Libptrace
A cross-platform process manipulation API

Ronald Huizer (rhuizer@liacs.nl)

January 18, 2008

Abstract

Support for tracing and manipulation of processes is an integral part of any modern operating system in order to provide debugging services. The way in which this is supported can vary enormously on different operating systems, and even on the same operating system running on different architectures. In this paper we will present a process tracing and manipulation API which abstracts away these differences and implements primitives for remote code injection and loading of shared library objects.

1 Introduction

Support for tracing and manipulation of both light- and heavyweight processes is radically different on some of the major operating systems available these days. BSD and some System V versions and derivatives provide a specialized systemcall called `ptrace` in order to provide means for one process to observe and change the memory and registers of another process, other System V versions provide this functionality through `ioctl` calls on process entries in the `/proc` filesystem, and Linux - although still using the `ptrace` systemcall - introduced the experimental `utrace` framework some time ago. Windows uses its own API which is fundamentally different from the other ones. To make matters worse, there are a lot of differences between systems seemingly providing the same interface; for instance, many different `ptrace` implementations each have their own peculiarities.

One of the aims of libptrace is to abstract away these differences and provide a common interface which can be used for process tracing. Another aim is to provide process manipulation functions for several common tasks, such as executing code, allocating pages, loading shared libaries, et cetera in remote processes. At the time of writing libptrace supports the Linux `ptrace` interface and the Windows debugging interface running on the i386 architecture, but extending the framework to cover other operating systems and architectures

shouldn't be problematic. This paper gives an overview of what functionality is available on the Linux and Windows platforms, and how we accomplish the various common tasks we mentioned. A more detailed description of libptrace itself can be found in appendix A.

2 Process tracing and manipulation

First we will discuss the part of the API that abstracts away the process tracing and manipulation primitives that both operating systems offer. Before we examine the underlying architecture and operating system closely, we need to determine what process and thread operations we want to perform in order to be able to control how a remote process behaves to the best possible extent.

We will assume that all systems supported will use the concept of processes and threads, where a process is identified by a unique number called a *process identifier*, or *PID*, and a thread is identified by a unique number called a *thread identifier*, or *TID*.

2.1 The Linux framework

Linux actually offers us two threading models, the older LinuxThreads model, and the newer Native POSIX Threads Library, or NPTL, which was introduced to address the shortcomings of the LinuxThreads model. An outdated overview of the NPTL can be found in [DM05] and the differences between NPTL and LinuxThreads are outlined in [Bar00]. The older LinuxThreads model has not been taken into account while designing the libptrace framework, and this paper as well as the libptrace source code assumes use of the NPTL model[†].

On Linux the set of PIDs and the set TIDs is not disjoint; for each process with some PID there is a *main thread* with a TID equal to this PID. This allows for referring to the *main threads* of a process by its PID. This is not the case on Windows, which does not have the concept of a *main thread*, and where the set of PIDs and the set of TIDs is disjoint.

The default Linux debugging interface at this time is the `ptrace` systemcall, which takes a special request type argument differentiating between the various operations the interface offers. It allows a process to open a thread in a remote process by its TID through the `PTRACE_ATTACH` request, and then allows the following tasks to be performed: reading and writing the virtual memory of the process the thread belongs to through `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`, reading and writing the CPU registers of the thread through `PTRACE_PEEKUSER`

[†]This does not mean the framework will not work on LinuxThreads systems, just that it hasn't been designed for it, and this could lead to complications.

and PTRACE_POKEUSER as well as PTRACE_GETREGS and PTRACE_SETREGS, waking the thread through PTRACE_CONT and suspending it (this happens automatically when a signal is received), mediating in delivery of signals sent to the thread, single-stepping through the thread, and a few less relevant operations. An introduction to process tracing and manipulation using ptrace can be found in [S02].

