Panpan

An expandable framework for the stimulation of software system activity

Daniel U. Thibault
DRDC – Valcartier Research Centre

Defence Research and Development Canada

Reference Document
DRDC-RDDC-2017-D060
July 2017

IMPORTANT INFORMATIVE STATEMENTS

Work conducted on behalf of the Real-Time Monitoring (RTM) work-stream of the Platform-to-Assembly Secure Systems (PASS) project.

© Her Majesty the Queen in Right of Canada, as represented by the Minister of National Defence, 2017

© Sa Majesté la Reine (en droit du Canada), telle que représentée par le ministre de la Défense nationale, 2017

Abstract

One of the goals of the Real-Time Monitoring (RTM) work-stream of the Platform-to-Assembly Secure Systems (PASS) project is to develop new advanced software system normality models for advanced online anomaly detection. Some of those models need to be trained using system traces captured while the monitored system is running normally. The need for a tool that could generate predictable and relatively simple system activity was identified. *Panpan* is such a tool: a modular application software framework that provides a simple means of stimulating normal activity in a controllable fashion.

Résumé

Un des buts du volet de travail Real-Time Monitoring (RTM, Surveillance en temps réel) du projet Platform-to-Assembly Secure Systems (PASS, Systèmes sécurisés de la plate-forme à l'assemblage) est de développer de nouveaux modèles avancés de normalité des systèmes logiciels pour la détection sophistiquée en temps réel d'anomalies. Certains de ces modèles doivent être entraînés à l'aide de traces capturées tandis que le système à surveiller exécute des activités considérées normales. Le besoin d'un outil pouvant générer des activités prévisibles et relativement simples a été identifié. *Panpan* est cet outil, un cadre d'application logicielle modulaire fournissant un moyen simple de stimuler l'activité normale de façon contrôlable.

Table of contents

List of figures

This page intentionally left blank.

1 Introduction

1.1 Overall description

One of the goals of the Real-Time Monitoring (RTM) work-stream of the Platform-to-Assembly Secure Systems (PASS) project is to develop new advanced software system normality models for advanced online anomaly detection. Some of those models need to be trained using system traces captured while the monitored system is running normally.

Panpan[1] is a modular application software framework that provides a simple means of stimulating normal activity in a controllable fashion. The main application executes a script consisting of a sequence of commands, each command invoking a module designed to use a set of system calls and standard C libraries in a predictable fashion. Ancillary applications provide simple services necessary for various multi-application activity patterns.

Panpan is written in C and should be portable across any Linux-based system. It is a framework in the sense that its headers provide macros that simplify the addition of new modules (implementing *commands*) as needed.

1.2 How to read this document

This document is intended as an introduction to users and would-be developers. A user who needs to capture traces should probably start with Chapters 2 and 5, and then use Annex A to guide the choice of commands to include in the *Panpan* script or scripts, consulting Chapter 3 as necessary. Chapter 4 is only necessary if the `bounce` command (simulating socket inter-process communication (IPC)) is used.

A developer who wants to write additional commands for *Panpan* should read the document in its entirety before turning to the source code. The latter includes a fair level of documentation.

1.3 Architecture

The main application, `panpan`, parses a script file, recognising the individual commands contained therein. Each command is then executed by *Panpan*'s library (shared object), `libpanpan`. Adding a new command implies only a simple recompilation of the library.

Besides the usual slew of support libraries present on any Linux system, *Panpan* uses the `curl`, `popt` and `config` libraries. Thus it has as prerequisites `libcurl4-openssl-dev`, `libpopt-dev` and `libconfig-dev` (Debian systems) or `libcurl-devel`, `popt-devel` and `libconfig-devel` (RPM systems).

[1] *Pépinot* was a pioneering French Canadian children's television puppet show that originally ran from 1954 until 1957 and reran several times over the next fifteen years. The titular hero's nemesis was *Panpan*, a villain whose disguises and schemes never availed him. Since the software doesn't actually ever do anything directly useful, this was thought fitting.

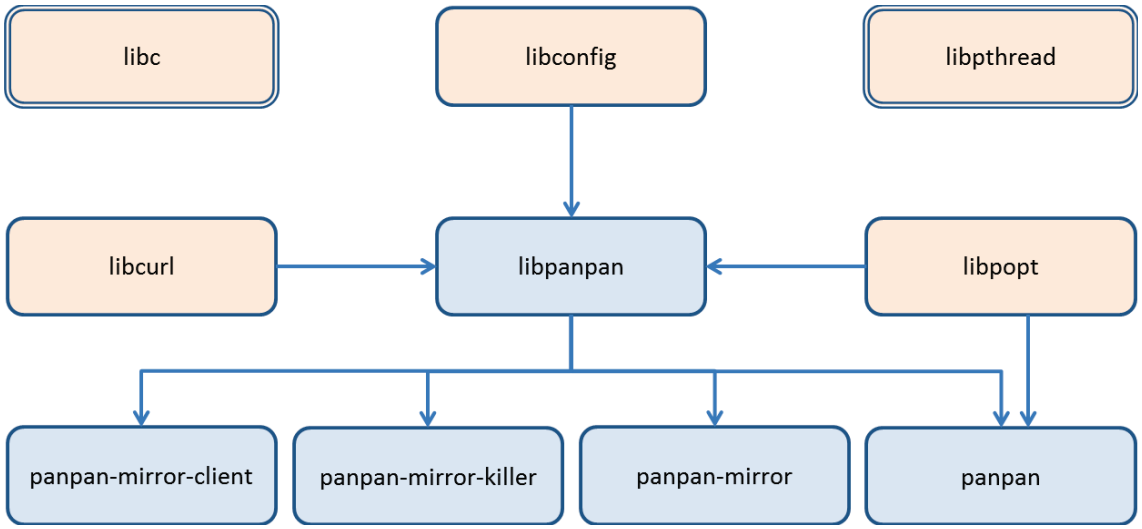

Figure 1: *Dependency diagram for the* Panpan *software suite. The* `libc` *and* `libpthread` *libraries are used by all;* `libpopt` *and* `libconfig` *have no further dependencies, while* `libcurl` *is the root of an extensive tree of dependencies.*

