Keeping the ABI [somewhat] sane

Gabriel F. T. Gomes

IBM

Agenda

- Introduction
- Breaking the ABI
- SONAME versioning
- Symbol versioning
- Header redirection
- Real world example
- Conclusion

Preface

prog: libc.so.6: version `GLIBC_2.XX' not found
(required by prog)

Introduction

- Application Programming Interface (API)
  - “An API, typically defined at the source code level, enables applications written to the API to be ported easily (via recompilation) to any system that supports the same API”.

    James E. Smith and Ravi Nair – Virtual Machines

  - Library routines;
  - Source-code level.

Introduction

- Application Binary Interface (ABI)
  - “A program binary compiled to a specific ABI can be run unchanged only on a system with the same ISA and operating system”.

    James E. Smith and Ravi Nair – Virtual Machines
  - ISA;
  - System call * interface:
    - Syscalls;
    - Library calls.

Introduction

- Application Binary Interface (ABI)
  - Calling convention;
  - Data structures;
  - Size and format of basic types;
  - Binary format of executables.

Introduction

- Fictional example: libamazing-0.1
  - git.inscontante.net.br/?p=libamazing.git
  - API:

```
1 /* Amazing library with amazingly useless functions.  */
2
3 #define AMAZING_CONSTANT 12
4
5 int
6 libamazing_constant (void);
```

Introduction

- Fictional example: libamazing-0.1
  - git.inscontante.net.br/?p=libamazing.git
  - ABI:

```
$ readelf --symbols libamazing.so | grep libamazing_constant
    11: 0005fa  11 FUNC  GLOBAL DEFAULT   11 libamazing_constant
```

Breaking the ABI

- API breaks
  - Routine removal;
  - Parameter list change;
  - Return type change;
  - Data structure change.

Breaking the ABI

- ABI breaks
  - Type format change;
  - Routine behavior change.

Breaking the ABI

- Trouble!
- Programs crash;
- Programs produce weird outputs;
- Cost to deal with the changes:
  - User programs need to be fixed upstream;
  - Downstream needs to rebuild user programs;
  - Downstream might need to distribute multiple versions of the library.

Breaking the ABI

- Fictional example: libamazing-0.2

```
1 /* Amazing library with amazingly useless functions.  */
3
4 #define AMAZING_CONSTANT 1.41421356237309504880168872420f
5
6 float
7 libamazing_constant (void);
```

  - Breaks previously-built user programs;
    - crash, wrong output.
  - Code fixing and rebuilds.

SONAME versioning

- Announce backwards-incompatible ABI breaks.
- Catch problems early
- SONAME version examples:
  - libc.so.6, libz.so.1, libgcrypt.so.20.

SONAME versioning

- Fictional example: libamazing-0.3
  - SONAME version: libamazing.so.2
  - How to:

```
$ gcc -Wl,-soname=libamazing.so.2 -o libamazing.so (...)
```

  - Checking:

```
$ readelf --all libamazing.so | grep SONAME
 0x000000000000000e (SONAME)  Library soname: [libamazing.so.2]
```

SONAME versioning

- Fictional example: libamazing-0.3
  - SONAME version: libamazing.so.2
  - Catching problems early:

```
# On a system with libamazing.so.2

$ ./prog
prog: error while loading shared libraries: libamazing.so.1:
cannot open shared object file: No such file or directory
```

Symbol versioning

- Avoid SONAME changes:
  - Avoid distribution of multiple versions of a library
- Allow function behaviour changes:
  - One function name, many versions;

Symbol versioning

- Binutils' VERSION command:
  - "The dynamic linker can use symbol versions to select a specific version of a function".

    https://sourceware.org/binutils/docs/ld/VERSION.html
- How to:
  - Version script (--version-script):

```
$ gcc -Wl,--version-script Versions-file (...)
```

Symbol versioning

- Version script syntax

```
1 <older_version_name> {
2   global:
3     <exported_function_name>;
4 };
5
6 <newer_version_name> {
7   global:
8     <exported_function_name>;
9 } <older_version_name>;
```

Symbol versioning

- Version script example (libamazing-1.0):

```
1 libamazing_0.1 {
2   global:
3     libamazing_constant;
4 };
5
6 libamazing_0.2 {
7   global:
8     libamazing_constant;
9 } libamazing_0.1;
```

Symbol versioning

- Selecting internal implementation:

```
 1 float
 2 __float_libamazing_constant (void) {
 3   return 1.41421356237309504880168872420969807f;
 4 }
 5 asm (".symver __float_libamazing_constant,"
        "libamazing_constant@@libamazing_0.2");
 6
 7 int
 8 __int_libamazing_constant (void) {
 9   return 12;
10 }
11 asm (".symver __int_libamazing_constant,"
        "libamazing_constant@libamazing_0.1");
```

Symbol versioning

- Displaying symbols versions
  -

```
$ readelf --all libamazing.so.2

  12: 06c8 FUNC GLOBAL DEFAULT libamazing_constant@libamazing_0.1
  13: 06ba FUNC GLOBAL DEFAULT libamazing_constant@@libamazing_0.2
```

  - Notice the single ‘@’ and double ‘@@’;
  - Double ‘@@’ is the default used in new builds.