2.2 The Windows framework

Windows supports process tracing and manipulation through their debugging API, which is modelled differently than the Linux ptrace API. There are separate functions available for retrieving a handle to a process (OpenProcess) or a thread (OpenThread), which can then be used in specific functions which manipulate the process. The more interesting of these are GetThreadContext and SetThreadContext for modifying the register state or the context of a thread, CreateRemoteThread for creating threads in remote processes, VirtualAllocEx for allocating memory in remote processes, and finally ReadProcessMemory and WriteProcessMemory for reading from and writing to memory in the remote process.

A running process can also be attached to by the full debugger framework through the DebugActiveProcess function, which is a lot more invasive, but provides additional advantages, such as suspending all threads in the target process[†], and being able to monitor the process and its threads for debugging events such as process and thread creation and termination, breakpoints, and the loading and unloading of libraries. However, there are several disadvantages to using the debugger framework as well. The most notable is the lack of support for detaching the debugger framework from a process once attached on older versions of Windows; the DebugActiveProcessStop function which detached the debugger framework is only provided from Windows XP and onward. Another potential issue is the ease with which process debugging can be detected by the host process, which is a fairly common practice for closed source software which wants to foil debugging attempts. An overview of some of the means to do this is given in [Fal07].

Libptrace currently uses the full debugger framework when attaching to a process, as this is the only way to get notified of software breakpoints in a remote process. Being able to catch breakpoints is necessary for the way libptrace currently performs remote code injection and execution as will be outlined in section 3.2.

[†]This may seem a trivial job, but is actually hard to emulate in userland in a race condition free manner on Windows, as there is no atomic way or function to list all threads in a process and then suspend them. This could be problematic when suspending threads that are actively creating other threads.

3 Remote code execution

We've seen that we can use the primitives discussed in section 2 to manipulate the behaviour of remote processes and threads. A good application of these primitives would be the ability to inject and execute code in a remote process. There are different methods for this, depending on the operating system, and they can vary a lot. Please take note that we are discussing executing code in a remote *process*, which libptrace currently does by executing code in the existing remote *thread* it attached to, but which is not the only way in which this can be done. The method libptrace uses to run code in a remote process performs the following steps:

1. Find or reserve space in the virtual memory area of the target process which will accomodate our payload.
   In order to execute the payload in a remote process, we need to store the payload in the virtual memory area of that process. We have two options here, either explicitly allocating memory in the remote process or using memory that is already there.

2. Make sure this space will contain our payload.
   Once we have found the memory to accomodate our payload, we can easily write out payload there using the primitives for writing process memory.

3. Have a remote thread execute the payload.
   We set a software breakpoint in the remote process at the end of our payload to detect when it finished running, save the context of the remote thread, switch to an alternative stack for platforms using a stack red zone (see section 4.2 for details), set the instruction pointer of the remote thread to the location of our payload, and continue running the thread[†]. After the breakpoint trap the thread suspends, and we restore the thread context and switch back from the alternative stack.

This is by no means the only way in which to execute code in a remote process, and especially on Windows there are a lot of alternative methods available, which have their own advantages and disadvantages. We will discuss them in section 3.2.

[†]We would like to continue running all threads in the process, to prevent one of them from being suspended in the critical section of a function our payload might invoke and causing a deadlock, but this could cause crashes in the other threads in case we're using memory that is already there. This is an issue on Linux systems, but fortunately, we will not suffer from deadlocks when waking a single thread that executes an `mmap` systemcalls for allocating memory which we can use to host code that could cause deadlocks.

3.1 Linux code injection and execution

Linux has no native support for allocating memory in a remote process, so we will have to settle for using memory that is already there. This can be done safely by suspending all threads in the process, thus ensuring none of the threads will access the memory area we're going to use, finding a mapped memory page, and saving the original data by reading it so that we can restore it after we're done executing our payload. The memory page we choose is the beginning of the page that the instruction pointer points to, which seems a good location for this purpose[†]. Everything else can be done using the primitives discussed in section 2.1.

