libC compartmentalization How-To

Marinos Tsantekidis
Technical University of Braunschweig, Germany
Institute of Computer Science – FORTH, Greece
m.tsantekidis@embeddedsecurity-tubs.eu

Table of Contents

1. Introduction

libC is the standard library for the C programming language which is used extensively for development in Linux environments. Every program written in C uses some version of libC by default. However, libC and more specifically the version that we are experimenting with – GNU libC, glibc – is big and contains an array of functions to perform many kinds of procedures. In a given program, many of these functions may not be necessary depending on what the program does, nevertheless they are loaded at runtime as part of the whole glibc. This results in increased memory consumption, when there may not be any need for it. There are instances of programs that take up 1KB of memory, when libC that needs to be loaded takes up 17 MB of memory. Up to a point, libC is already compartmentalized with respect to its portions that need to be specifically loaded when a program requires their use (e.g., libm, libcrypt, etc.). In this work, we present our efforts to compartmentalize libC at a greater granularity, so that only the absolutely necessary functions can be loaded during execution and less memory can be used by the running program.

In [1], the author presents his approach, where he first determines which objects from libC library will be needed by an application and then builds a custom version of libC that includes only these objects. Then the application is executed after being linked with the minimal custom libC. This approach is similar to our own in the sense that it tries to minimize the memory footprint and size

(i.e. attack surface) of libC, however it requires that every application be analyzed in order to determine the necessary objects and a separate version of libC be built for each specific application. This way, the user needs to interact with the tool and have technical knowledge in order to operate it, for each application they want to run with the minimal libC. Our approach is one-off, meaning that the library is built only once and can then be used by all applications automatically. Furthermore, it is totally transparent to the user, since they are not required to build libC themselves, but only link a specific extra library that they want to use at compile/execution time, similarly to other libraries of libC that need this by default (e.g., libm, libcrypt, etc.).

In our previous work [3][2], we separate the memory of a running process into regions along the lines of loaded shared libraries, one of which is libC. Furthermore, we install a “gate” before each region, that manages the access to the corresponding region based on security policies that the running program adheres to. This way only specific, allowed code can execute a region’s code, resulting in increased security during an attempted attack. For example, if a program requires only I/O operations but at run-time there is a call e.g. to a math function, it would be disallowed by the gate. Based on our work, by further compartmentalizing libC, besides smaller memory footprint, we also manage to strengthen the security of a running application since there are more gates – meaning more security checks – that need to be passed successfully in order to execute code from an intended region. Additionally, we decrease the attack surface of a potentially malicious attempt, since libC contains only the bare minimum and all other code that the application requires is loaded in the form of extra libraries, leading to much smaller size of the loaded libraries.

Based on the source code of glibc (see section 3), it contains more than 700 directories, 17.500 files and 4.000.000 Lines of Code (LoC). Consequently, it is apparent that it is a big library that offers ample attacking ground for a determined adversary. The more code resides into the loaded libC, the more code space it takes up during execution and the more possibilities an attacker has to exploit a bug and mount an offensive against the underlying system. With our approach, that requires only minimal changes and additions to glibc source, we aim to minimize the code loaded at runtime only to what is absolutely necessary, by loading only specific portions of libC that are mandatory for the program to run correctly and nothing else.

2. Implementation

This report contains details on two approaches: (a) how to extract specific functions from the source code of glibc and (b) how to extract a whole directory from the source code of glibc. In both cases, an extra dynamic library will be created that needs to be explicitly loaded at compile/execution time, if we want to use the specific functions in our program.

2.1. dysize function

In the first case, we extract a single function (dysize) from the *time* directory and create the extra dynamic library containing only this function.

Three files need to be modified, in order to successfully do this:

(a) <glibc_source_code_directory>/time/Makefile

(b) <glibc_source_code_directory>/time/Versions

(c) <glibc_source_code_directory>/shlib-versions

2.1.1 Makefile

In this file, we need to remove the dysize() function from the list of routines that will be built for the *time* part of glibc. We also need to state that we want an extra library to be built that contains only the dysize() function.

- In line 36, we delete *dysize*
- After line 41, we add:

```
extra-libs        = libmartsan_dysize
extra-libs-others = $(extra-libs)
libmartsan_dysize-routines = dysize
```

2.1.2 Versions

In this file, after line 82 (end of file), we add:

```
libmartsan_dysize { GLIBC_2.0 { dysize; } }
```