Symbol versioning

- Versioned symbols in use by user program:

```
  1 int main (void) {
  2   float f = libamazing_constant ();
```

```
$ gcc user.c -lamazing -o user

$ objdump -D user

  1   010000834 <main>:
  2-8   (...)
  9     bl 010000620 <libamazing_constant@@libamazing_0.2>
```

More API/ABI breaks

- libamazing-1.1 (API and ABI break)

```
1 /* Double-precision floating-point constant.  */
2 double
3 __double_libamazing_constant (void) {
4   return AMAZING_CONSTANT;
5 }
6 asm (".symver __double_libamazing_constant,"
       "libamazing_constant@@libamazing_1.1");
```

```
1 libamazing_1.1 {
2   global:
3     libamazing_constant;
4 } libamazing_0.2;
```

More API/ABI breaks

- libamazing-1.2 (ABI-only break)

```
1 /* Double-precision floating-point constant.  */
2 double
3 __new_double_libamazing_constant (void) {
4   return 12;
5 }
6 asm (".symver __double_libamazing_constant,"
       "libamazing_constant@@libamazing_1.2");
```

```
1 libamazing_1.2 {
2   global:
3     libamazing_constant;
4 } libamazing_1.1;
```

More API/ABI breaks

- All symbols

```
$ readelf --all libamazing.so.2

  13: 07c4 FUNC GLOBAL DEFAULT libamazing_constant@libamazing_0.1
  16: 07b6 FUNC GLOBAL DEFAULT libamazing_constant@libamazing_0.2
  17: 07a8 FUNC GLOBAL DEFAULT libamazing_constant@libamazing_1.1
  14: 079a FUNC GLOBAL DEFAULT libamazing_constant@@libamazing_1.2
```

- New builds use the default symbol (@@).
- What if we want to use the old version?

Header redirections

- Allows using older ABIs
- Compile-time feature
  - Header magic to redirect function calls.

Header redirections

- Fictional example: libamazing-1.3

```
1 #ifdef __LIBAMAZING_COMPAT_DOUBLE
2 extern __typeof (libamazing_constant)
3 libamazing_constant __asm__ ("__double_libamazing_constant");
4 #endif

$ gcc user.c -D__LIBAMAZING_COMPAT_DOUBLE
```

Header redirections

- Displaying the redirection
  - Not visible in .i files, but in .s (-save-temps)

```
 1 main:
 2 0:       addis 2,12,.TOC.-0b@ha
 3          addi 2,2,.TOC.-0b@l
 4          .localentry     main,.-main
 5          mflr 0
 6          std 0,16(1)
 7          std 31,-8(1)
 8          stdu 1,-144(1)
 9          mr 31,1
10          bl __double_libamazing_constant
11          nop
```

Real world example

- The long double ABI on powerpc (libc.so.6):
  - Version 2.0
    - long double == double
  - Version 2.4
    - long double == IBM Extended Precision
  - Version 2.29? (under development)
    - long double == IEEE Quadruple Precision

Real world example

- The symbols:

```
$ readelf --all /lib64/libm.so.6

 500: 0fe3d8  7732 FUNC GLOBAL DEFAULT   23 sinl@GLIBC_2.3
 502: 0fdb68   468 FUNC GLOBAL DEFAULT   23 sinl@@GLIBC_2.4
1019: 0fe3d8  7732 FUNC LOCAL  DEFAULT   23 __sin
1272: 0fdb68   468 FUNC LOCAL  DEFAULT   23 __sinl

$ readelf --all /lib64/libc.so.6

 653: 234c38   132 FUNC GLOBAL DEFAULT   29 printf@GLIBC_2.3
 651: 236198    96 FUNC GLOBAL DEFAULT   29 printf@@GLIBC_2.4
6966: 234c38   132 FUNC GLOBAL DEFAULT   29 __nldbl_printf
4812: 236198    96 FUNC LOCAL  DEFAULT   29 __printf
```

Real world example

- The redirections:
  - In /usr/include/sys/cdefs.h

```
1 #  define __LDBL_REDIR1_DECL(name, alias) \
2   extern __typeof (name) name __asm (__nldbl_##alias);
```

  - In /usr/include/bits/stdio-ldbl.h (included when -mlong-double-64)

```
1 __LDBL_REDIR_DECL (fprintf)
2 __LDBL_REDIR_DECL (printf)
3 __LDBL_REDIR_DECL (sprintf)
4 __LDBL_REDIR_DECL (vfprintf)
  (etc.)
```

Real world example

- IEEE Quadruple precision
  - Reusing _Float128 (API) implementation
  - New exported symbols

Conclusion

- ABI changes happen. Sometimes they are not backwards compatible.
- SONAME changes convey such information downstream (but might require rebuilds and concurrent distribution of versions).
- Symbol versioning avoids SONAME changes, but with an additional implementation cost.
- Downstream mindset for upstream development.

Thank you!

gabriel@inconstante.net.br