect, memAlign or free for example. We have also tried to use both const and static as much as possible.

For the execution, you may execute all function together (except trampoline and indirect call for step 7, you must choose one). Or merely execute one. Please remember that all subsequent functions are using value from step 1, so you must leave the call to it uncommented in our tracer. The steps 5 to 7 use the result of step 4, so the call to it must be left uncommented as well if we wish to execute these steps. Finally step 6 does not depends on call to step 5 in the main of our tracer (step 6 call step 5 from within its own functions), so you may comment the call to step 5 in our tracer if you wish to execute step 6. Same things with step 7 in regards to step 6 and 5.

We start our tracer by getting the necessary informations for the rest of our execution, this means the pid and address of the libc (for step 4 to 7) of both our tracer and tracee and the length of both pid. After step 1, we attach the tracer to our tracee, using the pid. We will only detach ourselves after the steps we wish to execute are done.

Step 1: let's start by looking at the map

This step has two main parts: first getting the offset of our function f1, and then adding it to the range of address where our f1 is. We also get the pid of our tracee in this step and save it. Both this PID and the address of f1 will be used throughout the rest of our program.

In order to find the range of address where our f1 is, we look inside /proc/pidTracee/maps, “scan” each line until we find one having both tracee and the execution right r--p in it. We then proceed to save it. For the offset, we simply check the result of the command nm on our tracee and save the information found on the line having our function name, here f1.

The three functions used here all act in the same general way, we prepare a command, execute it, and then get the desired informations from the result.

Step 2: trap, trap, trap

The goal of this step was to stop our tracee, and then set it free. In order to do so, we must modify the content of f1 in our tracee. We already know the pid of our tracee, as well as the address of f1 (from step 1) what we must now do is write our software interrupt in f1.

To do so we start by opening /proc/pidTracee/mem. We then position ourselves at the beginning of f1, backup the first instruction that was there (we will use it to cleanup our injection later) and then write our software interrupt. We then sleep in order to show the effect of our software interrupt on our tracee. After this is done we rewrite the original instruction at the beginning of f1. Our tracee will then proceed as usual.


As you may see later, the functions backupandwrite and rewriteBackup will be reused. This is why we were using a pointeur for our software interrupt even if a “normal” unsigned char would have been enough.

Step 3: Call me if you can

For step 3, we are supposed to call another function found in our tracee. to do so, we call the callFunction function. This is one of the most central part of our program, it will be used to call f2, pagesize, memAlign, mProtect and so on in our tracee.

callFunction:

This function takes several parameters: an address (always of f1), where we will write our indirect call, the pid of our tracee, the length of this pid(to malloc a pointer used to open /proc/pidTracee/mem) and finally a struct regs_user_struct. This struct is of utmost importance, some of its attribute will be set in order for our indirect call to function properly. For all its call, regs.rip will be set to f1, and rax will be set to the address of the function we wish to call. the three register rdi, rsi and rdx will either be set to zero or to the value of the parameter necessary for our call.



This function start by inserting a software interrupt in our function f1, using the function backupAndWrite, we then get the current registers for our tracee and modify them accordingly. rip and rax will always be modified, will the three others will be modified only if the attribute of the structure passed in parameter to callFunction are not set to zero.

We then send a cont signal to our tracee, and wait for its response. After that, we can write our indirect call safely, since our tracee is at a full stop. We also set our registers to those modified just now. We once again send a cont signal and wait. After our tracee as “stumbled” upon our second software interrupt (written at the same moment and right after our indirect call). We “clean” our injection just as we did in step 2 setting back our registers as well this time, and set free our tracee.

Step 4: Addresses of another dimension

The objective of this step is to call another function (pagesize), found in the libc, in our tracee. To do so we will reuse callFunction.

We must first get the address of our function pagesize. We have three information “directly” (through the execution of a command) given to us, the address of the libc of our tracee, the address of the libc of our tracer, and finally the position of our function pagesize. Since the position of pagesize is the same in every libc, once we have these three information, we can compute it by doing one subtraction and one addition.

We start by getting the address of both our libc (we actually got it at the beginning of our tracer, but we will explain it now), to do so, we open the /proc/pid/maps file for our tracee and tracer, and look inside it until we find a line having both libc and r-xp in it. Libc so that we know that our data windows is indeed belonging to the libc, and r-xp because we want our function to be executable. We then only select the first address of this line (the beginning of our data windows).

After having these three addresses (we get the address of pagesize by simply doing (long)getpagesize) we subtract the address of the libc of our tracer from the one of our tracee, then add the address of our function pagesize. This give us the address of the function pagesize inside the libc of our tracee. We now only have to call callFunction with the right register (rip will contains the address of f1, rax the address that we just computed, rdi, rsi, rdx will all be set to zero since our function pagesize take no parameters). We then return the result of this call, it will be used for our next steps.

