# Providing a GDB debug stub integrated into native\_client

## Overview

TL;DR: nacl64-gdb’s dependence on service\_runtime’s internal details isn’t good for maintainability in the long term, so eventually we want the native\_client codebase to provide a stable debugging interface by implementing a GDB debug stub. We could potentially implement a debug stub in trusted *or* untrusted code, since many of the interfaces we need for untrusted crash dumping are those needed by a debug stub.

## Introduction

Where are we today? The debugger we currently recommend to developers is nacl64-gdb (started by Evgeny Eltsin). This is a version of GDB that is patched to attach to sel\_ldr on x86-64 for Linux and Windows.

The nacl64-gdb approach is OK in the short term, but it has several problems:

* It depends on internal details of service\_runtime, such as the nacl\_user and nacl\_thread\_ids arrays and the NaClThreadContext struct. The dependencies are not documented, and this is not meant to be a stable interface anyway.
* It is not easily portable between architectures and operating systems.
* There are no automated tests for it, so if we change service\_runtime in a way that breaks nacl64-gdb, we won’t find out until later.
* There is little chance of upstreaming these changes, which makes it harder to update to newer versions of GDB.

In the longer term we want the native\_client codebase to provide a stable interface for debugging use, so that debuggers do not have to rely on internal details of native\_client. native\_client should provide a GDB debug stub, implementing GDB’s remote serial protocol (RSP).

## Debug stub primitives

A GDB debug stub needs to be able to do the following:

1. Suspend an untrusted thread; query its registers; resume the thread [**suspend\_thread**]
2. Catch a hardware exception; resume the thread with all registers restored [**sigreturn**]
3. Enumerate untrusted threads [**enum\_threads**]
4. Set breakpoints [**set\_breakpoint**]
5. Modify registers before resuming a thread. This includes:
   * Setting the x86 trap flag for single-stepping [**set\_trap\_flag**]
   * Setting general purpose registers for calling functions from GDB [**set\_registers**]

## Could we implement a GDB stub in untrusted code?

It is interesting to consider whether we could implement a debug stub in untrusted code. Some of the abilities required by a debug stub are abilities that we would like to expose to untrusted code for other reasons:

* **suspend\_thread**
  + suspend\_thread is necessary for implementing fully-featured **untrusted crash dumping**. In order to produce stack backtraces for all threads, a crash dumper needs the ability to suspend untrusted threads and get their register state.
    - This does not actually require a resume\_thread function.
  + suspend\_threadis also useful for finding GC roots in **multi-threaded garbage collection**. Currently, Mono finds threads’ GC roots by implementing an equivalent of suspend\_thread in user code: user code is instrumented to poll a variable which indicates whether the thread should voluntarily suspend itself.
    - If suspend\_thread coexists gracefully with NaCl syscalls, this should also remove the need for the GC hooks in libc (which wrap blocking syscalls). See below.
  + Similarly, suspend\_thread could be used to implement pthread’s **asynchronous thread cancellation** (see [pthread\_cancel()](http://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cancel.html)), as long as there is a set\_thread\_regs() function that can be used for forcing the thread to run a thread\_exit() function.
    - glibc normally implements this using asynchronous signals; see the discussion of emulating asynchronous signals below.
    - However, no-one has asked for pthread\_cancel() support.
* **sigreturn** *might* be useful for implementing page permission fixups, e.g. for implementing copy-on-write in user code, or for implementing write barriers for garbage collectors. If a write occurs on a read-only page, a fault handler can change the page permissions and resume execution at the instruction that faulted.
  + However, no-one has asked for this feature.
  + One can emulate this feature by emulating instructions in a bundle, or by relocating instructions to a dynamically-loaded block and executing them there.
  + A language runtime that wants to use page permissions for write barriers could place each memory write instruction at the start of a bundle, but this might bloat the generated code unnecessarily.