3.2 Windows code injection and execution

Windows provides us with a variety of means in which we can execute code in a remote process. We can use the method outlined above, create a new thread using the `CreateRemoteThread` function, or use Windows remote thread hooks. An excellent overview of several methods to inject code into a remote process and execute it is given in [Kun03]. Unfortunately, there are advantages and disadvantages to all methods, the drawback of the method libptrace uses being that for threads blocking on a Local Procedure Call, or LPC (an overview of LPC can be found in [DBP99]), attaching the debugging framework or changing the thread context will not cause the thread to wake from an LPC wait state. This means that until the call finishes the thread will not run, and not execute our injected payload. This issue would be resolved by using the `CreateRemoteThread` function to create a whole new thread in the process, but this does not allow tracing of processes in a different terminal services session. Finally, using remote thread hooks requires `user32.dll` to be mapped in the remote process, something which might not always be the case. We might implement other methods for code execution in a remote process on Windows later to complement each other.

As mentioned before, Windows comes with a convenient API function called `VirtualAllocEx` which can be used to allocate memory in processes by their process handle. Writing our payload to this memory is trivial as well, and having a remote thread execute it can also be done easily using the primitives from section 2. In order to trap software breakpoints we do need to attach the full debuffer framework, which is another disadvantage of the current libptrace method.

[†]In order for this to work we need to be sure that we can always write data no matter where the instruction pointer points to. This is not the case for the vdso sysenter page in a VDSO_COMPAT enabled kernel, as this page is mapped in the kernel memory range, instead of the userland range, where Copy-On-Write semantics are unsupported. The recent vdso pages are randomized and mapped in the user memory range, where we can write to them without any problems. In case of VDSO_COMPAT kernels libptrace will write out code to the ELF header base, which is at a fixed location.

4 Libptrace primitives

Using the ideas presented in section 2 and 3 we will present some more complex process manipulation functionality that we can now implement. This functionality is very platform dependent, but is abstracted away by libptrace completely, and aims to offer the same primitives on all platforms. These primitives will be outlined below.

4.1 Memory management

There are a lot of situations where one may need to allocate and free memory in the remote process, and because this is such frequent operation, libptrace provides functions for doing this. On Windows there are already calls present which accomplish this, and on Linux it is straightforward to emulate such behaviour by using the method for running code discussed in section 3, invoking the `mmap` systemcall to allocate pages, and providing a custom allocator in order to manage them similar to the ANSI C `malloc` interface.

4.2 Alternative stack management

When executing injected code in a remote process it is very likely this code will use the stack, and thus change the memory area above (for stacks growing up) or below (for stacks growing down) the stack pointer. This can be problematic depending on the situation. Some architecture ABIs, such as the MacOSX PPC ABI (see [App07]) and the System V AMD64 ABI (see [MHAJ07]), define a so called red zone, which is an area of the stack beyond the stack pointer which is guaranteed not to be used asynchronously by signals or interrupt handlers. This allows a compiler to use optimizations where it can use this memory as a scratchpad for data or even local stack frames until the next synchronous function call without explicitly reserving it. It is important that libptrace does not change the contents of the red zone when executing remote code, and to prevent this from happening it offers functions to facilitate management of alternative stacks, choosing to set up an alternative stack over saving the contents of the red zone and restoring it later[†].

[†]Next to being generic, this is actually more efficient as well, as libptrace can set up an alternative stack once, use it for multiple code executions, and destroy the stack when it detaches, as opposed to having to save the entire red zone every time remote code is executed.

4.3 Loading shared libraries

It is very convenient to be able to load a shared library object into remote processes as this provides a very easy way in which to load large code modules and take advantage of the way in which symbols in shared libraries are resolved by the dynamic linker in order to override existing symbols.

We could emulate the way in which the dynamic linker works, and do everything ourselves, but this is cumbersome and bug-prone, as we will need to manipulate all dynamic linker datastructures in order to have it recognize the new library, which will likely create a dependency on specific dynamic linker versions. Rather than doing this, we chose to have the remote process call existing functions for loading shared objects, thus providing means for loading shared objects remotely.