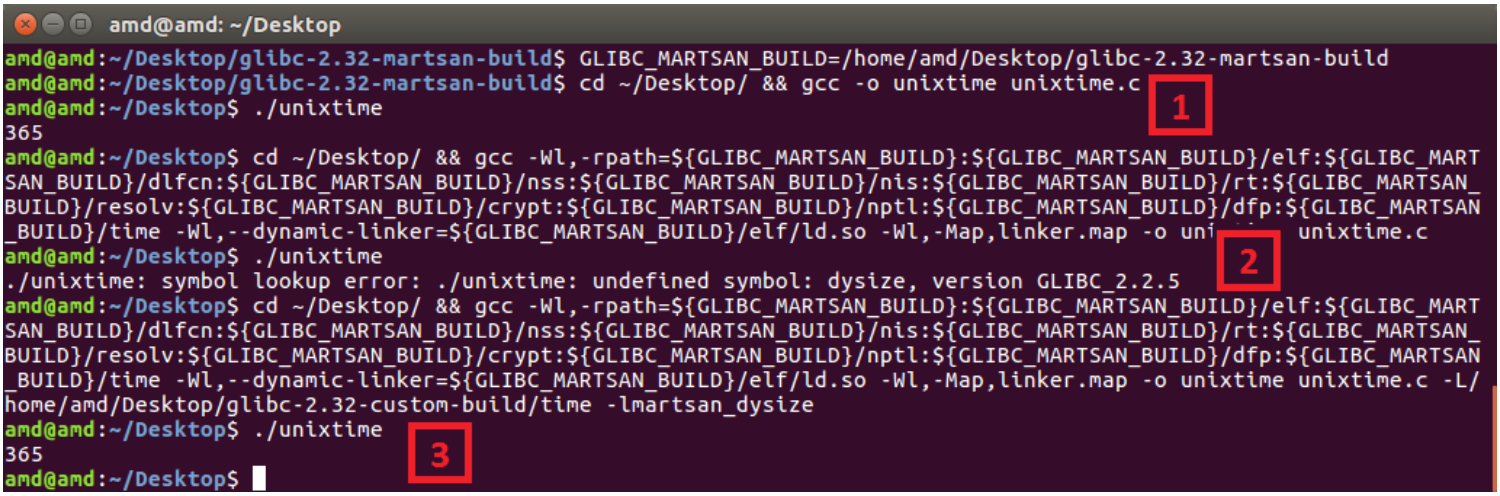
2.1.3 shlib-versions

In this file, after line 75 (end of file), we add:

```
libmartsan_dysize=1
```

2.1.4 Compile, link and run

Next, we build glibc normally (see section 3). Then, in order to use dysize() in a program, we must load it explicitly with -l flag, as shown in the figure below.



In point 1, we compile and link with the system libC, so the program runs as expected. When in 2, we compile and link with the custom libC that does not contain dysize(), when trying to execute we get an error since dysize() is missing. After linking in the extra library (see section 3), the program runs as in the normal case, in 3.

2.2. catgets directory

In this case, we extract the whole *catgets* directory from glibc, which deals with some translation aspects. In order to do this and create an extra dynamic library containing the related functions, we need to modify the following files:

(a) <glibc_source_code_directory>/Makeconfig

(b) <glibc_source_code_directory>/shlib-versions

Later on, this can be extended to include other functions from other portions of libC, if needed.

2.2.1 Makeconfig

In this file, we need to remove the *catgets* directory from the list of subdirectories containing the libC source. This way, the subdirectory does not get built into libC.

- In line 1270, we delete *catgets*

2.2.2 shlib-versions

In this file, we need to tell the final libC that there will be an extra shared library that we will be able to use. So, after line 75 (end of file), we add:

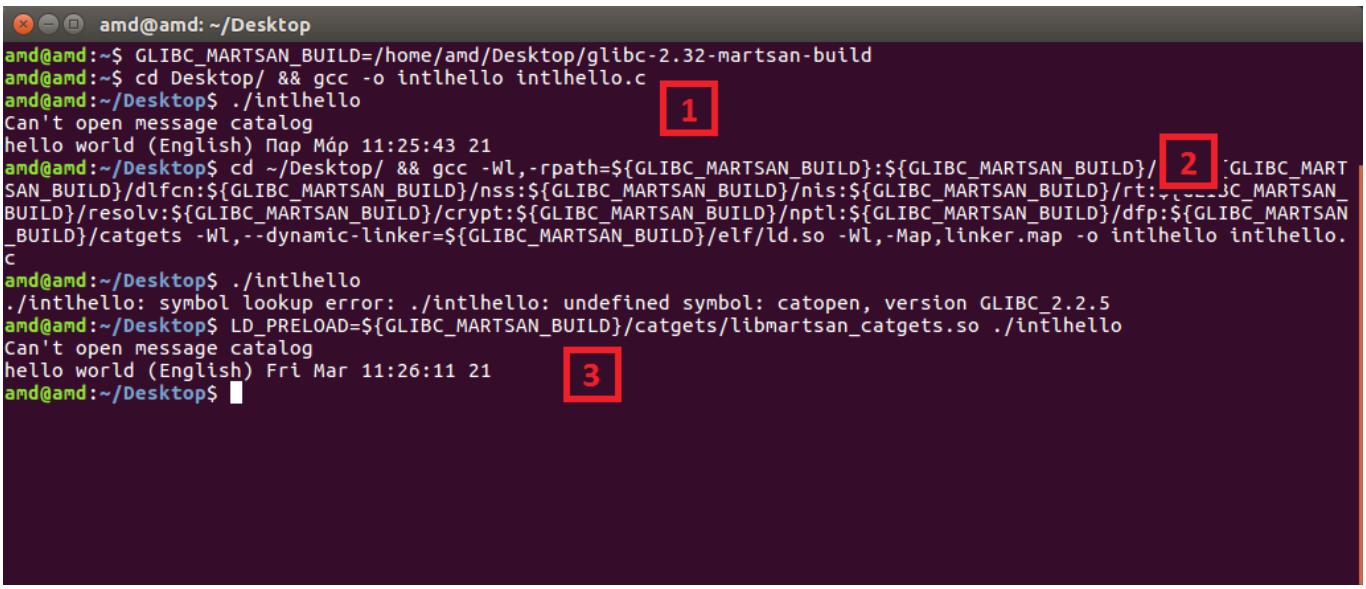
```
libmartsan_catgets=1
```

2.2.3 catgets_build

Additionally, we need to run the executable file *catgets_build*. This file takes care of some dependencies and compiles all catgets-related C files into a shared library, based on the normal-case build process of glibc. Keep in mind to modify the variables inside the file (first two lines) that point to the source code directory and build directory of the custom glibc, with the correct paths.

2.2.4 Compile, link and run

Next, we build glibc normaly (see section 3). Then, in order to use a catgets-related function in a program, we must load it explicitly, as shown in the figure below.



In point 1, we compile and link with the system libC, so the program runs as expected. When in 2, we compile and link with the custom libC that does not contain the catgets-related functions (in this case *catopen*), when trying to execute we get an error since these functions are missing. After linking in the extra library with LD_PRELOAD, the program runs as in the normal case, in 3.

2.3. Effort

In each of the previous cases, it is evident that only minor changes are required to make our approach happen:

a) dysize
- Deletion of one word
- Addition of 5 lines in total

b) catgets
- Deletion of one word
- Addition of 1 line in total
- Execution of a pre-made script

However, this approach deals with functions that do not take part anywhere in the building procedure of glibc, meaning that at compile/link time of glibc there is no need to use these functions. Thus, they can be extracted from the library and be made into libraries themselves that can be loaded externally when a program requires them. When attempted to extract portions that do take part in the building process (e.g. one of the *string* functions), we encountered numerous errors mainly because of missing function/variable declarations/definitions. Nevertheless, it is our understanding that this has primarily to do with the order of building the various portions, i.e. if we first compile such a function as a shared library and then link it externally to the rest of the procedure, we expect it will be completed without any problems.

Below we provide a list of portions of glibc with their respective final code space required, that can possibly be extracted and compiled as shared libraries. This definitely requires more effort to look into and will be the subject of a future report.

argp	1 MB	intl	2.4 MB	signal	1.9 MB
assert	190 KB	io	5 MB	socket	1 MB
csu	968 KB	libio	15.3 MB	stdio-common	9 MB
ctype	606 KB	malloc	2.6 MB	stdlib	7.1 MB
dirent	2.5 MB	nscd	4.7 MB	string	8.5 MB
gmon	593 KB	posix	8.3 MB	sysvipc	717 KB
grp	1.5 MB	pwd	1.2 MB	termios	653 KB
gshadow	1 MB	resource	566 KB	time	3.3 MB
iconv	3.6 MB	setjmp	191 KB	wcsmbs	6.7 MB
inet	8.2 MB	shadow	1.2 MB	wctype	927 KB

Table 1: glibc portions and their size

3. glibc build

In order to first obtain glibc's source code, it can be downloaded from https://ftp.gnu.org/gnu/glibc/. All modifications in this report were performed in version glibc-2.32 (file glibc-2.32.tar.gz).

Instructions on how to build the custom glibc can be found here https://sourceware.org/glibc/wiki/Testing/Builds – section "Building glibc without installing". All compilation is based on the commands in section "Compile against glibc build tree".

Bibliography

[1] Darren Hart, Building a Minimal Glibc with Componentization, In Linux Journal, 2001, Available: https://www.linuxjournal.com/article/5457

[2] Marinos Tsantekidis and Vassilis Prevelakis, Software System Exploration Using Library Call Analysis, In Model-driven Simulation and Training Environments for Cybersecurity, 2020, Available: https://doi.org/10.1007/978-3-030-62433-0_8

[3] Marinos Tsantekidis and Vassilis Prevelakis, Efficient Monitoring of Library Call Invocation, In Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), 2019, Available: https://doi.org/10.1109/IOTSMS48152.2019.8939203