Assessing the Correctness of Debloating Binary Shared Libraries with LibFilter

Jearson Alfajardo
Brown University

Abstract

Shared libraries (Dynamically-linked libraries in Windows) are application components that allow for a single copy of common code to be shared by multiple applications at runtime, allowing for reduced binary sizes and application memory footprints. However, not all code within these libraries are necessarily used; only a subset is needed while the rest is loaded into memory as *bloat*.

While more-or-less benign from a performance standpoint, security-wise code bloat proves to be an additional liability. All executable code, regardless of its use by the application, is susceptible to being co-opted by attackers performing code-reuse-based exploits such as ret2libc and ROP. Thus, eliminating such bloat would go a ways to improving the security posture of a given application.

LibFilter [10] was a system developed specifically to this end. It aimed to reduce the number of functions and gadgets useful to code-reuse-based exploits by using static analysis to determine the set of unused shared library functions imported by stripped x86_64 executables and erasing their bytes.

In theory, removing anything unused from an application, be it code or data, should have no bearing on its overall functionality. However, in practice, this property is dependent not only on the accuracy of the unused function analysis, but as well on the absence of programming errors. For LibFilter, any mistakes made in either of these aspects equate to unexpected program crashes later on at runtime. Thus, validating LibFilter's correctness is imperative for correct program execution.

In this paper, we extend previous work done on LibFilter by verifying its correctness through the test suites of several popular, open-source applications: *GNU coreutils* [1], *Nginx* [7], and *GNU m4* [4].

We test the LibFilter prototype as presented in the original paper and fix any issues we find. We then show the results of binaries erased by the updated LibFilter on their respective test suites, showing that its analysis was correct enough for most of the tests to pass.

1 Introduction

Most modern systems make use of data execution prevention (DEP) features to prevent the direct execution of code injected into a process's address space. This widespread adoption has prompted attackers to shift tactics towards code-reuse as a basis for software exploits, stringing together sequences of existing instruction sequences and/or existing program functions in order to bypass DEP. Today, code-reuse as an attack technique is prevalent in exploits.

In response, defenses such as Control-Flow Integrity (CFI), eXecute Only Memory (XOM), and Code Pointer Integrity (CPI) have been introduced and developed by the security community to either mitigate or outright prevent variants of code-reuse, either by restricting the set of permissible control flow transfers for the target program (CFI, CPI) or by restricting the information that could be leaked by reads on sensitive memory addresses (XOM).

Code debloating is an orthogonal means of mitigating the effectiveness of code-reuse attacks. In contrast to adding instructions or components to programs, its goal is to mitigate through subtraction, by eliminating unnecessary instructions within programs. This technique is predicated on the notion that, all code, regardless of its use/non-use by the program, is fair game to be co-opted by an attacker performing code-reuse. As such, removing unneeded code automatically raises the bar for attackers, by reducing the amount of material available for use in their payloads.

However, subtracting code from a binary is not without its pitfalls, the most prominent of which occur in the *unused analysis* that identifies and marks unneeded code segments for erasure. A mistake in the unused analysis eventually manifests itself as an unexpected program crash later on at runtime, where the eliminated code is used. Therefore, any unused analysis performed by code debloating tools needs to have results that are sound, i.e. have all of the results it returns as actually unused by the program.

Being able to prove the soundness of a result, however, is often difficult. In the case of LibFilter, which analyzes and

erases code at the function level, this requires that we possess a test suite with complete function coverage for the target program (i.e. one that executes all of its needed functions at least once). The modified program would then need to pass this test suite in order to demonstrate that the analysis is correct. Unfortunately, test suites with complete coverage are something generally unavailable for non-trivial programs, especially those undergoing continual development.

Still, while proving the soundness of the unused analysis may be out of reach in the general case, a number of open source programs do exist with sizable test suites that (by virtue of open source) have been authored by a number of contributors over the lifetime of the software. This results in test suites that have attained a respectable amount code coverage and feature coverage necessary for its continued development. Thus, we can leverage these test suites to vouch for LibFilter's correctness, in place of proving it. Programs erased by LibFilter that pass their test suites allow for us to make the case that the modified result is, at least, correct enough to be pushed to production.

2 Background and Related Work

In this section we overview LibFilter and its workings. Interested readers should refer to the original paper for more details.

2.1 LibFilter

To determine the and erase the set of unused functions from stripped binaries, LibFilter is composed of several main steps.