4.3.1 Windows

On Windows shared library objects are called Dynamic Link Libraries or DLLs, which can be loaded by a process through calling the `LoadLibrary` function. This function is present in `kernel32.dll`, which is mapped at the same virtual address in all processes[†]. This makes it easy to locate `kernel32.dll` function addresses through a `GetProcAddress` call, which will return an address that is the same in all processes. Loading a library now becomes as easy as calling `LoadLibrary` in a remote process with the correct parameters, for which we can use the primitives outlined in section 3.

An interesting existing library which does this for Windows only is [Fei05], which can be used to load libraries in a remote process by using the `CreateRemoteThread` technique and calling `LoadLibrary` in that thread. This has the advantage that it works nicely on Windows 9x as well, and does not need to use the Windows debugging framework which we need to do when having an existing thread in a process execute our code and trigger a breakpoint event when done.

[†]Windows DLLs do not contain position independent code, but specify a preferred base address when loaded, and perform relocation if they cannot be loaded there. Forcing `kernel32.dll` to the preferred base address in every process avoids this relocation, and is likely done because of this optimization.

4.3.2 Linux

Linux calls its shared library objects Dynamic Shared Objects or DSOs (an overview is given in [Dre06]), which can be loaded dynamically using the `dlopen` function. This function is present in libdl, which normally is not linked in in most target processes. However, libdl is a simple wrapper library provided by glibc, and the real functions for managing DSOs are present in the core glibc library which is linked in in most normal processes.

Finding the location of functions in DSOs mapped in remote processes is more problematic than on Windows however, partly because of ASLR[†]. We can find the address of a function in a DSO mapped in a remote process in at least two ways, the first by examining the relevant memory map layout of a process through the `maps` file `/proc` filesystem, reading out the load address of the DSO, and adding to that the offset of the function we're looking for[‡]. This is a fair idea which works in practice, but requires the readability of the `/proc/[pid]/maps` file, which may not always be accessible depending on the security enforcement on the system.

Another method is outlined in [ano02], which resolves function addresses in loaded libraries independently of the dynamic linker. The full method is outlined in the phrack article, but in short the ELF header in the remote process[§] is used to locate the Global Offset Table or GOT, in which the second entry is the head of a linked list structure keeping track of all loaded libraries in the process. This linked list is in turn used to find the appropriate library description, and this description can be used to determine the exported symbols.

Libptrace currently implements the second approach, but there are situations where it could fail, such as where a custom C library and dynamic linker are used for an object or when a process purposefully changed its own ELF header in memory. We might like to implement the first approach as well to complement the second.

[†]Address Space Layout Randomization, where the locations of data sections such as the stack, the heap and loaded libraries are randomly positioned in the address space of a process as an additional security measure to prevent exploits from using predictable addresses for their purposes.

[‡]This offset can be determined in a number of ways, say by using `dlopen` and `dlsym` in our own process to load the DSO and the function address and subtracting the base address of the DSO in our process space.

[§]Note that the ELF header is mapped at a fixed location in memory, even when using ASLR.

A Libptrace

The latest development version of libptrace can be found by checking out the sourceforge SVN repository using the command:
`svn co https://libptrace.svn.sourceforge.net/svnroot/libptrace`

A.1 API Reference

The libptrace API reference is divided in several segments, depending on the availability of functions. The generic part of the library is functions which are present on the majority of platforms. This does not mean the function will always be available on every platform, as there might be reasons some functions cannot be implemented on some platforms, but they are intended to be as portable as possible.

The operating system specific part of the library contains functions which are specific to a certain operating system. Most often these functions will call operating system specific API functions or systemcalls.

The architecture specific part of the library contains functions which are specific to a certain architecture, for instance functions which modify specific registers by their name.