There are currently three ancillary applications: `panpan-mirror`, `panpan-mirror-killer`, and `panpan-mirror-client`; more could be added as the need arises. These support stimulation of the Unix domain or IPC (Inter-Process Communication) sockets. This was thought simpler and somewhat less 'incestuous' than relying on additional command modules to launch and shut down a separate process to communicate with using IPC sockets. The `panpan-mirror-client` ancillary application is not even necessary for *Panpan*; it is supplied merely to allow a user to test out the `panpan-mirror` manually.

2 The `panpan` main application

This chapter describes the *Panpan* main application, `panpan`, following a format similar to that used by the `man` pages.

2.1 Synopsis

```
panpan [OPTIONS] SCRIPT
```

`SCRIPT` is the path to the `panpan` script to execute. There are just two options; the first is `h`, `help`, and the second is `t`, `thread`.

2.1.1 `h`, `help` option

The `help` option displays the usual version information and instructions giving the usage syntax, the parameter and its meaning, the options and their meaning, but also the instructions specific to each script command available.

```
$ panpan --help
panpan 0.30 lib 0.30
Usage: panpan [OPTIONS] SCRIPT
  SCRIPT                   Path to the script to execute.
                           Empty lines and lines beginning with '#' are ignored.
                           The possible COMMANDs are listed below.
  -h, --help               This help message.
  -t, --thread             Run the script commands in a separate thread.

  COMMAND
  append FILE STRING       Appends STRING to FILE.
  bounce FILE ELIF         Bounces each line of FILE against the panpan-mirror
                           daemon, which must already be running. The result is
                           stored in ELIF. Prints a count of the number of
                           lines mirrored.
  count_letters [-n N | --nthreads N] FILE
                           Prints a count of the number of occurrences of the
                           letters a-z in FILE. The count is computed using N
                           threads to scan the FILE in parallel (default: 13).
                           FILE can be an http[s]:// ftp:// or file:// URL.
  exec COMMAND             Invokes the shell on the COMMAND.
                           The COMMAND need note be quoted or escaped.
  fibonacci INTEGER        Computes the INTEGERth Fibonacci number.
                           The zeroth one is 0, the first is 1.
  hello                    Prints a welcoming message.
  scan [-d | --directory] PATH
                           Prints the number of newlines in PATH (a file).
                           PATH can be an http[s]:// ftp:// or file:// URL.
                           If PATH is local and d/directory is specified, each
                           file in the PATH directory is scanned using a
                           separate thread.
  sum INTEGER              Computes the sum of the first INTEGER positive
                           integers.
  exit                     Terminates the script.
```

2.1.2 `t`, `thread` option

The `thread` option wraps each command module call in a simple thread which is launched and then rejoined to recover the command's return value (an `error_struct`, see 2.3). It is meant to allow a simple script to add minimal thread-handling activity.

For instance, a one-line script invoking the `hello` command merely prints a "`Hello…\n`" line to `stdout`. Running the same script with the `thread` option and comparing the two traces would allow one to identify the basic system thread-handling activity pattern.

Note that because `panpan` waits for each thread to complete before moving on to the next command, the linearity of the script is maintained: the outputs of one command will have completed before the next command starts. If `panpan` merely launched each command in a separate thread, the sequence of outputs to `stdout` would interweave all the commands together in an essentially unpredictable pattern.

2.2 Example

```
$ panpan ~/test.script
```

Where `test.script` could be something as simple as this:

```
# Say hello
hello
```

Panpan expects the script to be a text file. The stream is broken up by new line characters ('`\n`'); each empty line is ignored, and so is any line beginning with '`#`'. Otherwise, the first word of the command is matched against the inventory of commands implemented by `libpanpan`. *Panpan* also has a single built-in command, `exit`, which serves to terminate the script (although seemingly redundant as the last line of a script, it is useful during debugging to make *Panpan* ignore the following lines of a script). If a match is found, the entire line is sent to that command module as a 'command line' (except in the `exit` case). It is up to the module to parse that command line as it sees fit. If an error occurs, script processing is aborted.