Step 5: Make some space for my virus please

This step's goal was to allocate space in the memory of the tracee. To do so, we must call the `posix_memalign` function in it. If your call is successful, the struct register returned by `callFunction` will have its attribute `rdi` set to an address within the heap of the tracee while `rax` will be set to zero.

We proceed as in step 4, calculating the address of `posix_memalign` inside our tracee, and then settings our register accordingly: As always, `rip` will be set to the address of `f1`, and `rax` to the address of the function we wish to call. `posix_memalign` need three parameters, which mean that we will set `rdi`, `rsi` and `rdx`. the first parameter will be an address, in order to get it, we must subtract an arbitrary (but still in accordance with the man of `posix_memalign`) value from `rsp`. This mean that this value cannot be set when we are not inside our `callFunction`, since it depends of the value of `rsp` of `f1` in our tracee. This is why we are using a macro indicating `callFunction` that it is in this special case, and must thus compute this value. The second parameter will be our alignment, according to the man it must be a power of two and a multiple of `(void*)`. And finally the size of the allocation, we here decide to arbitrarily assign a whole page, using the information we got from step 4.

We finally return the `rdi` of our register. In debug mode, we also print these two value, to access whether or not this call was successful.

Step 6: Fabricate and Inject

In step 6, we are supposed to create a code cache, which is an executable memory zone, and fill it with the code from our virus found in our tracee. We start by getting the length of our virus, to do so we simply execute the command "`nm -S -t d tracer`" and search for a line containing our function virus (which we have name `virusFunc`, so as to not confuse it with our attribute `virus`, which will contain the content of our virus function). the `-S` option print the size of our elements will `-t d` specify that we want this size in decimal.

After that we will compute once again the address of `mProtect` and set up its call by setting the struct register accordingly. But first, we must align the result of our call to `memAlign` with our `pagesize`, otherwise our call to `mProtect` will fail. since, according to the man, the address passed to `mprotect` must be aligned to a page boundary.

To do so, we compute the following operation, where `pagesize` is the result of step 4, and `memAlignRes` the result of step 5:

```
alignedToPageBoundary = memAlignRes & ~(pagesize - 1)
```

We want to align the address that memAlign has given us to our pagesize. For this example, let's say the address is 51231234, our pagesize 4096.

1 give us 4095, which is 1111 in binary. we then apply the bitwise NOT operator ~ on this, giving us 0000.

We then apply the bitwise or between our address and 0000, which will "force" our Address to be aligned to our pagesize.

In our example, our address thus change from 51231234 to 51230000. Which is aligned to a page boundary.

We then proceed to fill our the remaining register: as always, rip is f1 and rax is the address of the function to call. Our first parameter (rdi) is our newly computed address ,while the second one (rsi, the size of the memory we wish to change the rights of) will be pagesize. Finally our third argument are the execution, as we wish to add the execution rights to our memory windows, which as already the write and read rights, we fill our rdx with PROT_EXEC | PROT_WRITE | PROT_READ. This is because our rights will be overwritten by the call to mProtect, not added.

We finally get our virus, and write it to the memory zone beginning with the return of step 5, not at the beginning of our re-alignment. Our virus will still be within the zone with execution rights, since we have changed the right for a memory zone much bigger than what was necessary for the virus. With this end step 6.

Step 7: My first hack at last.

This step will make us call our virus, that we have already written in our tracee in two different ways: through a trampoline and an indirect call. In both cases, we start by calling step 6, that will do all preliminary works necessary for the call.

Trampoline:

The trampoline acts in the same general way as our callFunction, except that we must this time write a trampoline instead of an indirect call. This means that instead of writing our indirect call, we write the first part of our trampoline (0x48 and 0xB8), the address where we wish to jump to (where our virus is) and finally the last part of our trampoline (0xFF, 0xE0, and 0xC3). We do not need to cleanup our trampoline, since we want to permanently call it from our tracee.

Indirect Call:

Our indirect call will once again reuse callFunction: our rip is set to the address of f1, our rax to the address where the virus was just written, and our first argument to any integer we desire, since our virus function take one as parameter. We then use callFunction to call it, get its return value and display it. We must then do some additional cleanup to remove our marks on the tracee. First is setting back the rights to our data windows to PROT_READ and PROT_WRITE only. To do so, we setup our call in the exact same way as in step 6, this time with rdx being set to PROT_WRITE | PROT_READ. We then proceed to free the aligned address in our tracee, and then finally overwrite our virus by replacing it with 0. To do so we simply create an array of size virusFunc, fill it with '\0' and then write it to the desired memory address. Our tracer execution is now done, we got the right return value, and all of our traces were removed.

