

THE PELUDO SYSTEM

December 8, 2009

1 What is Peludo

Peludo is a system to create and run platform independent, self-contained, network-transportable, injectable applications written in the C programming language. It provides a cross-compilation environment and other tools needed to generate applications using a new binary format called PLD as well as a Runtime to launch these applications.

A common application usually involves a main executable, some dependency libraries and data files; under Peludo these files are normalized using the PLD binary format. The runtime component called Peludo Kernel is in charge of launching the main executable after loading and resolving all the required dependencies.

The reason behind this new binary format in contrast to other well known standards (like PE or ELF) is that PLD has a very small footprint, it is extremely scalable, and can be deployed almost everywhere.

2 PLD File Format

The PLD file format is the core of the Peludo Platform, everything is based on it and it is as simple as a TLV (Tag-Length-Value) format.

A PLD file is a container for *modules*(2.3), each *module* is composed of one or more *sections* (2.4) and each *section* is encoded as a *{name, size, data}* tuple.

If not otherwise specified, all values are encoded using the big endian byte order.

As a convention, PLD files are stored in the file system using the *.pld* extension.

2.1 Symbol Names

All symbol names are 32 bits values (file names, section names, exported and imported symbol names, etc.):

```
typedef uint32 PLDNAME;
```

2.2 Addresses

Addresses are specified using a 64 bits *section:offset* pair:

```
typedef struct {
    PLDNAME addr_section;
    uint32  addr_offset;
} PLDADDR;

addr_section Name of the section where the referenced element is
located;
addr_offset Offset into the section where the referenced element is
located.
```

2.3 Modules

Modules are containers for generic code and data and have the following features:

1. Modules may not share the same virtual address space (spread over running processes)¹

¹Not implemented in Peludo Cachicamo

2. Modules may contain executable code
3. Modules may contain readable and/or writable data
4. Modules may export symbols
5. Modules may import symbols from other modules
6. Modules may import symbols from native DLLs (provided by the underlying Operating System)²

Each PLD file contains one module that is composed of an arbitrary number of sections. It starts with the following "header":

```
typedef struct {
    PLDNAME  pld_name;
    uint32   pld_sections;
} PLDFILE;

pld_name Name of the PLD file;
pld_sections Number of sections contained in this PLD file.
```

A sequence of *pld_sections* sections continue after this information.

2.4 Sections

A *section* is a container for any kind of data:

```
typedef struct {
    PLDNAME  sec_name;
    uint32   sec_size;
    uint32   sec_flags;
} PLDSECTION;

sec_name Name of the section.
sec_size Size of the section's content.
sec_flags Section flags; a bitwise OR combination of:
PLDSECTION_FL_X: The section contains eXecutable code;
PLDSECTION_FL_R: The section contains Readable data;
PLDSECTION_FL_W: The section contains Writable data;
PLDSECTION_FL_Z: The section content is compressed.
```

The actual data continues after this information.

2.5 Type of Sections

The following standard section types are adopted:

2.5.1 CODE Section

The *.code* section contains executable code (like *.text* or *.code* in ELF or PE files).

2.5.2 DATA Section

The *.data* section contains the module's data (like *.[ro]data* in ELF or PE files).

²Not implemented in Peludo Cachicamo under unix.

2.5.3 EXPORT Section

The *.export* section contains a list of exported symbols; this list is implemented as a *NULL* terminated array of *PLDEXPORT_ENTRY* elements each of them associated to a single exported symbol:

```
typedef struct {
    PLDNAME exp_symname;
    PLDADDR exp_symaddr;
} PLDEXPORT_ENTRY;

exp_symname Name of the exported symbol;
exp_symaddr Address of the exported symbol.
```