Because of its comment convention, *Panpan* is shebang-compatible, meaning a script such as the one above can be made executable by turning its executable attribute on and adding a leading shebang line that points to the `panpan` binary. For example:

```
#!/usr/bin/panpan
# Say hello
hello
```

Where `/usr/bin/panpan` could be a symbolic link to the actual `panpan` binary. Then the script could be directly invoked from the command line:

```
$ ~/shebang.script
```

We will conclude this topic with a more complete example script that demonstrates all of the current *Panpan* modules:

```
#!/usr/bin/panpan
# Say hello
hello

# Exec a command line
exec ls -l ./src/*.script

# Compute some Fibonacci values
fibonacci 0
fibonacci 10
fibonacci 100

# Compute some sums
sum 0
sum 10
sum 100

# Create a file (outer script should rm the file)
append -n /home/user/testoutput "This is bunk."

# Bounce a file (panpan-mirror must be running)
bounce /home/user/some.file /home/user/elif.emos

# Scan a directory
scan -d /home/user/Documents

# Scan a big file
scan /home/user/big.file

# Scan remote files
scan http://ftp.us.debian.org/debian/pool/main/z/zzzeeksphinx/
scan https://drdc-rddc.gc.ca/
scan ftp://speedtest.tele2.net/512KB.zip

# Scan a local file as if it were remote
scan file:///home/user/big.file

# Count letters
count_letters --nthreads=10 /home/user/big.file

exit
```

2.3 Internal error handling

Because C is notoriously weak and inconsistent in its error handling, all parts of *Panpan* make use of a home-brewed error-handling solution. The `error_struct` structure is meant to be created explicitly and then passed along to any invoked functions. It detects normal error statuses (it cannot do anything about access violations and the like, unfortunately) and allows the *Panpan* components to react appropriately. Once it has outlived its usefulness, the `error_struct` must be destroyed explicitly.

The `error_struct` subsumes the `errno` mechanism and can augment system-supplied error messages with additional custom informative strings.

However, a *Panpan* module (command) could be written to be deliberately incorrect, even to the extent of knowingly exposing a vulnerability. Tracing a system being "exploited" in a known fashion should prove as instructive as tracing one behaving safely and normally.

2.4 Configuration file

Panpan uses the `config` library to look up global configuration parameters. The configuration signature (e.g., `mirror.socket-name`) is first sought in the current user directory (`~/.panpan`); if not found, the global configuration file is used (`/etc/panpan.conf`); if this also fails, a default value is used. Here is a sample `~/.panpan` file:

```
# Panpan configuration file
# Names begin with a letter (A-Z, a-z) or an asterisk (*) and continue with
# alphanumeric characters (A-Z, a-z, 0-9), dashes (-), underscores (_), and
# asterisks (*).
# String values can include the following escape sequences: \" \\ \f \n \r \t
# { group } ( list ) [ array ]

/* (string) conf-version == "1.0" */
conf-version = "1.0";

mirror : {
  socket-name = "~/.panpan-mirror";
  socket-shutdown = "~/.panpan-shutdown";
};

application : {
  exit-sleep-time = 0;
};
```

3 The `panpan` commands

This chapter describes each of the *Panpan* commands, following a format similar to that used by the `man` pages. They are presented in rough order of increasing complexity.

3.1 The `exit` command

3.1.1 Synopsis

```
exit
```

The `exit` built-in command ignores any options or arguments in its command line and merely terminates script parsing. The main application terminates shortly after.

The configuration value `application.exit-sleep-time` (defaulting to zero) specifies how many seconds the main application should pause (using the `sleep()` library function) before issuing its final instruction of `pthread_exit`.

3.1.2 Output

```
exit:
   Terminating script
```

3.2 The `hello` command

3.2.1 Synopsis

```
hello
```

The `hello` command is *Panpan*'s "Hello world!". It ignores any options or arguments in its command line and merely prints "`Hello from libpanpan.\n`" to `stdout`.

3.2.2 Output

```
hello...
   Hello from libpanpan.
```

3.3 The `sum` command

3.3.1 Synopsis

```
sum INTEGER
```

The `sum` command computes the sum of the first `INTEGER` positive integers (using a dumb accumulator loop) and prints this value to `stdout`. If `INTEGER` is absent, not an integer, or a negative integer, an error occurs. If `INTEGER` is `65536` or larger, the result will be incorrect because of integer rollover. Any additional arguments are ignored unless they begin with a hyphen (‘`-`’), which triggers an “`Unknown option`” error from `popt`.

3.3.2 Output

```
sum...
   sum <INTEGER> is <INTEGER>
```

3.4 The `fibonacci` command

3.4.1 Synopsis

```
fibonacci INTEGER
```

The `fibonacci` command computes the `INTEGER`th Fibonacci number and prints this value to `stdout`. The zeroth number is zero and the first is one; the module uses a loop instead of a recursive approach. If `INTEGER` is absent, not an integer, or a negative integer, an error occurs. If `INTEGER` is `47` or larger, the result will be incorrect because of integer rollover. Any additional arguments are ignored unless they begin with a hyphen (‘`-`’), which triggers an “`Unknown option`” error from `popt`.

3.4.2 Output

```
fibonacci...
   fibonacci <INTEGER> is <INTEGER>
```

3.5 The `exec` command

3.5.1 Synopsis

```
exec [COMMAND]
```

The `exec` command is *Panpan*'s open-ended extension mechanism. It invokes the `system()` library function on `COMMAND` (the remainder of its command line) and prints the return code to `stdout`. See the Linux Programmer's manual for details of the `system()` library function.

3.5.2 Output

```
exec <COMMAND>
<COMMAND output, if any>
   return code is <INTEGER>
```

3.6 The `append` command

3.6.1 Synopsis

```
append FILE STRING
```

The `append` command appends `STRING` to the specified `FILE`. Any spaces in the `FILE` or `STRING` must be escaped or the `FILE` and/or `STRING` quoted. Any additional arguments are ignored unless they begin with a hyphen ('`-`'), which triggers an "`Unknown option`" error from `popt` (except for the '`-n`', '`--newline`' option).

The `FILE` must be a true path: it may contain '.' and '..' but not an initial '~' (expansion of '~' is a shell function, not part of the file system's path specification). This restriction could be lifted in the future.