Disassembly Disassembly is the process of taking the machine instructions present within a binary and converting it into assembly language, where it can be more easily analyzed by tools and humans. LibFilter operates on stripped binaries for its analysis, which pose a particular challenge for disassemblers due to the absence of metadata that would more easily allow for code and data within the binary to be distinguished, as well as provide information regarding function boundaries. To be able to disassemble stripped binaries, LibFilter relies on previous research, *Egalito* [11].

Egalito [11] is a binary recompilation framework capable of performing a complete and precise disassembly of binaries in the absence of debug symbols. It does this by leveraging artifacts and metadata within the binary that are left over from the compilation process, claiming they are enough to generate an accurate disassembly from. The result is a low-level, platform-agnostic intermediate representation (IR) that can be used by researchers for creating binary analysis tools or by developers wanting to cross-compile to a different target architecture.

FCG Extraction LibFilter uses the IR returned by *Egalito* to extract the function call graph (FCG) for each of the binaries it processes, starting with the main program binary and then each of its shared libraries and their dependencies. Analysis begins at the specified entry points of each binary, where discovered `call` and `jmp` instructions are followed to their target functions to generate the caller-callee relationships that form that binary's FCG.

Used/Unused Function Analysis Once extracted, LibFilter then analyzes the FCG to determine the subgraph composed of used functions. The steps for the used function analysis are identical to the process performed by LibFilter's sister work Sysfilter [9], which analyzes the set of used functions to determine the minimal set of syscalls needed by a program, generating a syscall whitelist that could be enforced at runtime.

The result of the used function analysis is a pruned ('vaccummed') function call graph that, due to the limitations inherent in static analysis (such as the ambiguity of control-flow targets for indirect `jmp` instructions), is an overapproximation of the function call graph actually used by the program. From there, the set of unused functions is defined as any function within the binary not contained within the used FCG, and each of their offset and size information are emitted.

Function Erasure Once the set of unused functions have been determined for each of the shared libraries, the information emitted by the previous step is given to an eraser program that generates a copy of the library with the code bytes of the unused functions replaced with `hlt` instructions.

2.2 Including Dynamically-loaded Code

A known limitation of LibFilter is that, since its analysis is static, it does not include code loaded dynamically through interfaces such as `LD_PRELOAD`, `dlopen()`, or `dlsym()` when generating the used FCG. Code loaded at runtime potentially contain references to functions located within other already analyzed libraries and determined to be unreachable. When the modified program runs, it predictably crashes when the dynamically loaded function symbol is called.

To work around this, LibFilter has a feature that allows for its users to manually include functions from shared libraries in the form of a JSON file. Users simply have to specify the file path of the shared library as well as the function symbols to include in the FCG. This is useful for handling the aforementioned case, provided the shared library and the name of the needed function symbol are known beforehand.

3 Correctness Tests

In this section we briefly discuss the programs used for testing and their accompanying test suites.

3.1 GNU Coreutils

GNU Coreutils is a set of 108 utility binaries created by the GNU Foundation for basic file, text, and shell manipulation.

```
--with-http_ssl_module
--with-http_v2_module
--with-debug
--with-threads
--with-http_mp4_module
--with-http_gunzip_module
--with-http_gzip_static_module
--with-http_auth_request_module
--with-mail
--with-mail_ssl_module
--with-http_perl_module
--with-stream
--with-http_realip_module
--with-stream_ssl_module
```

Figure 1: Flags used to configure Nginx

For our tests, we cloned version 8.30 from the readonly Git mirror of the Coreutils repository [2].

This version of the source contains a non-root portion of the tests that we used, consisting of 617 separate test files, 557 of which are shell scripts and 63 of which are Perl scripts, all exercising various functionality for each of the Coreutils binaries. The tests were run with the command `make check-very-expensive` at the root of the repository.

3.2 Nginx

Nginx is a popular open-source HTTP, reverse-proxy, mail-proxy, and generic TCP/UDP proxy server created by Igor Sysoev. We use version 1.16.1 of Nginx, pulled from the readonly Git mirror of the main repository [6].

The repository consists of a configure script that accepts flags which allow for additional modules to either be statically compiled into or dynamically include by the main Nginx program, such as those relating to SSL, mp4 delivery, compression, etc. Since LibFilter does not support analyzing dynamically loaded modules (i.e. those that are loaded in at runtime), we opt to compile the modules in statically using the configure flags listed in Figure 1.