A.1.1 Data structures

Libptrace provides several datastructures, which we will not cover extensively, as most common operations are dealt by with accessor functions relying on these structures, but their content seldomly needs to be accessed directly by the programmer.

- `struct ptrace_context`
  A structure describing the state of libptrace when attached to a specific thread and the process it belongs to. This is not meant to be accessed directly, as it differs on all platforms.
- `struct ptrace_error`
  A structure outlining the last error that occured when executing a libptrace function. It is currently part of the `ptrace_context`, and described in appendix A.1.2.

- `struct ptrace_registers`
  `struct ptrace_fpu_registers`
  `struct ptrace_mmx_registers`
  `struct ptrace_sse_registers`
  A structure containing the set of registers common to a specific architecture. This can be used to request all registers at once, and contains members equal to the names of the registers. Optionally there may be structures available which describe the FPU, MMX and SSE registers of an architecture if applicable.

- `struct ptrace_altstack`
  A structure describing an alternative stack, and which defines its base address, its size, and the registers used for tracking it.

A.1.2 Error handling

Before presenting the functions in the libptrace API, it is convenient to understand how libptrace implements error handling. The library distinguishes between three different types of errors: those produced in the local process by the operating system API functions it calls, those libptrace will generate itself when encountering problems, and those produced in the remote process by the operating system API functions libptrace makes it call. The first variant is referred to as an external error, the second as an internal error, and the last as a remote error. Internally libptrace tracks these types in a structure which is part of the `ptrace_context` structure and updated for every libptrace library call using that context, but most often the library user will use the `ptrace_errmsg` functions for retrieving a textual representation of the error that occured.

In the API function reference presented below, all error reporting is done through the integer return value, where 0 implies success and -1 implies an error, unless specifically noted otherwise.

A.1.3 Generic functions

- `int ptrace_open(struct ptrace_context *pctx, ptrace_pid_t tid)`
  Opens a thread with TID `tid` and the process it belongs to for tracing and manipulation. Libptrace keeps state information in a `ptrace_context` structure, which `pctx` must point to. All threads in the process are suspended[†].

- `int ptrace_attach(struct ptrace_context *pctx, ptrace_pid_t tid)`
  An alias for `ptrace_open`.

- `int ptrace_close(struct ptrace_context *pctx)`
  Stops the tracing and manipulation of a thread and the process it belongs to which are described by `pctx`.

- `int ptrace_detach(struct ptrace_context *pctx)`
  An alias for `ptrace_close`.

- `int ptrace_write(struct ptrace_context *pctx, void *dst, const void *src, size_t len)`
  Writes `len` bytes of data from `src` in the local process to the memory address `dst` in the remote process described by `pctx`.

- `int ptrace_read(struct ptrace_context *pctx, void *dst, const void *src, size_t len)`
  Read `len` bytes of data from virtual memory address `src` in the remote process described by `pctx` to `dst` in the local process.

- `int ptrace_get_registers(struct ptrace_context *pctx, struct ptrace_registers *regs)`
  Read the CPU registers from the remote thread described by `pctx` into a ptrace_registers structure pointed to by `regs`.

- `int ptrace_get_registers(struct ptrace_context *pctx, struct ptrace_registers *regs)`
  Write the CPU registers described in the ptrace_registers structure pointed to by regs to the remote thread described by pctx.

- `int ptrace_wait_breakpoint(struct ptrace_context *pctx)`
  Resumes execution of all threads in the process described by `pctx` until a breakpoint event occurs in the thread described by `pctx`.

- `int ptrace_wait_breakpoint_at(struct ptrace_context *pctx, void *location)`
  Resumes execution of all threads in the process described by `pctx` until a breakpoint event at address `location` occurs in the thread described by `pctx`.

[†]On Linux libptrace currently only suspends the main thread. This will change in the future.

- `void *ptrace_malloc(struct ptrace_context *pctx, size_t len)`
  Allocates `len` bytes of memory in the target process described by `pctx`.
  Returns `NULL` on error, and a pointer to the memory area in the remote process otherwise.

