## NaCl: three kinds of crash handling

## Overview

There are three things that must work together in trusted code:

* untrusted exception handling
* trusted crash reporting (Breakpad)
* RSP server (debug stub)

## Hardware exception handling

Windows:

* Cover both x86-32 and x86-64
* Hook up in Chromium too. Which process acts as the debugger?
* This will probably just work with Breakpad
* Making this work with an out-of-process RSP server is harder...
* Performance: Using Windows debugging means that all threads in the NaCl process will be stopped every time a thread is created. This is potentially quite a serious performance problem.
  + However, for comparison, we already pause all untrusted threads during mmap()/munmap() calls. In nacl-glibc, pthread\_create() uses mmap() to allocate a stack (unless it’s reusing a previously-allocated one). So there is a precedent for this *kind* of performance hit, though obviously we don’t want to add to it.
  + We might want to attach the debugger process only after an exception handler has been registered, so that we don’t pay the cost for this feature unless it’s used. But this still creates a dilemma for developers -- speed or crash reporting?
  + We could preallocate and/or reuse threads. But this creates a tuning problem: how many threads to keep around?
  + This would mean we wouldn’t want to use the same approach on x86-64.

Mac:

* This probably has to work via Mach exception handling, since Mach exception handlers run before Unix signal handlers, and Mac Breakpad registers a Mach exception handler.
* Make this work with an RSP server...

Linux:

* Not too hard
* Breakpad is not hooked up yet for NaCl

Isuses for a public ABI:

* Do we dump registers in a signal frame?
  + To remove the need for assembly code (and hence be PNaCl-friendly), any register dumping should be done by the IRT or the TCB, and not in user code.
  + Should the register dump be padded to a fixed size across architectures to be PNaCl-friendly, as jmp\_buf is? This shouldn’t be necessary, because user code does not allocate the register dump.
* How to handle crashes inside a signal handler
* Stack unwinding across signal frame: backtraces, C++ exceptions, pthread\_exit()
* Do we provide POSIX-like signal numbers and a signal mask?

Use cases:

* Turning NULL pointer dereferences into language-level exceptions, e.g. for Mono. This does not require returning to the instruction where the exception occurred.
  + The same applies for numeric exceptions (floating point and division by zero).
* Page permission fixups. e.g. If a write occurs to a read-only page, the page permissions are changed to be read+write, and the fault handler returns to the faulting instruction.
  + Requires some NaCl equivalent to sigreturn(). We would need to re-validate the target bundle to check the address.
* Crash reporting. Unlike the other use cases, we never need to return to the crashed thread. Full exception handling is unnecessary for crash reporting; a snapshot of the process would suffice.

## Asynchronous signals

On Unix, in-process asynchronous signals have these use cases:

* Pausing threads for GC. On Linux, I believe Mono pauses threads by sending them an asynchronous signal. This interrupts a thread and causes it to run a signal handler. The signal handler can make the saved register state available to the GC and wait for the GC to finish. Unlike Windows’ SuspendThread() interface, signals do not pause threads per se. They just provide an interface on top of which thread pausing can be implemented.
* POSIX asynchronous thread cancellation. On Linux, glibc’s libpthread uses asynchronous signals to implement asynchronous thread cancellation. This does not work on NaCl currently.

It’s not clear that asynchronous signals provide any benefit over a SuspendThread() interface.

It’s a small mercy that Windows does not have asynchronous signals, because it eliminates a class of ways in which things could go wrong.

Asynchronous signals can be implemented in terms of SuspendThread(), particularly in a system without strong thread identity, like NaCl and unlike Linux. What I mean by that is:

* On Linux, threads have identity/state: The gettid() syscall returns a unique thread ID. Threads have a CLEARTID location associated with them, which is set up by clone() (and set\_tid\_address()) and assigned to on thread exit.
* On NaCl, the only thread identity is TLS state, which one thread can assume on behalf of another.

## RSP server (debug stub)

Let's assume we have a separation between RSP-server and RSP-client. (The alternative, a non-RSP GDB, is considered in another section below.)

Questions:  
Who maintains the RSP server?  
How is it versioned? Supplied with chromium?  
Is it maintained in the NaCl repo?  
  
Ideally the RSP server is:

* supplied with Chromium, or at least versioned in sync (e.g. via component installer)
* small
* maintained in the NaCl repo
* There is no need to stabilise service\_runtime's internal data structures.

Features:

* set breakpoints
* set memory watchpoints?
* single-step threads
* enumerate threads
* get a notification when a thread gets a hardware exception
* get register state
  + get saved register state for a thread that is inside a NaCl syscall
* modify register state
* pause and resume threads
  + if the thread is inside a NaCl syscall, it should be paused when the syscall returns
* read and write memory
  + should be scoped to within the sandbox’s address space
* call a function in untrusted code
  + This should be covered by “modify register state” and “resume threads”, though I’m not sure exactly how GDB implements calling functions. Presumably GDB can catch the function return by setting the return address to some location on which a breakpoint has been set.