For testing, an Nginx has a separate repository consisting Perl scripts that exercise functionality from both the main binary and any additional modules. The version of the tests we used was from the Feb 7th commit (SHA 80337d6b0245e793705dd682c485662fcb9304e7) of the readonly Git mirror of the test repository [8]. The tests use Perl 5's `Test` module as its harness and the suite itself consists of 356 separate Perl scripts. These tests were run by running the command `TD: prove .` at the root of the nginx-tests repository.

3.3 GNU m4

GNU m4 is an implementation of the traditional Unix macro-processor, and is the simplest of the programs we tested.

The repository consists of a single binary with a test suite of 149 test binaries and 51 shell scripts for a total of 200 test files. We make use of version 1.4.18 downloaded from the GNU FTP server [5], and applied a patch [3] to work around a `glibc` deprecation that prevented the program from being built. We ran the tests with the command `make check` at the root of the repository.

4 Evaluation Results

We begin this section by discussing bugs we encountered and fixed during testing. We then move on to present the function symbols that we needed to manually include in LibFilter's analysis of several binaries in order to run specific tests, due to known limitations discussed in Section 2.2. Next, we report the results generated by this updated version of LibFilter and discuss them. Finally, we discuss challenges faced during testing as well as present a future work direction.

4.1 Bug Fixes

During the course of our testing, we discovered a small programming bug within LibFilter's analysis that led to a tenth of the Nginx test suite to fail, due to the incorrect erasure of a single common function within Nginx's SSL code.

The edge case occurs duing the FCG generation phase of the analysis, when the function being analyzed contains a `jmp` instruction whose target is to the middle of another function (as opposed to its beginning). The only instance we observed such a case was with the incorrectly erased function itself: `bn_mul4x_mont` of `libcrypto.so.1.1`, which is jumped to just 6 bytes beyond its first instruction.

The bug itself was the absence of a single line of code that records the target function as a callee of the one currently being analyzed. This resulted in a single edge being accidentally omitted from the resulting FCG, partitioning the target function's subgraph from the rest of the FCG, and causing it and its callees to be incorrectly erased by the analysis.

The fix for this bug consisted of a simple four line patch that adds in the missing statement to the code, after which all affected Nginx test files (all testing SSL features) were passed upon rerunning the test suite.

This scenario illustrates the purpose of this work. Small errors such the single forgotten line of code could lead to the complete unavailability of a program feature. Thus, the tolerance for error in erasing programs is small.

4.2 Manually-included Symbols

Certain tests within several of the suites made use of function symbols that were dynamically included at runtime through interfaces such as `LD_PRELOAD` or `dlsym`. To run these tests, we needed to manually include these dynamic symbols to the analysis of some binaries. Table 1 lists the binaries that required these additions, what symbols were added, and which test file they were needed for.

In total 7 binaries (6 in Coreutils, and the `nginx` binary in Nginx) required for at least one symbol to be included manually.

4.3 Test Results

After fixing the bugs and including the symbols discussed in Sections 4.1 and 4.2, we evaluated the binaries erased with the updated version of LibFilter on the test suites of *GNU Coreutils*, *Nginx*, and *GNU m4* on an x86_64 Debian system.

Table 2 shows the results for each of the test suites. *Baseline* refers to to the test results for the original, unmodified version of the binaries, and *Erased* refers to the test results of the binaries after being analyzed and erased by LibFilter.

Of the three test suites, only the m4 suite experienced no change in results before and after erasure, indicating that there were no (visible) errors in the analysis. In contrast, the Nginx suite experienced a previously passed test fail, and the Coreutils suite experienced a previously skipped test pass after their respective binaries were erased.

Skipped Coreutils Test Upon closer examination, we found the additional pass in the Coreutils test suite to be a result of the differences in how the baseline and erased tests were run in our testing system. The test file in question (`tests/misc/stty-row-col.sh`) requires control of the input terminal in order to be run, otherwise skipping the test. Running the test directly using the Coreutils test interface instead of through our test script resulted in a pass for both baseline and erased version of the Coreutils binaries.

Failed Nginx Test We found the failed test case (`perl_sleep.t`) to be the result of a runtime-loaded library used by Nginx's optional Perl module to provide callback functionality. Since automatically analyzing dynamically loaded symbols is out of scope for LibFilter, the normal course of operation would be to manually include the required symbol into the analysis, which allows for testing of the rest of the file. However, we faced an unexpected complication regarding the symbol needed to be loaded. The symbol and its code were located within the main Nginx executable, and, to provide the symbol, the Perl module would effectively load the main Nginx executable *itself* as a shared library to do so. Attempts to manually include the required symbols were met with assertion failures within the Egalito tool, making the results for this test inconclusive.