Advantages of an untrusted debug stub:

* It can simplify the code, and make it easier to write portable code.
  + service\_runtime provides an abstraction layer. While we can provide abstraction layers in trusted code, they will never be as clearly defined as the separation between trusted and untrusted code.
* It expands the number of people who can work on the debug stub, because it’s not TCB code.
* It helps us dogfood interfaces such as **suspend\_thread** that we eventually want to expose as stable interfaces.
* It reduces the chance that the debug stub opens up security holes.
* It increases the chance of being able to run a debugger like GDB in the browser in future.

However, running the debug stub in the same address space as the debuggee has some disadvantages:

* The debug stub’s data structures can be corrupted by the debuggee, so there are limits to the kinds of failures it can debug.
  + This is analogous to the problems with debugging the Linux kernel using the Linux kernel’s GDB stub.
* Untrusted code would be able to send malformed packets to GDB. If GDB is not sufficiently robust, it might be exploitable, and we don’t expect GDB to be sandboxed.
  + This only matters if you’re using GDB to debug someone else’s code. I would expect this to be a common use for NaCl GDB in the long term, though.

## Implementing debug stub primitives

### suspend\_thread

NaCl’s host OSes provide two different models for stopping a thread and querying its registers:

* **Thread suspension:**
  + Windows provides SuspendThread()/GetThreadState()/ResumeThread(). This suspends a thread without executing any code on that thread.
  + Mac OS X implements a similar model: suspend\_thread()/thread\_get\_state()/resume\_thread()
  + Linux provides this via ptrace(), but we should assume that ptrace() is not available to us inside the outer sandbox.
  + Note that this should coexist gracefully with syscalls: if a thread is suspended while executing a syscall, the syscall itself will not necessarily be suspended, but GetThreadState() should return the state of the user registers at the point where the thread entered the syscall.
  + The OS provides no mechanism for a thread to say “don’t suspend me” or “defer any suspension”. However, we implement one ourselves via NaClAppThreadSetSuspendState().
  + Windows provides SetThreadContext() but it does not provide **sigreturn**.
* **Thread interruption via asynchronous signals:**
  + On Linux, tgkill() sends an asynchronous signal to a thread. This interrupts a thread, forcing it to run a signal handler, which receives a copy of the register state at the point the thread was interrupted.
  + This can be used to emulate SuspendThread()/GetThreadState(). The signal handler can save the register state somewhere. It may need to use a condvar to indicate to a querying thread when the register state has been saved.
  + The OS provides a mechanism for a thread to say “defer suspension” by blocking the signal via its signal mask. However, this is expensive: Changing the signal mask requires a syscall.
    - In practice, NaCl would not use the signal mask. A NaCl suspend\_thread syscall would use NaClAppThreadSetSuspendState() to protect the NaCl syscall from being interrupted.
  + Async signals do not coexist very gracefully with syscalls:
    - An in-progress syscall will need to be aborted for the signal handler to run.
    - If a syscall cannot be interrupted, we have no way to query the register state at the point where the syscall was entered.
    - This does not matter if NaCl is *using* async signals to implement a NaCl suspend\_thread syscall. It just means that async signals are a poor interface for NaCl to provide.
  + The OS provides a **sigreturn** syscall for returning from the signal handler. This is a “restore all registers” syscall. Restoring all registers requires help from the kernel on x86.
  + Async signal handlers are hard to write: Only a small number of standard library functions are “async signal safe”.