2.5.4 IMPORT Section

The *.import* section contains information about symbols that should be imported from other PLD modules; it is implemented as a *NULL* terminated array of *PLDIMPORT_ENTRY* elements:

```
typedef struct {
    PLDNAME imp_modname;
    PLDADDR imp_symbols;
} PLDIMPORT_ENTRY;

imp_modname Name of the PLD module that exports required symbols.
imp_symbols Address containing the table of imported symbols from the
specified module; this table is implemented as a NULL terminated
array of PLDIMPORT_SYMBOL elements:

typedef struct {
    PLDNAME imp_symname;
    PLDADDR imp_symaddr;
} PLDIMPORT_SYMBOL;

imp_symname Name of the imported symbol.
imp_symaddr Address that should be updated with the virtual address
of the imported symbol.
```

2.5.5 NIMPORT Section

The *.nimport* (Native IMPORT) section contains information about symbols that should be imported from native libraries (DLL's provided by the underlying Operating System). It is implemented as a *NULL* terminated array of *PLDNIMPORT_ENTRY* elements:

```
typedef struct {
    PLDADDR nimp_dllname;
    PLDADDR nimpimpsym;
} PLDNIMPORT_ENTRY;

nimp_dllname Address containing the ASCIIIZ string describing the name
of the native DLL.
nimpimpsym Address containing the table of imported symbols from
the specified DLL; this table is a NULL terminated array of
PLDNIMPORT_SYMBOL elements:

typedef struct {
    PLDADDR nimp_symname;
    PLDADDR nimp_symaddr;
} PLDNIMPORT_SYMBOL;

nimp_symname Address containing the ASCIIIZ string describing the
name of the imported symbol.
nimp_symaddr Address that should be updated with the virtual address
of the imported symbol.
```

2.5.6 RELOC Section

The `.reloc` section contains information about relocatable elements. Usually, the `.code` section references values located into the `.data` section but since the virtual address of `.data` is not known in advance it is not possible to reference any data from other sections until it is loaded into memory. The `.reloc` section contains a table that instructs the PLD loader to update specified addresses with the virtual address of the referenced element; this table is implemented as a `NULL` terminated array of `PLDRELOC_ENTRY` elements:

```
typedef struct {
    PLDADDR rel_symbol;
    PLDADDR rel_address;
    uint32 rel_type;
} PLDRELOC_ENTRY;
```

rel_symbol Address of the referenced symbol.

rel_address Address to be updated with the virtual address of the referenced symbol.

rel_type Type of relocation:

- PLDRELOC_TYPE_ABS** `rel_address` should be updated with the absolute virtual address of the `rel_symbol`:
`*VirtualAddress(rel_address) = *VirtualAddress(rel_symbol);`
- PLDRELOC_TYPE_ADD** The absolute virtual address of `rel_symbol` should be added to the content of `rel_address`:
`*VirtualAddress(rel_address) += *VirtualAddress(rel_symbol);`

2.5.7 HASH Section

The `.hash` section is present only on PLD files compiled in debug mode and contains a table used to translate from `PLDNAMEs` to ASCIIZ strings. It is used by the PLD loader (that should be compiled in debug mode) to print out debugging information in a human readable form. This table is implemented as a `NULL` terminated array of `PLDHASH_ENTRY` elements:

```
typedef struct {
    PLDNAME hash_name;
    PLDADDR hash_string;
} PLDHASH_ENTRY;
```

hash_name A `PLDNAME`.

hash_string Address containing the ASCIIZ string of the `PLDNAME`.

3 PLD Chain

A PLD Chain is a sequence of PLD files in dependency order.

For example, if an application is associated to the following three modules:

- `main.pld`: The Main Application (where the `main()` function resides)
- `mainlib.pld`: The application's library (exporting symbols needed by `main.pld`)
- `libc.pld`: Libc library (needed by both `main.pld` and `mainlib.pld`)

A PLD Chain containing this application can be generated using the unix `cat(1)` command as follows:

```
$ cat libc.pld mainlib.pld main.pld > application.pld
```

4 PLB Files (Peludo Binary Files)

To execute an application encoded as a PLD Chain two additional components are provided:

- The *Peludo Bootstrap Code* (*krn0.plo*)
- The *Peludo Kernel* (*krn1.plo*)

A PLB file is created when a PLD Chain is concatenated to these components (see 6.1). It could be seen as a large executable embedding all dependency libraries and data files. As a convention, PLB files are stored in the filesystem using the *.plb* extension.