- `int ptrace_free(struct ptrace_context *pctx, void *mem)`
  Frees previous allocated memory pointed to by `mem` in the target process described by `pctx`.

- `const char *ptrace_errmsg(struct ptrace_context *pctx)`
  Returns a pointer to an UTF-8 error message string, given the error kept in the context `pctx`.

- `const wchar_t *ptrace_errmsg16(struct ptrace_context *pctx)`
  Returns a pointer to an UTF-16 error message string, given the error kept in the context `pctx`.

- `int ptrace_get_pagesize(struct ptrace_context *pctx, int *page_size)`
  Retrieves the size of a memory page in bytes, and stores the result in `page_size`. The `pctx` argument is currently ignored.

- `ptrace_library_handle_t ptrace_library_load(struct ptrace_context *pctx, const char *library)`
  Loads a shared library object with filename `library` into the process space of the remote process described by `pctx`.
  Returns `NULL` on error, and a handle to the shared library in the remote process otherwise.

- `int ptrace_library_unload(struct ptrace_context *pctx, ptrace_library_handle_t handle)`
  Unloads a shared library object described by `handle` from the process space of the remote process described by `pctx`.

- `ptrace_function_ptr_t ptrace_library_get_function_addr( struct ptrace_context *pctx, ptrace_library_handle_t handle, const char *function)`
  Resolve the address of the function called `function` in the remote process described by `pctx`. The function must be present in the library described by `handle`.
  Returns `NULL` on error, and a pointer to the function in the remote process otherwise.

- `int ptrace_altstack_init(struct ptrace_context *pctx, struct ptrace_altstack *stack, size_t size)`
  Allocate an alternative stack of `size` bytes in the remote process descrived by `pctx`. Information regarding the alternative stack will be written to `stack`.

- `size_t ptrace_altstack_align(size_t size)`
  Returns the real size that `ptrace_altstack_init` will allocate for stacks when passed a size argument of `size`. This can vary on different systems; Windows for instance does not allow `kernel32.dll` function calls when the stack is improperly aligned.

- `int ptrace_altstack_current(struct ptrace_context *pctx, struct ptrace_altstack *stack)`
  Writes information about the altstack currently used by the remote thread described by `pctx` to `stack`.

- `int ptrace_altstack_switch(struct ptrace_context *pctx, struct ptrace_altstack *stack, struct ptrace_altstack *old_stack)`
  Makes the remote thread described by `pctx` use the alternative stack described by `stack`. If `old_stack` is unequal to `NULL`, the information about the previous altstack will be written there.

- `int ptrace_altstack_destroy(struct ptrace_context *pctx, struct ptrace_altstack *stack)`
  Destroys the altstack described by `stack` in the process described by `pctx`. The altstack may not be currently in use.

- `int ptrace_push16(struct ptrace_context *pctx, uint16_t word)`
  Pushes the 16 bit value `word` on the stack used by the remote thread described by `pctx`.

- `int ptrace_push32(struct ptrace_context *pctx, uint32_t dword)`
  Pushes the 32 bit value `dword` on the stack used by the remote thread described by `pctx`.

- `int ptrace_pop16(struct ptrace_context *pctx, uint16_t *word)`
  Pops a 16 bit value from the stack used by the remote thread described by `pctx`, and writes the value to `word`.

- `int ptrace_pop32(struct ptrace_context *pctx, uint32_t *dword)`
  Pops a 32 bit value from the stack used by the remote thread described by `pctx`, and writes the value to `dword`.

- `int prace_call_function(struct ptrace_context *pctx, void *code, int *retval)`
  Makes the remote thread described by `pctx` call a function at location `code` storing the return value in `retval`. The context of the thread is saved before the call, and restored afterwards.

- `int ptrace_call_procedure(struct ptrace_context *p, void *code)`
  Similar to `ptrace_call_function`, but omits storing the return value, because it is irrelevant or not present.