To some extent, one model can emulate the other:

* Emulating SuspendThread() with asynchronous signals:
  + Thread 1: faux-SuspendThread() calls tgkill() to send a signal to thread 2.
  + Thread 2: signal handler runs:
    - it copies the saved register state to save area (or saves a pointer);
    - it calls pthread\_cond\_signal() to notify thread 1 that the register state is available (**however**, pthread\_cond\_signal() is not officially async-signal-safe);
    - it calls pthread\_cond\_wait() to wait for an event to say that the thread should be resumed;
    - it copies possibly-modified register state back to the signal frame and returns (which uses **sigreturn** internally).
  + Meanwhile, thread 1 might be doing this:
    - faux-GetThreadState(): calls pthread\_cond\_wait() to wait for the register state to be available.
    - faux-ResumeThread(): calls pthread\_cond\_signal() to tell the thread to resume.
  + This is complex and requires co-operation from the thread being suspended!
* Emulating asynchronous signals with SuspendThread():
  + Force a thread to suspend using SuspendThread().
  + Get register state using GetThreadState() and write the register state to the stack of the interrupted thread.
  + Use SetThreadContext()+ResumeThread() to force the thread to run a signal handler function.
  + The signal handler function may require help returning to the interrupted code if the OS doesn’t provide a **sigreturn** syscall.
  + Cygwin implements this. See [how-signals-work.txt](http://cygwin.com/cgi-bin/cvsweb.cgi/src/winsup/cygwin/how-signals-work.txt?rev=1.14&content-type=text/x-cvsweb-markup&cvsroot=src).
  + What happens if you apply this to a thread suspended in a syscall, assuming that is even allowed? When the syscall returns, does its return value in %eax/%rax/r0 overwrite the value set with SetThreadContext(), or vice-versa?
    - This issue goes away if syscall results are returned in memory rather than in registers.

Which model should NaCl provide to untrusted code? I think we should provide the SuspendThread() model. It will interact better with NaCl syscalls. Providing useful async signals to NaCl would require making some or all of NaCl’s blocking syscalls interruptible by async signals, which would be unnecessarily complex.

### enum\_threads

In trusted code, enumerating untrusted threads is straightforward. service\_runtime already maintains a list of threads. We already iterate through this list to suspend threads in service\_runtime/win/thread\_suspension.c.

In untrusted code, there is currently no interface for enumerating threads. There are three ways we could do this:

* Add a NaCl syscall for enumerating threads.
* Add a libpthread function for enumerating threads, and require libpthread to maintain a list of all current threads.
* Get the IRT to maintain a list of all current threads, based on thread\_create and thread\_exit syscalls. (This might be worthwhile if we put the debug stub into the IRT, for example.)

### set\_breakpoint

In service\_runtime, we can implement **set\_breakpoint** by writing to the dynamic code area.

Currently, the first executable loaded by sel\_ldr is treated as a special case. Its code is loaded into the static code area, not the dynamic code area. We should un-special-case this, and load it into the dynamic code area instead, so that setting breakpoints works the same for all code, on all host OSes.

Safety:

* A safe implementation will interact properly with dyncode\_create(), dyncode\_modify() and dyncode\_delete(). e.g. It would clear breakpoint records for addresses when they are unloaded with dyncode\_delete().
* An unsafe implementation wouldn’t bother with any of that.
* A safe implementation would be great to have in the long term, but isn’t essential for a first cut.

Note: The debug stub currently built into sel\_ldr does not support the “z” (set-breakpoint) command. GDB knows how to fall back to setting a breakpoint by modifying code when the debug stub does not support “z”. However, it’s not safe to allow arbitrary code modifications, so our debug stub should support “z”. (We might have an unsafe mode that allows arbitrary code modifications, but we should have a safe mode too.)

## nacl\_sigreturn: Resuming untrusted threads with registers modified

In general, untrusted code cannot jump to a dynamically-chosen address after restoring all general purpose registers, because the address to jump to must be stored in a register in order to be masked.

Therefore, restoring all registers requires some help from trusted code.

On x86-64, I think NaCl trusted code can use the IRET instruction to restore %rip and %rsp to untrusted-code values in one go, using values stored outside of the sandbox’s address space.