The `append` command is meant to stimulate simple file access activity (open, seek, write, close).

3.6.2 `n`, `newline` option

The `newline` option instructs `append` to first write a newline ('`\n`') to the `FILE`, and only then append the `STRING`.

3.6.3 Output

```
append '<STRING>' to '<FILENAME>'...
   append '<STRING>' to '<FILENAME>': done
```

3.7 The `scan` command

3.7.1 Synopsis

```
scan [OPTION] PATH
```

The `scan` command scans the `PATH` (a file, local or remote, unless the `directory` option is specified) and reports how many newlines (‘`\n`’) were found in it. Any spaces in the `PATH` must be escaped or the `PATH` quoted. Any additional arguments are ignored unless they begin with a hyphen (‘`-`’), which triggers an “`Unknown option`” error from `popt` (except for the ‘`-d`’, ‘`--directory`’ option).

The `PATH` can be a file system path or a `file:/`, `http:/`, `https:/`, or `ftp:/` URL (Uniform Resource Locator). Currently, the `https:/` support is rudimentary and specific to the DRDC – Valcartier research Centre configuration: it relies on `DREnet-Web-Proxy-Root-CA.crt` as its sole certificate authority root certificate.

The `PATH` must be a true path: it may contain ‘.’ and ‘..’ but not an initial ‘~’ (expansion of ‘~’ is a shell function, not part of the file system’s path specification).

The `scan` command is meant to stimulate network access activity or detached thread activity as well as buffer allocation and de-allocation activity. Several of its functions are reused by the `count_letters` command.

3.7.2 `d, directory` option

The `directory` option instructs `scan` to treat the `PATH` as a directory and to scan each file in it using a separate detached thread. If the `PATH` is not local, the option is ignored.

3.7.3 Output

```
scan '<PATH>'...
   scan '<PATH>': 2000 lines
scan 'ftp://speedtest.tele2.net/512KB.zip'...
   scan 'ftp://speedtest.tele2.net/512KB.zip': 86 lines

scan '/home/user/Notes'...
scan '/home/user/.cp'...
   scan '/home/user/Notes': 99 lines
scan '/home/user/.project'...
   scan '/home/user/.project': 28 lines
   scan '/home/user/.cp': 99 lines
scan '/home/user/More Notes'...
   scan '/home/user/More Notes': 10 lines
```

The last example (a directory scan) shows how the output of separate threads can interweave in a not-necessarily-sequential fashion: the `.project` command (line 4), although started after the `.cp` command (line 2), completes (line 5) before the latter (line 6).

3.8 The `count_letters` command

3.8.1 Synopsis

```
count_letters [OPTION] FILE
```

The `count_letters` command scans the `FILE` (a file, local or remote) and reports how many times each of the lower-case letters '`a`' through '`z`' occur in it. Any spaces in the `FILE` must be escaped or the `FILE` quoted. Any additional arguments are ignored unless they begin with a hyphen ('`-`'), which triggers an "`Unknown option`" error from `popt` (except for the '`-n`', '`--nthreads`' option).

The `FILE` can be a file system path or a `file:/`, `http:/`, `https:/`, or `ftp:/` URL (Uniform Resource Locator). Currently, the `https:/` support is rudimentary and specific to the DRDC – Valcartier Research Centre configuration: it relies on `DREnet-Web-Proxy-Root-CA.crt` as its sole certificate authority root certificate.

The `FILE` must be a true path: it may contain '.' and '..' but not an initial '~' (expansion of '~' is a shell function, not part of the file system's path specification).

The `count_letters` command is meant to stimulate complex thread management activity. The command thread acts as master and creates a number of detached worker threads. Using four condition-mutex-variable synchronisation barriers, the master obtains a buffer's worth of data from the `FILE`, lets the worker threads compete to scan the buffer for each of the letters, and then moves on to the next buffer. This is repeated until the `FILE` has been completely processed. The barriers act in pairs to effect a sort of handshake synchronisation between the master and the workers, and there are two pairs in order to implement a two-stroke cycle. The array of accumulating letter counts also protects each of its cells with individual mutexes. A number-of-running-threads semaphore is used in an inverted fashion (with an active wait) to protect the synchronisation structures against premature tear-down during the `count_letters` command's clean up.

3.8.2 `n, nthreads` N option

The `nthreads` option expects an argument (`N`) and specifies how many worker threads `count_letters` should create (the default is 13).

3.8.3 Output

```
count_letters '<FILE>'...
   count_letters[a]: 47
   count_letters[b]: 15
[…]
   count_letters[y]: 5
   count_letters[z]: 2
```

3.9 The `bounce` command

3.9.1 Synopsis

```
bounce FILE ELIF
```

The `bounce` command reads the `FILE`, bounces the contents of each of its lines off the panpan-mirror daemon, and stores the resulting mirrored lines in `ELIF`. Any spaces in the `FILE` or `ELIF` must be escaped or the `FILE` and/or `ELIF` quoted. Any additional arguments are ignored unless they begin with a hyphen ('`-`'), which triggers an "`Unknown option`" error from `popt`.

`FILE` and `ELIF` must be true paths: they may contain '.' and '..' but not an initial '~' (expansion of '~' is a shell function, not part of the file system's path specification).

The mirror socket's name is supplied by the configuration value `mirror.socket-name`; that of the killing socket by `mirror.socket-shutdown`. They default to `~/.panpan-mirror` and `~/.panpan-shutdown`, respectively. The `bounce` command does not use the killing socket at all, nor does it try to launch the mirror daemon if it isn't running: these operations are the responsibility of the user (or of the script running *Panpan* itself).