This is how a PLB file looks like:

```
+-----+-----+-----+-----+-----+...+-----+
| BOOTSTRAP | KERNEL CODE   KERNEL DATA | PLD1 | PLD2 |     | END |
+-----+-----+-----+-----+-----+...+-----+
```

Refer to section 6 for more information about the bootstrap code and the kernel.

Returning to the example of section 3, the Peludo Binary file for that application can be generated using the unix *cat(1)* command as follows:

```
$ cat krn0.plo krn1.plo application.pld end.pld > application.plb
```

where *end.pld* is a 0 filled PLD used to mark the end of the PLB file.

To execute it the only thing that should be done is to load the entire PLB file into page aligned executable memory and call its first byte, the bootstrap code, whose prototype is:

```
long boot(int argc, char **argv)
```

The *argc* and *argv* arguments are the same passed to the common C *main()* function.

5 Peludo Toolchain

To cross-compile, the current version of Peludo uses the GNU GCC Core Compiler (<http://gcc.gnu.org/>) and some tools provided by the GNU Binutils package (<http://www.gnu.org/software/binutils/>), both of them are automatically downloaded from the internet during the building process.

The provided third party commands are: *ar*, *as*, *gcc*, *cpp*, *ld*, *nm*, *objcopy*, *objdump*, *ranlib*, and *strip*. Refer to the appropriate man pages to get more information about them.

Peludo adds new commands to deal with PLD and PLB files: *prun*, *plzma*, *pldhash*, *plb2elf*, *plb2c*, *elf2pld*, *data2pld*, *dir2pld*.

5.1 HOST and TARGET

The HOST is the system where the toolchain runs; it is where modules are compiled.

The TARGET is the platform where generated applications or modules are intended to run.

5.2 New Commands

5.2.1 prun

```
prun <plb>
```

Executes a PLB file (this command should be launched in the target platform).

5.2.2 plzma

```
plzma <e|d> <infile> <outfile>
```

Compress/Uncompress files using the same LZMA method used to compress PLD files (See section flags in 2.4).

5.2.3 pldhash

```
pldhash <string>
```

Generates a hash value associated to the specified string. This command is used to generate *PLDNAMEs* from user defined strings (See section 2.1).

5.2.4 plb2elf

```
plb2elf [-q] -p <platform> <plb> <elf>
```

Generates a static executable ELF file embedding an entire PLB that runs on a specified target platform.

5.2.5 plb2c

```
plb2c <plb>
```

Generates a simple C program that, once compiled, launches the specified PLB file. The output of this command can be used as a template when one or more PLB files should be embedded into some container application at C level.

The Peludo Toolchain is not needed to compile the generated code (just take any gcc compatible C compiler in the target platform, compile it, run it).

5.2.6 elf2pld

```
elf2pld [options] <elf> [<pld>]
```

Generates a PLD from a "specially crafted" ELF shared library. The user should not use this command directly (see 5.3).

5.2.7 data2pld

```
data2pld [options] <file> [<pld>]
```

Generates a PLD containing a single *.data* section (see 2.5.2) embedding the specified file.

5.2.8 dir2pld

```
dir2pld [options] <dir> <pld>
```

Generates a PLD containing a *.data* and a *.export* (see 2.5.2 and 2.5.3) sections embedding all the regular files under the specified directory (and subdirectories).

The content of these files are stored in the data section; the export table is created to make them easily accesible from other PLD modules.

5.3 MK System

The MK system is composed of a set of makefiles using the *.mk* extension, Its purpose is to make it easy to create PLD modules and PLB files. These files are:

5.3.1 pbase.mk

Contains the base definitions needed by all the other *.mk* files.

5.3.2 objhost.mk

Used to compile C or Assembly code that should run in the HOST system.

5.3.3 binhost.mk

Used to generate static ELF files able to be launched in the HOST system.

5.3.4 objtarget.mk

Used to compile C or Assembly code that should run in the TARGET system.

5.3.5 libtarget.mk

Used to generate libraries (.a files or .pld modules) for the TARGET system.