User interface: Choices of how to start up the RSP server:

* Run separately, attach to an existing NaCl process
* Pass an option to Chromium

Security considerations:

* Ideally, untrusted code that is running while a debugger is attached should not be able to escape the sandbox.
  + It would be undesirable if using specific debugger features (e.g. setting breakpoints) broke the sandbox. It would be worse if the sandbox became broken as soon as the debugger were attached.
  + This is not a “must have” requirement, because we can state that “debugging may make the NaCl sandbox insecure”. However, in the long term we can expect that people will attach the debugger to programs they didn’t create, and we can expect that some developers may create anti-debugger code that does bad things.
* Ideally, an RSP client should not be able to break the NaCl sandbox through its use of the RSP protocol.
* Enabling debugging should not, by default, give machines on the local network the ability to debug the NaCl process. If the debug stub listens on a TCP socket it should listen on localhost by default.
  + An unsafe default here would be particularly bad if the debug stub allowed escaping from the sandbox.

## Untrusted crash reporting

Tracing stacks and adding symbols:

* What happens when the IRT has frames on the stack? This includes:
  + crashes in system calls
  + crashes in PPAPI callbacks
* Just make sure IRT uses a frame pointer?
* Online stack tracing, i.e. inside the web app:
  + Do we have runtime unwind info? Would need an ABI for getting this for the IRT.
  + Do we have symbols for the IRT? Takes space in chrome download. How would this info be acquired?
* Offline tracing, e.g. on a server, from a crash dump uploaded by a web app:
  + Seems easier. IRT versions can easily be archived.
  + This is how Breakpad works. It can be awkward to use during Chromium development because the symbols for Chromium’s executables and libraries have to be saved somewhere.
* How does Linux’s vdso handle this?

Recording other threads:

* Breakpad for Linux/Mac/Windows can record the state of other threads at the time of a crash.
* Implementing this for NaCl would involve adding interfaces for:
  + enumerating threads (could be done in untrusted libpthread)
  + getting other threads’ register state (requires TCB extension)
  + finding the extent of a thread’s stack, based on registers (libpthread’s responsibility)
  + ideally, stopping the threads so that they don’t interfere with the crash recording (requires TCB extension)
* Note that pausing threads is something we opted \*not\* to do for GC support.
* Note that this overlaps with what an RSP server would support.

Who does untrusted crash reporting? There is a spectrum of options:

* system service vs. standard library
  + A standard library means a library linked in by the web app. This provides more flexibility for the web app author. It would be easier for such a library to interact with the untrusted libpthread, since they are versioned together.
  + A system service means something built into the IRT, sel\_ldr or the trusted plugin.
* on by default vs. enabled via an API call
  + “On by default” only makes sense for a system service.
  + “Enabled via an API call” allows specifying a destination for crash dump uploading.
* upload to Google server vs. upload to web app’s server
  + Uploading to the web app’s server requires no new trusted code, and provides more flexibility, but requires more server-side configuration for the web app.
  + Uploading to a Google server ties NaCl more to Chrome.
* upload dump via existing PPAPI HTTP support vs. via a specialised interface
  + If the dump is uploaded via the existing PPAPI interfaces, the upload will be terminated if the web page is closed.
  + How much of the NaCl untrusted PPAPI process is likely to work if we’ve just crashed?
  + A specialised interface could queue the dump to be uploaded in the background.

What format should we use for untrusted crash dumps?

* One option is the minidump format, which Breakpad uses. (The format was invented by Microsoft, I believe.)
* Another option might be the ELF core dump format.

## Debugger

The debugger will need a way to get IRT debug info (unwind info, symbols, etc.)

## Alternative: Debugging without RSP

The original nacl-gdb port (for x86-32 Linux) and the newer nacl64-gdb port (for x86-64 Linux) allow debugging a sel\_ldr process directly without using GDB’s Remote Serial Protocol.

Advantages:

* Allows debugging both trusted and untrusted code. However, this has a limited audience.
* Might have lower performance overhead, but we haven’t measured it.

Disadvantages:

* GDB-specific. It does not help with writing other debugging tools - for example, execution trace tools (see <http://lackingrhoticity.blogspot.com/2009/05/really-simple-tracing-debugger-part-2.html>).
* So far, it has been Linux-specific
* It typically requires GDB to know about service\_runtime’s internal data structures
  + It needs to discover the location of untrusted address space (relatively easy).
  + It would need to distinguish between trusted and untrusted crashes. On x86-32, this involves discovering the trusted value of %cs (nacl\_global\_cs, NaClGetGlobalCs()).
  + It would need a way to enumerate untrusted threads. Would it read through NaClApp.threads -> NaClAppThread.thread (non-trivial)? Or would it enumerate all host-OS threads and, for each, determine whether it hosts a NaCl untrusted thread (also non-trivial)?
  + If a thread is in a NaCl syscall, we would want to get a backtrace for untrusted code. This involves finding the register state at the point the syscall was made. This involves reading NaClAppThread.user.
* Making this work with untrusted exception handling on Windows would be difficult
  + GDB would need to duplicate service\_runtime’s logic for bouncing exceptions back to untrusted code.
  + This might be difficult for Mac OS X as well, given that service\_runtime’s untrusted hardware exception handling will have to use Mach exception handling rather than signals.
* Hard to test
  + If we are going to make this work on 5 configurations -- Windows (32-bit and 64-bit), Linux (32-bit and 64-bit) and Mac -- automated testing is essential.
  + If we need to keep GDB working with multiple versions of service\_runtime, this makes automated testing essential too.
  + We had a test for the original nacl-gdb (x86-32 Linux) in native\_client/tools/tests, but it was disabled for being flaky. This does not bode well.

Questions:

* If a NaCl GDB were ported to Windows and Mac, would it be able to handle trusted-code debugging? Would GDB be able to handle a Mach-O (Mac) or PE (Windows) ABI at the same time as ELF (NaCl)?

It’s relatively easy to get a basic NaCl non-RSP GDB working quickly such that it gives a backtrace on an untrusted crash. However, it’s a lot harder to get it to do the right thing in 90% of the cases (e.g. enumerating threads, getting untrusted backtraces inside syscalls), and doing so creates tricky maintenance problems.

Conclusion: A non-RSP GDB should not be our preferred way of doing debugging in the long term.

## Use case: Reporting stack overflows

Suppose a NaCl app crashes because it runs out of stack. This happens on a thread created with pthread\_create() (i.e. not on the initial thread).

For comparison, Linux Breakpad does not handle this, because its signal handler will run on the same stack and so fail. Linux Breakpad attempts to register a sigaltstack, but it does so only for the thread on which Breakpad was initialised. Linux glibc’s libpthread does not provide a way to hook into pthread\_create()/pthread\_exit() to register sigaltstacks for new threads. An application would have to manually register a sigaltstack in a new thread for Breakpad to work on stack exhaustion for these threads.

Reporting on stack exhaustion is important because it is one of the cases in which NaCl programs will not behave the same across architectures, even with PNaCl. Stack frame sizes will be different between x86-32, x86-64 and ARM.

Possible solutions:

* Extend the untrusted libpthread with hooks to allow registering and freeing sigaltstacks on thread creation and exit.
  + This would consume a lot of extra address space, which is limited on x86-32 and ARM. It would also slow down thread creation and thread exit.
* Allow crashes to be handled by a separate process or separate thread.
* Introduce the concept of an exclusive-use signal stack for the whole process: The first thread to crash gets use of the signal stack. Subsequent threads to crash get suspended until the first thread releases the signal stack.
  + Easy to implement.
  + Rather than making this process-global, untrusted code could specify a mutex descriptor when registering the signal stack.
  + Does not require cluttering up address space or slowing down the thread creation path.
* When a thread crashes, suspend all other threads. Suspending other threads is desirable for a crash reporter anyway: the reporter will want to snapshot their register state, and it’s best to suspend them as soon as possible. If the TCB suspends threads automatically, threads can be suspended sooner.

## Use case: Reporting out-of-memory crashes

Suppose a NaCl app crashes because it has run out of address space and malloc() or mmap() has returned an error. This is more likely to happen on x86-32 or ARM systems since NaCl provides only 1GB of address space there.

How do we upload a crash report in this case? If the crash reporter runs in the same process, we will have problems with any dynamic memory allocation it does. The PPAPI proxy in the IRT will be particularly troublesome because it expects to do dynamic allocation outside the control of user code.

Possible solutions:

* Do out-of-process untrusted crash reporting.
* Carefully write the PPAPI proxy to do no dynamic allocation for the case we care about.
* Put the crash dump uploading behind a NaCl syscall so that it does no dynamic allocation in untrusted address space.
  + This does not help if a crash occurs because we ran out of trusted address space.

## Use case: Finding bugs in web apps in the field

This is the main use case for untrusted crash reporting. I’m listing it for completeness.

## Use case: Finding bugs in the IRT in the field

If the IRT contains bugs which cause crashes, ideally we (as developers of NaCl and Chromium) want to get reports of these crashes. In particular, the untrusted PPAPI proxy is the most complex part of the IRT and so the most likely to contain bugs.

The difficulty here is that there can be no automated way to distinguish crashes caused by the IRT and crashes caused by user code, because there is no hard boundary between the two.

If a developer receives crash reports for their app that they believe are due to IRT bugs, they can forward the crash reports to us. Having a standard library and standard format for NaCl untrusted crash reports will make this easier. If we notice a crash is common across NaCl apps, this will suggest an IRT bug or a bug in a standard library or a common mistake.