On x86-32, it is harder for NaCl trusted code to restore all registers:

* Ideally we would use the IRET instruction for this. The kernel can use IRET to restore %eip, %cs, %esp, %ss and %eflags all in one go. However, in non-privileged mode, on x86-32, IRET does not restore %esp and %ss.
* Linux provides a **sigreturn** syscall, which makes IRET’s ability to restore %esp and %ss available to user code.
* Windows does not provide a sigreturn syscall. However, it does allow setting all registers via SetThreadContext()/ResumeThread(). It is awkward to provide a sigreturn equivalent this way, though. If thread A wants to do a sigreturn, it must get a helper thread to suspend thread A and set its registers.
* Mac OS X provides both sigreturn and thread\_set\_state()/thread\_resume(), but both are unusable for our purposes. Both of these ignore the provided %cs and reset %cs back to the kernel’s default %cs value for userland programs. There does not appear to be a way to set registers selectively without resetting %cs.
* Workaround for Mac OS X, v1:
  + NaCl sets up a page that will contain register snapshots. It will be mapped twice: read-only in the sandbox’s address space, and read+write in trusted address space.
  + service\_runtime will ensure that only safe values for %eip are written to this snapshot.
  + The trampoline/springboard area is extended to contain a new instruction sequence that restores %eip and %esp (and others) in one go using IRET.
  + We will probably need one register snapshot area per thread.
  + We might as well use this implementation on all OSes for consistency. This would avoid the awkwardness with using SetThreadContext()/ResumeThread() on Windows.
* Workaround for Mac OS X, v2 (this occurred to me on 2012/06/26):
  + Change the %gs segment to point into trusted address space (inside NaClAppThread) instead of untrusted address space. %gs:0 still contains the user thread pointer. (%gs:4 can contain the IRT thread pointer; this is a separate optimisation for IRT TLS access.)
  + We expand the segment to be 12 bytes in size. %gs:8 will contain an instruction pointer value that has been previously vetted by the TCB, which we will use to restore %eip from.
  + We need to ensure that the %gs segment is read-only. One level of protection is to set the read-only flag in the segment’s LDT entry. Another level of protection is to ensure that the validator disallows writes using %gs (see <http://code.google.com/p/nativeclient/issues/detail?id=2250>).
  + Then “jmp \*%gs:8” will be a safe instruction for untrusted code to use for jumping to a pre-vetted address without the address having to be stored in a temporary register.
  + In practice, we won’t want the validator to allow “jmp \*%gs:8”, and we will put this instruction into a code sequence in the trampoline area.

## How a trusted-code debug stub’s fault handler gets called

We currently use different mechanisms for handling faults on each host OS because each has a different threading model:

* Windows: A separate process handles the fault, using the Windows debug API.
* Mac: A separate thread handles the fault, using Mach exception handling.
* Linux: The thread that produces the fault handles it, using POSIX signal handling.

The three don’t share much code: NaCl doesn’t have a common trusted-code handler that is used across Windows, Mac and Linux. That’s partly because none was needed. It’s also because using a common handler would have added unnecessary overhead on Linux and Windows. (On Linux and Windows we can transfer control directly back to untrusted code using sigreturn()/SetThreadContext(). On Mac, though, we have to go via a small trusted-code helper routine.)

However, if we are going to provide a trusted debug stub, the fault handlers will need to transfer control to a common handler in the debug stub. There are two ways we could do this:

* **Same-thread**: The debug stub’s handler runs on the thread that faulted (in the style of POSIX signal handling).
* **Separate-thread**: The debug stub runs an event loop in one thread and receives events about faults (in the style of Mach exception handling).

The same-thread model is easiest to implement:

* Same-thread is the model we already use for untrusted fault handlers.
* The trusted fault handler can run on the same stack used for NaCl syscalls (NaClAppThread’s sys.stack\_ptr). We can copy the register state onto the trusted stack; no need to allocate space for it in NaClAppThread. The fault handler is effectively a NaCl syscall, and should be uninterruptible in the same way.
* For a breakpoint handler to resume from a breakpoint, the handler will need to use nacl\_sigreturn (see above).