We then detach our tracer from our tracee in both case, and we are done.

ScreenShots:

Here are two screenshots of our execution, one with the debug mode on, the other without. We are here executing every steps (step 7 is the indirect call):

Debug mode on:

```
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ ls
include  makefile  src
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ make debug
gcc -fpie -pie -Wall -Wextra -o tracee src/tracee.c -Iinclude
gcc -fpie -pie -Wall -Wextra -D'DEBUG_MODE' -o tracer src/tracer.c src/addrFunctions.c src/ptrace_util
ls.c src/step2.c src/step3.c src/step4.c src/step5.c src/step6.c src/step7.c -Iinclude
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ sudo ./tracer tracee f1
[sudo] password for nail:
Beginning of step: 1
In file: src/addrFunctions.c, in function:getFunctionAddr and at line: 29 Return is 0000000000001169
In file: src/addrFunctions.c, in function:getAddrFuncExec and at line: 123 Return of step 1 is 562b
4053b169
In file: src/step4.c, in function:getAddrLibcOf and at line: 38 Address of Libc returned is 7f3556c2d
000
In file: src/step4.c, in function:getAddrLibcOf and at line: 38 Address of Libc returned is 7f0372394
000
Beginning of step: 2
In file: src/addrFunctions.c, in function:getFunctionAddr and at line: 29 Return is 000000000000119c
In file: src/addrFunctions.c, in function:getAddrFuncExec and at line: 123 Return of step 1 is 562b
4053b19c
Beginning of step: 3
Step 3 - Return of f2 function: 110
Beginning of step: 4
In file: src/step4.c, in function:step4 and at line: 146 Pagesize is 4096
Beginning of step: 5
In file: src/step5.c, in function:step5 and at line: 31 Mprotect memory is 562b4198b012
In file: src/step5.c, in function:step5 and at line: 32 Mprotect return value is 0
Beginning of step: 6
In file: src/step6.c, in function:getVirusLength and at line: 33 Virus length is 23
In file: src/step5.c, in function:step5 and at line: 31 Mprotect memory is 562b4198b012
In file: src/step5.c, in function:step5 and at line: 32 Mprotect return value is 0
In file: src/step6.c, in function:step6 and at line: 63 Return of memAlign is 562b4198b012
In file: src/step6.c, in function:step6 and at line: 64 Aligned return of memAlign is 562b4198b000
Beginning of step: 7-Indirect Call
In file: src/step6.c, in function:getVirusLength and at line: 33 Virus length is 23
In file: src/step5.c, in function:step5 and at line: 31 Mprotect memory is 562b4198b012
In file: src/step5.c, in function:step5 and at line: 32 Mprotect return value is 0
In file: src/step6.c, in function:step6 and at line: 63 Return of memAlign is 562b4198b012
In file: src/step6.c, in function:step6 and at line: 64 Aligned return of memAlign is 562b4198b000
STEP 7 - Return of Virus Functions: 10100
In file: src/step6.c, in function:getVirusLength and at line: 33 Virus length is 23
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$
```

```
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ ./tracee
In F1, PRINT! 0
In F1, PRINT! 1
In F1, PRINT! 2
In F1, PRINT! 3
In F1, PRINT! 4
In F1, PRINT! 5
In F1, PRINT! 6
In F1, PRINT! 7
In F1, PRINT! 8
In F1, PRINT! 9
In F1, PRINT! 10
In F1, PRINT! 11
In F1, PRINT! 12
In F1, PRINT! 13
In F2, value of return is 110
In F1, PRINT! 14
In F1, PRINT! 15
In F1, PRINT! 16
In F1, PRINT! 17
In F1, PRINT! 18
In F1, PRINT! 19
^C
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$
```


Debug mode off:

```
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ ls
include makefile src
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ make
gcc -fpie -pie -Wall -Wextra -o tracer src/tracer.c src/addrFunctions.c src/ptrace_utils.c src/step2.
src/step3.c src/step4.c src/step5.c src/step6.c src/step7.c -linclude
gcc -fpie -pie -Wall -Wextra -o tracee src/tracee.c -linclude
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ sudo ./tracer tracee f1
Step 3 - Return of f2 Function: 110
STEP 7 - Return of Virus Function: 10100
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$

nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$ ./tracee
In F1, PRINT! 0
In F1, PRINT! 1
In F2, value of return is 110
In F1, PRINT! 2
In F1, PRINT! 3
In F1, PRINT! 4
In F1, PRINT! 5
In F1, PRINT! 6
In F1, PRINT! 7
^C
nail@nail-HP-EliteBook-840-G3:~/Documents/SoftSec/Project/step7$
```