A.1.4 Linux specific functions

- `int ptrace_continue(struct ptrace_context *pctx)`
  Resumes execution of all threads in the process described by `pctx`.
- `int ptrace_continue_signal(struct ptrace_context *pctx, int signum)`
  Resumes execution of all threads in the process described by `pctx`, delivering signal `signum` to the process described by `pctx`.
- `int ptrace_wait_signal(struct ptrace_context *pctx, int signum)`
  Resumes execution of all threads in the process described by `pctx` until a signal `signum` is delivered to the process.
- `void *ptrace_mmap(struct ptrace_context *pctx, void *start, size_t length, int prot, int flags, int fd, off_t offset)`
  Makes the remote thread described by `pctx` perform the `mmap` systemcall. The rest of the parameters to this function are the regular parameters to the `mmap` systemcall.
  Returns `MAP_FAILED` on error, and a pointer to the mapped pages in the remote process otherwise.
- `int ptrace_munmap(struct ptrace_context *p, void *start, size_t length)`
  Makes the remote thread described by `pctx` perform the `munmap` systemcall. The rest of the parameters to this function are the regular parameters to the `munmap` systemcall.
- `int ptrace_get_orig_eax(struct ptrace_context *pctx, uint32_t *reg)`
  Stores the value of the special software register `orig_eax` in the thread described by `pctx` at the location specified by `reg`. This call is specific to the i386 version of Linux.
- `int ptrace_set_orig_eax(struct ptrace_context *pctx, uint32_t reg)`
  Sets the value of the special software register `orig_eax` in the thread described by `pctx` to the value `reg`. This call is specific to the i386 version of Linux.

A.1.5 Windows specific functions

- `HMODULE ptrace_load_library(struct ptrace_context *pctx, const char *dll)`
  Performs a `LoadLibrary` call with argument `dll` in the remote process described by `pctx`.
- `BOOL ptrace_free_library(struct ptrace_context *p, HMODULE dll)`
  Performs a `FreeLibrary` call with argument `dll` in the remote process described by `pctx`.

- `FARPROC ptrace_get_proc_address(struct ptrace_context *p, HMODULE dll, LPCSTR func)`
  Performs a `GetProcAddress` call with arguments `dll` and `func` in the remote process described by `pctx`.

A.1.6 i386 specific functions

- `int ptrace_get_eax(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_ebx(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_ecx(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_edx(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_esi(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_edi(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_eip(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_esp(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_ebp(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_eflags(struct ptrace_context *pctx, uint32_t *reg)`
  Stores the value of the named register in the thread described by `pctx` at the location specified by `reg`.

- `int ptrace_set_eax(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_ebx(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_ecx(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_edx(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_esi(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_edi(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_eip(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_esp(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_ebp(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_eflags(struct ptrace_context *pctx, uint32_t reg)`
  Sets the value of the named register in the thread described by `pctx` to the value `reg`.

- `int ptrace_get_cs(struct ptrace_context *pctx, uint16_t *reg)`
  `int ptrace_get_ds(struct ptrace_context *pctx, uint16_t *reg)`
  `int ptrace_get_es(struct ptrace_context *pctx, uint16_t *reg)`
  `int ptrace_get_fs(struct ptrace_context *pctx, uint16_t *reg)`
  `int ptrace_get_gs(struct ptrace_context *pctx, uint16_t *reg)`
  `int ptrace_get_ss(struct ptrace_context *pctx, uint16_t *reg)`
  Stores the value of the named segment register in the thread described by `pctx` at the location specified by `reg`.