5.3.6 bintarget.mk

Used to generate the *main PLD module* intended to run in the TARGET system. This module should export the *main()* function.

5.3.7 plbtarget.mk

Used to generate PLB files intended to run in the TARGET system. It can also be used to generate native ELF embedding a PLB.

5.3.8 clean.mk

Used to clean generated files, leaving only the source code.

5.3.9 dir.mk

Used to traverse directories.

5.3.10 install.mk

Used to install generated files.

5.3.11 download.mk

Used to download packages from the internet. It requires the *openssl* (<http://www.openssl.org>) and the *wget* (<http://www.gnu.org/software/wget>) utilities.

6 Peludo Runtime

6.1 Peludo Core

6.1.1 Bootstrap Code

The Bootstrap Code is the first piece of code that gets executed when a PLB file is launched; it resolves and executes the Peludo Kernel.

6.1.2 Peludo Kernel

The main purpose of the Peludo Kernel is to load, resolve and execute an attached PLD Chain. It contains the PLD loader and other components allowing the application to access the operating system resources. If the chain is successfully loaded and some module in it exports the *main()* function it is executed. In other words: The kernel loads and starts the user application.

When coding kernel components the user must initialize all global variables even if they should be set to 0, otherwise it is generated a *.bss* section that is discarded when the kernel binary file is created. As an extension to this rule, all global pointers must be initialized to *NULL* (they should not point to any data); an initialization function should take care of these pointers. The reason behind this last restriction is that the bootstrap code relocates only the *.got* but initialized global pointers belongs to *.data*.

Example of a non-working code:

```

char *hello = "hello world\n";
void foo()
{
    print(hello);
}

```

The following code instead is correct:

```

char msg[] = "hello world\n";
char *hello = NULL;
void init()
{
    hello = msg;
}
void foo()
{
    init();
    print(hello);
}

```

6.2 Special PLD Files

6.2.1 CMDLINE

When the Kernel is resolving a PLD Chain (see 3) it recognizes a special PLD file named *.cmdline* containing the default command line options. These arguments should be placed in the PLD's *.data* section (2.5.2) encoded as follows:

```
arg1\0arg2\0arg3\0...argn\0\0
```

They are used when the application is called without arguments and it is really easy to generate using the unix shell:

```
$ printf "arg1\0arg2\0arg3\0\0" > cmdline.tmp && \
  data2pld -0 -p .cmdline cmdline.tmp cmdline.pld
```

See 5.2.7.

6.3 C Library

For historical reasons, Peludo provides a minimalistic C library that is not a full ANSI/POSIX library. Although, 90% of its methods are ANSI/POSIX compliant. It will become a minimalistic full ANSI/POSIX C library in future releases.

One of the major differences is in the *printf(3)* implementation where only a few format string options are recognized and some of them are not standard at all:

```

%c Same as standard printf(3)
%d Same as standard printf(3)
%u Same as standard printf(3)
%p Same as standard printf(3)
%s Same as standard printf(3)
%x Equivalent to %08X
%X Equivalent to %016lX

```

printf(3) will become more or less standard in the future.

The main goal of the Peludo C Library is portability not speed. Speed improvements will be added in future releases (see 6.7).

6.4 Multithread Library

Peludo provides a subset of the POSIX Threads API: Most of the POSIX Threads methods are supported but not all of them.

The main goal of the Peludo Multithread Library is portability not speed. Speed improvements will be added in future releases (see 6.7).

6.5 PLZMA Library

The PLZMA library is a port of the LZMA library created by the 7Zip project (see <http://www.7-zip.org/sdk.html>).

Sections of a PLD file are compressed (if so instructed) using these methods.

The PLZMA decoder is integrated into the Peludo Kernel (see 6.1.2).

6.6 Peludo File System

The Peludo File System is a work in progress.

It is an on-memroy file system where user applications can store temporal data.

6.7 Profiler Library

The Profiler is a work in progress.

It is a library whose main purpose is to generate runtime usage information that is precious to improve the C Library and others.