The `bounce` command is meant to stimulate simple IPC socket activity. The command creates a socket, connects to it, writes to it, reads from it, shuts it down and closes it for *each* line it needs mirrored.

3.9.2 Output

```
bounce '<FILE>' to '<ELIF>'...
   bounce '<FILE>' to '<ELIF>': <N> lines mirrored
```

4 The ancillary `panpan` applications

This chapter describes each of the *Panpan* ancillary applications, following a format similar to that used by the `man` pages. The ancillary applications serve as foils for *Panpan* to interact with in a known fashion. There are three such applications, meant to give *Panpan* a partner to hold a socket-based conversation with. As future modules are added, more ancillary applications may be needed such as, for example, an application to hold a shared-memory conversation with.

Care should be taken to keep these applications named `panpan-*` to facilitate their extraction from traces.

4.1 `panpan-mirror`

4.1.1 Synopsis

```
panpan-mirror [SOCKET_NAME [KILLING_SOCKET_NAME]]
```

The `panpan-mirror` ancillary application launches a detached thread that services connections on its "mirror socket" and then waits for a connection on its "killing socket." Any spaces in the `SOCKET_NAME` or `KILLING_SOCKET_NAME` must be escaped (alternately, the arguments can be quoted). The `panpan-mirror` ancillary application does not recognise options at all. Its behaviour is very daemon-like, although the application is not yet explicitly daemonised.

The mirror socket's name is supplied by the command line (`SOCKET_NAME`); if no argument was specified, then the configuration value `mirror.socket-name` is obtained from the local configuration file (`~/.panpan`); if that fails, then it is obtained from the global configuration file (`/etc/panpan.conf`); if that fails too, the default name of '`~/.panpan-mirror`' is used.

The killing socket's name is supplied by the command line (`KILLING_SOCKET_NAME`); if no argument was specified, then the configuration value `mirror.socket-shutdown` is obtained from the local configuration file (`~/.panpan`); if that fails, then it is obtained from the global configuration file (`/etc/panpan.conf`); if that fails too, the default name of '`~/.panpan-shutdown`' is used.

The `SOCKET_NAME` and `KILLING_SOCKET_NAME` can contain '.' and '..' as well as an initial '~'.

When a connection to the mirror socket occurs, it is serviced by reading the socket's buffer, mirroring it (e.g., 'abcd' becomes 'dcba'), writing the mirrored buffer back to it, and then shutting it down and closing it before accepting new connections.

When a connection to the killing socket occurs, the mirror socket is shut down and closed, the killing socket is also shut down and closed, the file system socket "files" are deleted, and the application and its detached child exit.

4.1.2 Output

`Panpan-mirror` has no output.

4.2 `panpan-mirror-killer`

4.2.1 Synopsis

```
panpan-mirror-killer [KILLING_SOCKET_NAME]
```

The `panpan-mirror-killer` ancillary application serves to send a killing signal to `panpan-mirror` by merely connecting to its "killing socket." `panpan-mirror-killer` immediately shuts its connection down, closes it, and exits. Any spaces in the `KILLING_SOCKET_NAME` must be escaped or the `KILLING_SOCKET_NAME` quoted. The `panpan-mirror-killer` ancillary application does not recognise options at all.

The killing socket's name is supplied by the command line (`KILLING_SOCKET_NAME`); if no argument was specified, then the configuration value `mirror.socket-shutdown` is obtained from the local configuration file (`~/.panpan`); if that fails, then it is obtained from the global configuration file (`/etc/panpan.conf`); if that fails too, the default name of '`~/.panpan-shutdown`' is used.

The `KILLING_SOCKET_NAME` must be a true path: it may contain '.' and '..' but not an initial '~' (expansion of '~' is a shell function, not part of the file system's path specification).

4.2.2 Output

```
panpan-mirror should shut down now
```

4.3 `panpan-mirror-client`

4.3.1 Synopsis

```
panpan-mirror-client [SOCKET_NAME]
```

The `panpan-mirror-client` ancillary application is a very simple demonstration client that sends a line of text to the `panpan-mirror` "daemon" and prints the answer it got back. Any spaces in the `SOCKET_NAME` must be escaped or the `SOCKET_NAME` quoted. The `panpan-mirror-client` ancillary application does not recognise options at all.

The mirror socket's name is supplied by the command line (`SOCKET_NAME`); if no argument was specified, then the configuration value `mirror.socket-name` is obtained from the local configuration file (`~/.panpan`); if that fails, then it is obtained from the global configuration file (`/etc/panpan.conf`); if that fails too, the default name of '`~/.panpan-mirror`' is used.

The SOCKET_NAME must be a true path: it may contain '.' and '..' but not an initial '~' (expansion of '~' is a shell function, not part of the file system's path specification).

4.3.2 Output

```
Please enter your message:
This is the line I typed
The return message was
depyt I enil eht si sihT
```

5 Example of use for trace capture

This chapter describes how *Panpan* was installed and exploited to generate "model traces" for anomaly detection algorithm training.