## Chromium integration

I propose the following:

* Chromium can be configured to launch an external **debug helper** every time a NaCl process is started.
* The debug helper is started as a new process each time. The new process is created with a socket descriptor that is connected to sel\_ldr’s debug stub.
* The debug helper will be launched the same on Linux, Mac and Windows.
* sel\_ldr begins in a “stopped” state.
  + This appears to be a requirement of the GDB RSP protocol. At least, GDB assumes that the process is in a stopped state when it connects, and it expects the “?” query to work. The Linux kernel’s debug stub follows this by stopping the kernel when the debug stub is enabled, even though it is very inconvenient for a kernel to be stopped.
  + We might want to tweak the protocol and/or GDB to remove this requirement.
* The debug helper should also receive some metadata, such as the NMF URL and the filename of the IRT library.

This means that using TCP ports will not be built into Chromium’s NaCl debug support. We don’t have to worry about allocating TCP port numbers.

The debug helper *can* just be GDB. We would have to change (or wrap) GDB so that it can receive the RSP socket at startup rather than requiring a command like “target remote localhost:4014”. Stock GDB has some support for [non-TCP connections](http://sourceware.org/gdb/current/onlinedocs/gdb/Connecting.html), e.g. “target remote /proc/self/fd/N” or “target remote | *command*”.

However, it would be possible to have more sophisticated wrappers around GDB to support these use cases:

* **Debug-on-crash**: The developer wants the GDB UI to pop up only when their program crashes.
  + Implementation: When the debug helper is launched, it immediately tells sel\_ldr to resume. When an untrusted crash occurs, the debug helper launches GDB for debugging the crash.
* **Attaching to NaCl processes interactively**: The developer wants to choose a NaCl process to debug from a list of currently-running processes.
  + Implementation: When the debug helper is launched, it immediately tells sel\_ldr to resume; it then sends the RSP socket to a daemon process and exits. The daemon process keeps a list of currently running NaCl processes and their RSP sockets. The user can list these processes and attach GDB to one of the processes (which will retrieve the RSP socket from the daemon).
* **Debug-from-startup**: Sometimes the developer wants to inspect what happens during the NaCl process’s startup, before it crashes (if it crashes at all).
  + This is supported by the simplest debug helper that runs GDB directly. However, we want this use case to coexist with others, such as debug-on-crash. Switching between debug-on-crash and debug-from-startup should not require the developer to restart Chromium or change Chromium’s configuration to use a different debug helper.
  + Implementation: When the debug helper is launched, it checks a configuration setting to see whether it’s in “debug-on-crash” or “debug-from-startup” mode and acts accordingly. There is a UI for the user to change this mode.

## Testing

Debugging support can be tested at multiple levels:

* Testing GDB itself: The appropriate level for doing this is using GDB’s Machine Interface, [GDB/MI](http://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html).
* Testing the debug stub at the RSP level. See native\_client/tests/debug\_stub for an example of this.
* Testing the interfaces that the debug stub uses.
  + For an untrusted debug stub, this would involve testing the NaCl syscalls that the debug stub uses.
  + For a trusted debug stub, this would involve testing internal interfaces; see thread\_suspension\_test.cc for an example.

We would also like to test the Chromium integration, but this only requires a small test to check that the RSP socket gets passed through correctly. We would not need to test the full debug stub functionality through Chromium.

## Providing a custom build of GDB

While a stock Linux GDB might be usable with a NaCl debug stub, we will need to provide GDB for Windows and Mac ourselves. (I expect the stock GDB for Mac is built for Mach-O and not ELF, and if there were a stock GDB for Windows it would be built for PE and not ELF.)

There are two ways we could provide GDB:

* As a trusted executable built for Linux, Mac or Windows. We’ve already built a GDB for Windows for use with NaCl debug stubs.
* As a .nexe. This is worth considering and might not be very difficult.
  + gdb.nexe would only need access to an inherited RSP socket descriptor, plus (read-only?) filesystem access for reading debug info.
  + We would only need to ship two builds (x86-32 and x86-64) rather than three.
  + In the long term we might be able to run GDB in the web browser, perhaps as part of dev tools or as a Chromium extension.