- `int ptrace_set_cs(struct ptrace_context *pctx, uint16_t reg)`
  `int ptrace_set_ds(struct ptrace_context *pctx, uint16_t reg)`
  `int ptrace_set_es(struct ptrace_context *pctx, uint16_t reg)`
  `int ptrace_set_fs(struct ptrace_context *pctx, uint16_t reg)`
  `int ptrace_set_gs(struct ptrace_context *pctx, uint16_t reg)`
  `int ptrace_set_ss(struct ptrace_context *pctx, uint16_t reg)`
  Sets the value of the named segment register in the thread described by `pctx` to the value `reg`.

- `int ptrace_get_db0(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_db1(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_db2(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_db3(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_db6(struct ptrace_context *pctx, uint32_t *reg)`
  `int ptrace_get_db7(struct ptrace_context *pctx, uint32_t *reg)`
  Stores the value of the named debug register in the thread described by `pctx` at the location specified by `reg`.

- `int ptrace_set_db0(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_db1(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_db2(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_db3(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_db6(struct ptrace_context *pctx, uint32_t reg)`
  `int ptrace_set_db7(struct ptrace_context *pctx, uint32_t reg)`
  Sets the value of the named debug register in the thread described by `pctx` to the value `reg`.

- `int ptrace_get_mmx_registers(struct ptrace_context *pctx, struct ptrace_mmx_registers *reg)`
  Stores the value of the mmx registers in the thread described by `pctx` at the location specified by `reg`.

- `int ptrace_set_mmx_registers(struct ptrace_context *pctx, struct ptrace_mmx_registers *reg)`
  Sets the value of the mmx registers in the thread described by `pctx` to the values in `reg`.

- `int ptrace_get_sse_registers(struct ptrace_context *pctx, struct ptrace_sse_registers *reg)`
  Stores the value of the sse registers in the thread described by `pctx` at the location specified by `reg`.

- `int ptrace_set_sse_registers(struct ptrace_context *pctx, struct ptrace_sse_registers *reg)`
  Sets the value of the sse registers in the thread described by `pctx` to the values in `reg`.

References

[ano02] anonymous. Runtime process infection. *Phrack Magazine*, 2002.
URL `http://www.phrack.org/archives/59/p59-0x08.txt`

[App07] Apple Inc. *Mac OS X ABI Function Call Guide*, 2007.
URL `http://developer.apple.com/documentation/DeveloperTools/Conceptual/LowLevelABI/LowLevelABI.pdf`

[Bar00] Moshe Bar. Kernel korner: The linux process model. *Linux Journal*, page 24, 2000. ISSN 1075-3583.

[DBP99] Prasad Dabak, Milind Borate, and Sandeep Phadke. *Undocumented Windows NT*, chapter Local Procedure Call. M&T Books, 1999.
URL `http://www.windowsitlibrary.com/Content/356/08/toc.html`

[DM05] Ulrich Drepper and Ingo Molnar. *The Native POSIX Thread Library for Linux*. Red Hat Inc., 2005.
URL `http://people.redhat.com/drepper/nptl-design.pdf`

[Dre06] Ulrich Drepper. *How To Write Shared Libraries*. Red Hat Inc., 2006.
URL `http://people.redhat.com/drepper/dsohowto.pdf`

[Fal07] Nicolas Falliere. Windows anti-debug reference. *Securityfocus*, 2007.
URL `http://www.securityfocus.com/infocus/1893`

[Fei05] Antnio Feijo. Injlib - a library that implements remote code injection for all windows versions. *The Code Project*, 2005.
URL `http://www.codeproject.com/KB/library/InjLib.aspx`

[Kun03] Robert Kunster. Three ways to inject your code into another process. *The Code Project*, 2003.
URL `http://www.codeproject.com/KB/threads/winspy.aspx`

[MHAJ07] Michael Matz, Jan Hubicka, and Mark Mitchell Andreas Jaeger. *System V Application Binary Interface AMD64 Architecture Processor Supplement*, 2007.
URL `http://www.x86-64.org/documentation/abi-0.99.pdf`

[S02] Sandeep S. Process tracing using ptrace. *Linuxgazette*, 2002.
URL `http://linuxgazette.net/issue81/sandeep.html`