5.1 Preparation

On the target Linux system, deploy a suitable Eclipse environment, such as Kepler (we used Eclipse 4.3.2, obtained from `eclipse.org` as the archive `eclipse-cpp-kepler-SR2-(4.3.2)-linux-gtk-x86_64.tar.gz`). Copy the *Panpan* source code and import the projects into Eclipse, configuring them as appropriate. Compile (build) the *Panpan* components. It is easiest to do this once on one system and then copy the entire Eclipse project directory structure to the other systems. Alternately, it should be possible to copy just the *Panpan* executables to the target systems, as long as the required support packages (`popt`, `config`, `curl`) are installed there.

5.1.1 Detailed steps

To go more into detail, the archive `eclipse-cpp-*.tar.gz` is copied to the virtual machine's `/home/<user>/` directory (using a shared directory provided by the hypervisor, or a virtual CD-ROM image mounted on the virtual machine, or an `ssh` connection using `scp`, etc.) and deployed in place (e.g., to `/home/<user>/eclipse`). Launching Eclipse can be facilitated by creating a symbolic link to the executable in `/usr/bin` (e.g., '`# ln -s /home/<user>/eclipse/eclipse /usr/bin/eclipse-kepler`').

The source for *Panpan* is then copied into the virtual machine (using similar pathways) and deployed to `/home/<user>/eclipse-4.3.2-c/panpan`, for example, as well as its sibling folders `panpanlib`, `panpan-mirror`, `error_handler`, etc.

Launch Eclipse, indicate `/home/<user>/eclipse-4.3.2-c/` as the workspace path, and go to the Workbench. The projects (`panpan` and its siblings) will be imported one after the other, starting with `panpanlib`—which is the keystone of the *Panpan* suite.

5.1.1.1 `panpanlib`

Using `File: New: C Project`, select `Shared Library: Empty project` as your `Project type`. Pick a `Toolchain`, such as `Linux GCC`. Name the project `panpanlib`. You can safely ignore the "`Directory with specified name already exists`" warning. Click `Next`. The `Release` configuration should be enough on its own. Click `Finish`. Because the source code was already in place (as the warning reminded us), it shows up in the project without needing to import it. Sadly, this roundabout approach is required because just doing `File: Import: C/C++: Existing Code as Makefile Project` offers no way of turning the project into a shared library or auto-generating its `make` files. Now request the `Project: Properties`.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Compiler: Miscellaneous and check the Position Independent Code (-fPIC) box. Accept your way back to the Project Properties window.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Libraries and Add popt, config, and curl to the Libraries (-l) box.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Miscellaneous and add -pthread to the Linker flags text box.

Navigate to C/C++ Build: Settings: Build Artifact and change the Artifact name from ${Projname} to panpan (without this change, the library would be named libpanpanlib.so).

Navigate to C/C++ General: Paths and Symbols: Includes for GNU C, and click Add. Click File System and browse to /home/<user>/eclipse-4.3.2-c/error_handler.

So that libpanpan.so can be found by panpan and the other applications, create a symbolic link in /usr/lib, like so:

```
# ln s /home/<user>/eclipse-4.3.2-c/panpanlib/Release/libpanpan.so /usr/lib/libpanpan.so
```

5.1.1.2 panpan

Using File: New: C Project, select Executable: Empty project as your Project type. Pick a Toolchain, such as Linux GCC. Name the project panpan. You can safely ignore the "Directory with specified name already exists" warning. Click Next. The Release configuration should be enough on its own. Click Finish. Because the source code was already in place (as the warning reminded us), it shows up in the project without needing to import it. Now request the Project: Properties.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Libraries and Add popt, config, and panpan to the Libraries (-l) box. You can omit config from this list: although it is expected that the panpan application will use libconfig at some point in its future, this is not the case as yet.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Miscellaneous and add -pthread to the Linker flags text box.

Navigate to C/C++ General: Paths and Symbols: Includes for GNU C, and click Add. Click File System and browse to /home/<user>/eclipse-4.3.2-c/error_handler. Click Add a second time, check the Is a workspace path box, then Workspace and browse to panpanlib/src. Note that you could also import error_handler.h as an error_handler project and then include it as a workspace path for all the *Panpan* projects.

The panpan project should build at this point, building panpanlib at the same time. You may need to invoke Index: Rebuild on the project beforehand.

5.1.1.3 panpan-mirror

Using File: New: C Project, select Executable: Empty project as your Project type. Pick a Toolchain, such as Linux GCC. Name the project panpan-mirror. You can safely ignore the "Directory with specified name already exists" warning. Click Next. The Release configuration should be enough on its own. Click Finish. Because the source code was already in place (as the warning reminded us), it shows up in the project without needing to import it. Now request the Project: Properties.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Libraries and Add panpan to the Libraries (-l) box.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Miscellaneous and add -pthread to the Linker flags text box.

Navigate to C/C++ General: Paths and Symbols: Includes for GNU C, and click Add. Click File System and browse to /home/<user>/eclipse-4.3.2-c/error_handler. Click Add a second time, check the Is a workspace path box, then Workspace and browse to panpanlib/src.

The panpan-mirror project should build at this point. You may need to invoke Index: Rebuild on the project beforehand.

5.1.1.4 panpan-mirror-killer

Using File: New: C Project, select Executable: Empty project as your Project type. Pick a Toolchain, such as Linux GCC. Name the project panpan-mirror-killer. You can safely ignore the "Directory with specified name already exists" warning. Click Next. The Release configuration should be enough on its own. Click Finish. Because the source code was already in place (as the warning reminded us), it shows up in the project without needing to import it. Now request the Project: Properties.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Libraries and Add panpan to the Libraries (-l) box.