## GDB customisations

We might need the following changes to GDB:

* Provide a portable equivalent of “target remote /proc/self/fd/N”. See “Chromium integration” above.
* Handle return addresses on nacl-x86-64. Return addresses on the stack will have had the r15 base address added to them. I suspect GDB will need to be changed to ignore the top 32 bits so that it does not get confused.
  + All the addresses that GDB deals with should be <4GB. While we can make the debug stub’s read-memory and write-memory operations truncate the addresses they receive to 32 bits, that is a hack.
  + @Evgeny: I expect nacl64-gdb already addresses this?
* GDB seems to be quite slow when debugging via RSP. There might be some fixes we can make to speed it up.
  + For example, “disassemble” reads memory byte by byte. This involves many round trips to the debug stub. Even so, “disassemble” seems unreasonably slow; I wouldn’t expect the round trip time to be as slow as GDB appears.
  + However, I’m not using a stock GDB. (“gdb --version” says “7.3.1-gg4”.)

## GDB RSP multi-process mode

I don’t expect we will want to use the RSP protocol’s multi-process mode. NaCl’s debug stub implementation will be running inside the sel\_ldr process and will handle only a single NaCl process. I don’t know of any benefits that we’d get by using multi-process mode.

## Appendix: History of NaCl debugging support

* nacl-gdb
  + Patches to gdb to support x86-32 NaCl on Linux.
  + Developed by Doug Evans.
  + Developed before 2009?
  + Used to be built as part of the NaCl GNU toolchain. There is still some support for building gdb in tools/Makefile, but it’s not used.
  + There is a test case under tools/tests/hello\_world\_gdb.\*, but it was flaky and got disabled.
  + It allowed debugging trusted code too. There was a command for switching between trusted and untrusted modes.
* Original built-in GDB RSP debug stub for NaCl
  + Built into sel\_ldr and Chromium.
  + It lives in src/trusted/debug\_stub and src/trusted/gdb\_rsp
  + Developed by Ian Lewis and Noel Allen in 2010.
  + Developed in a branch of NaCl and later committed to the NaCl tree.
  + Originally for x86-64 Windows only. Later partially ported to Linux for x86-64 and x86-32 (in 2011).
  + It originally had no tests, but Mark Seaborn added a test (in 2011), which lives under tests/debug\_stub.
  + For a long time, it was not compiled in by default, until it was cleaned up to hook in via dependency injection (in 2011).
  + Some integration with Chromium: There is a Chromium flag for enabling the debug stub, but it has been broken because the test case for it has been disabled. sel\_main\_chrome.c no longer enables nacl\_signal when the Chromium flag is passed through.
* Out-of-process GDB RSP debug stub
  + It lives in the native\_client\_sdk repo.
  + Developed by Mike Linck and Alexei Garianov in 2011.
  + It supports Windows only: definitely x86-32, and maybe x86-64.
  + I’m not sure whether it has any tests.
  + Some hooks were added to service\_runtime for exporting some internal details via debug messages.
  + I think it relied on some of service\_runtime’s internal data structures. I’m not sure of the exact details.
  + It was intended to be used with a client embedded in MSVS, which has not been finished.
  + Integration with Chromium: none?
* nacl64-gdb
  + A patched version of GDB, similar to the original nacl-gdb.
  + Developed by Evgeny Eltsin starting around April 2010.
  + It supports x86-64 only. Originally implemented for Linux, and then extended to Windows in late 2011.
  + It supports debugging both trusted and untrusted code.
  + No automated test suite?
  + Some changes were made to service\_runtime to export symbols. I’m not sure exactly what internal details of service\_runtime nacl64-gdb relies on. It probably assumes the layout of NaClThreadContext.
  + Integration with Chromium: a --nacl-gdb option for launching GDB. Work is ongoing.