Navigate to C/C++ General: Paths and Symbols: Includes for GNU C, and click Add. Click File System and browse to /home/<user>/eclipse-4.3.2-c/error_handler. Click Add a second time, check the Is a workspace path box, then Workspace and browse to panpanlib/src.

The panpan-mirror-killer project should build at this point. You may need to invoke Index: Rebuild on the project beforehand.

5.1.1.5 panpan-mirror-client

Using File: New: C Project, select Executable: Empty project as your Project type. Pick a Toolchain, such as Linux GCC. Name the project panpan-mirror-client. You can safely

ignore the "Directory with specified name already exists" warning. Click Next. The Debug configuration is more appropriate than Release for this application. Click Finish. Because the source code was already in place (as the warning reminded us), it shows up in the project without needing to import it. Now request the Project: Properties.

Navigate to C/C++ Build: Settings: Tool Settings: GCC C Linker: Libraries and Add panpan to the Libraries (-l) box.

Navigate to C/C++ General: Paths and Symbols: Includes for GNU C, and click Add. Click File System and browse to /home/<user>/eclipse-4.3.2-c/error_handler. Click Add a second time, check the Is a workspace path box, then Workspace and browse to panpanlib/src.

The panpan-mirror-client project should build at this point. You may need to invoke Index: Rebuild on the project beforehand.

5.2 Execution

Trace capture will be handled by LTTng, although other methods such as strace are possible. A trace template will be generated (creating the LTTng channels, enabling the kernel system call events as desired, setting the trace storage path) and saved using LTTng's save facility. A trace can then be captured using a simple shell script such as this one:

```
#!/bin/bash
lttng load panpan-session
lttng start
/home/<user>/eclipse-workspace/panpan/Release/panpan /home/<user>/panpan.script
lttng destroy
```

The specific system call activity desired is specified in the panpan.script, which could be something as simple as:

```
scan /home/<user>/sample.big.file
exit
```

If we were interested in just the mmap system call (with some context thrown in), for instance, the /home/<user>/.lttng/panpan-session.lttng file (used to resolve the 'lttng load panpan-session' command) could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<sessions>
  <session>
    <name>panpan-session</name>
    <domains>
      <domain>
        <type>KERNEL</type>
        <buffer_type>GLOBAL</buffer_type>
        <channels>
          <channel>
            <name>mmap</name>
            <enabled>true</enabled>
            <overwrite_mode>DISCARD</overwrite_mode>
            <subbuffer_size>33554432</subbuffer_size>
            <subbuffer_count>2</subbuffer_count>
            <switch_timer_interval>0</switch_timer_interval>
            <read_timer_interval>200000</read_timer_interval>
            <output_type>SPLICE</output_type>
            <tracefile_size>0</tracefile_size>
            <tracefile_count>0</tracefile_count>
            <live_timer_interval>0</live_timer_interval>
            <events>
              <event>
                <name>mmap</name>
                <enabled>true</enabled>
                <type>SYSCALL</type>
              </event>
            </events>
            <contexts>
              <context>
                <type>PID</type>
              </context>
              <context>
                <type>PPID</type>
              </context>
              <context>
                <type>TID</type>
              </context>
            </contexts>
          </channel>
        </channels>
        <trackers/>
      </domain>
    </domains>
    <started>false</started>
    <output>
      <consumer_output>
        <enabled>true</enabled>
        <destination>
          <path>/home/user/ltraces/panpan-session</path>
        </destination>
      </consumer_output>
    </output>
  </session>
</sessions>
```

Annex A Inventory of system calls

This annex inventories the system calls that were observed when tracing *Panpan*. Future development will likely aim at adding command modules that stimulate the remaining system calls. Current Linux 64-bit kernels typically have a little over 300 system calls (not counting the `compat_` family of thunking system calls, and considering each `entry/exit` pair as a single occurrence), some of which are extremely specialised.

A.1 Baseline

Invocation of *Panpan* with minimal activity (such as the `hello`, `sum`, or `fibonacci` commands) will generate the following system calls (listing only the entry events for conciseness):

```
syscall_entry_access
syscall_entry_brk
syscall_entry_close
syscall_entry_execve
syscall_entry_exit_group
syscall_entry_futex
syscall_entry_getrlimit
syscall_entry_ioctl
syscall_entry_mmap
syscall_entry_mprotect
syscall_entry_munmap
syscall_entry_newfstat
syscall_entry_newstat
syscall_entry_open
syscall_entry_read
syscall_entry_rt_sigaction
syscall_entry_rt_sigprocmask
syscall_entry_set_robust_list
syscall_entry_set_tid_address
syscall_entry_unknown
syscall_entry_write
```

Only those system calls that can be unambiguously attributed to *Panpan* are listed. The very first one is `execve`, generated by the shell when it launches the `panpan` application. The `syscall_entry_unknown` system call is a label used by LTTng to identify system calls whose names do not appear in the system's kernel headers. This is normally a transient problem which fixes itself as the kernel headers are updated to "catch up" to the kernel.

A.2 Threaded baseline

Invocation of *Panpan*'s `hello`, `sum`, and `fibonacci` commands in threaded mode (see 2.1.2) adds these three system calls:

```
syscall_entry_clone
syscall_entry_exit
syscall_entry_madvise
```

A.3 Exec

Invocation of *Panpan*'s `exec` command may add a wide variety of system calls depending on what it invokes. For this reason, no inventory has been attempted.

A.4 Append

Invocation of *Panpan*'s `append` command adds just one system call:

```
syscall_entry_lseek
```

A.5 Scan file

Invocation of *Panpan*'s `scan` command adds nothing when used on a local file (using either a local path or `file://` prefix). When used on a remote file (`http://` prefix), these nine system calls appear:

```
syscall_entry_connect
syscall_entry_fcntl
syscall_entry_getpeername
syscall_entry_getsockname
syscall_entry_getsockopt
syscall_entry_poll
syscall_entry_recvfrom
syscall_entry_sendto
syscall_entry_socket
```

A.6 Scan directory

Invocation of *Panpan*'s `scan` command for a directory launches separate threads for each file in the directory; this adds these 29 system calls:

```
syscall_entry_accept
syscall_entry_clone
syscall_entry_connect
syscall_entry_epoll_wait †
syscall_entry_exit
syscall_entry_fadvise64 †
syscall_entry_fcntl
syscall_entry_getdents †
syscall_entry_getegid †
syscall_entry_geteuid †
syscall_entry_getgid †
syscall_entry_getgroups †
syscall_entry_getuid †
syscall_entry_lseek
syscall_entry_madvise
syscall_entry_newstat
syscall_entry_openat †
syscall_entry_poll
syscall_entry_recvmsg †
syscall_entry_select †
syscall_entry_sendmsg †
syscall_entry_setitimer †
syscall_entry_setsockopt †
syscall_entry_shutdown
syscall_entry_socket
syscall_entry_splice †
syscall_entry_sync_file_range †
syscall_entry_wait4 †
syscall_entry_writev †
```

The system calls marked with a dagger (†) are exclusive to this *Panpan* command so far. It is remarkable that this relatively simple multi-threaded command should turn out to make use of more system calls than the `count_letters` command (see A.7), which has a much more complex internal control flow (making use of mutexes and semaphores).

A.7 Count letters

Invocation of *Panpan*'s `count_letters` command on a local file adds these 4 system calls:

```
syscall_entry_clone
syscall_entry_exit
syscall_entry_madvise
syscall_entry_nanosleep
```

The collection of system calls is independent of the number of threads used.

Invocation of *Panpan*'s `count_letters` command on a remote file combines the above system calls with the `scan` command's additional nine (see A.5).

A.8 Bounce

Invocation of *Panpan*'s `bounce` command adds these 12 system calls (spread between `panpan`, `panpan-mirror`, and `panpan-mirror-killer`):

```
syscall_entry_accept
syscall_entry_bind †
syscall_entry_clone
syscall_entry_connect
syscall_entry_exit
syscall_entry_getsockopt
syscall_entry_listen †
syscall_entry_madvise
syscall_entry_shutdown
syscall_entry_socket
syscall_entry_tgkill †
syscall_entry_unlink †
```

The system calls marked with a dagger (†) are exclusive to this *Panpan* command so far.

List of symbols/abbreviations/acronyms/initialisms

DRDC	Defence Research and Development Canada
IPC	Inter-Process Communication
LTTng	Linux Trace Toolkit Next Generation
PASS	Platform-to-Assembly Secure Systems
RTM	Real-Time Monitoring
URL	Uniform Resource Locator

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security markings for the title, abstract and indexing annotation must be entered when the document is Classified or Designated)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g., Centre sponsoring a contractor's report, or tasking agency, are entered in Section 8.)

DRDC – Valcartier Research Centre
Defence Research and Development Canada
2459 route de la Bravoure
Quebec (Quebec) G3J 1X5
Canada

2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.)

UNCLASSIFIED

2b. CONTROLLED GOODS

(NON-CONTROLLED GOODS)
DMC A
REVIEW: GCEC DECEMBER 2013

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C or U) in parentheses after the title.)

Panpan : An expandable framework for the stimulation of software system activity

4. AUTHORS (last name, followed by initials – ranks, titles, etc., not to be used)

Thibault, D.U.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.)	6b. NO. OF REFS (Total cited in document.)
July 2017	36	0

7. DESCRIPTIVE NOTES (The category of the document, e.g., technical report, technical note or memorandum. If appropriate, enter the type of report, e.g., interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Reference Document

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

DRDC – Valcartier Research Centre
Defence Research and Development Canada
2459 route de la Bravoure
Quebec (Quebec) G3J 1X5
Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
05aa	

10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC-RDDC-2017-D060	

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

Unlimited

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13. ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.)

One of the goals of the Real-Time Monitoring (RTM) work-stream of the Platform-to-Assembly Secure Systems (PASS) project is to develop new advanced software system normality models for advanced online anomaly detection. Some of those models need to be trained using system traces captured while the monitored system is running normally. The need for a tool that could generate predictable and relatively simple system activity was identified. *Panpan* is such a tool: a modular application software framework that provides a simple means of stimulating normal activity in a controllable fashion.

Un des buts du volet de travail Real-Time Monitoring (RTM, Surveillance en temps réel) du projet Platform-to-Assembly Secure Systems (PASS, Systèmes sécurisés de la plate-forme à l'assemblage) est de développer de nouveaux modèles avancés de normalité des systèmes logiciels pour la détection sophistiquée en temps réel d'anomalies. Certains de ces modèles doivent être entraînés à l'aide de traces capturées tandis que le système à surveiller exécute des activités considérées normales. Le besoin d'un outil pouvant générer des activités prévisibles et relativement simples a été identifié. *Panpan* est cet outil, un cadre d'application logicielle modulaire fournissant un moyen simple de stimuler l'activité normale de façon contrôlable.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g., Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Software; Linux; System Calls; C; Tracing; Activity Patterns