4.4 Challenges

In this section we describe some of the challenges we encountered during testing.

Symbol-resolution Order: Loader vs. Egalito The original LibFilter paper [10] made mention how the different symbol resolution orders between Egalito and the system loader would lead to the incorrect code bytes being erased when a symbol with the same name exists across multiple libraries. The example described used the `read` symbol, which exists in both `libc` and `libpthread` but with the code in both libraries representing separate implementations. When the program was run, the system loader would bind the `read` symbol to the `libc` implementation, while during analysis Egalito would bind the symbol to the `libpthread` version, leaving it unerased if the `read` symbol is within the program's FCG. This leads to a crash at runtime when the `libc` version of the symbol is used, but was marked unused and erased by Egalito.

The paper mentioned the bug to be patched 'soon' within the Egalito repository, but as of the version of Egalito we used in our testing (which has commits up to March 24th, 2020), we found this has yet to occur.

Debug Symbols Another challenge we faced was the need to have the debug symbol of shared libraries present on our system in order for the disassembly performed by Egalito to be successful.

If Egalito detects that debug symbols are present for a given ELF (e.g. the corresponding entry for the listed build id exists in `/usr/lib/debug/.build-id`), then the tool would automatically make use of them to guarantee a correct disassembly. In the absence of these symbols, however, Egalito would attempt the disassembly itself, regardless. During our tests, we discovered that for Nginx, debug symbols were needed for its shared libraries in order for the analysis tool not to crash. Lack of these symbols would lead to assertion failures within the Egalito, implying an incorrect disassembly. We note, however, that only shared libraries needed to have their debug symbols present on the system, and not the main executables themselves.

5 Future Work

There were a number of files for each test suite that were skipped, due to a lack of root permissions on the test system. The test files either needed to be run as root directly, or require modifications to the system environment that require root permissions aside from `apt-get install` in order to run. Future work would involve installing the required software, making the necessary system environment configurations, and running the remaining tests as root for each of the test suites.

Binary (Suite)	*Symbol (Library)*	*Test File*
cp (coreutils)	__xstat (k.so)	tests/cp/nfs-removal-race.sh
hostid (coreutils)	__libc_readline_unlocked (libc.so.6)	tests/misc/help-version.sh
rm (coreutils)	readdir (k.so)	tests/rm/rm-readdir-fail.sh
uniq (coreutils)	setvbuf (libc.so.6)	tests/misc/stdbuf.sh
df (coreutils)	fopen (k.so)	tests/df/no-mtab-status.sh tests/df/skip-duplicates.sh
pinky (coreutils)	_nss_resolve_gethostbyname4_r (libnss_resolve.so.2) __libc_readline_unlocked (libc.so.6)	tests/misc/help-version.sh
true (coreutils)	setvbuf (libc.so.6)	tests/misc/help-version.sh tests/misc/stdbuf.sh
ls (coreutils)	getxattr (k.so) lgetxattr (k.so) print_call_count (k.so)	tests/ls/getxattr-speedup.sh
nginx (nginx)	Perl_dowantarray (libperl.so.5.30) Perl_sv_setref_pv (libperl.so.5.30) zlibVersion (libz.so.1z)	gunzip_perl.t perl_gzip.t

Table 1: Symbols manually-included per binary, and associated test files

Test Result	**Coreutils**			**Nginx**			**m4**		
	Baseline	*Erased*	*Delta*	*Baseline*	*Erased*	*Delta*	*Baseline*	*Erased*	*Delta*
Pass	544	545	+1	291	290	-1	151	151	+0
Skip	73	72	-1	65	65	+0	19	19	+0
Fail	0	0	+0	0	1	+1	0	0	+0
Total	617			356			170		

Table 2: The number of test files passed, skipped, or failed for each test suite before and after its binaries were erased, represented by the *Baseline* and *Erased* columns, respectively. *Delta* corresponds to the increase/decrease in number of test files from *Baseline* to *Erased*

6 Conclusion

We presented a method for verifying LibFilter's correctness through the use of several open source test suites: *GNU Coreutils*, *Nginx* and *GNU m4*. We have shown that, after a bug fix in the analysis tool and inclusion of certain runtime-loaded symbols for the analysis, LibFilter is correct enough to pass most of these test suites. We also described challenges, noting how a previous issue mentioned to be fixed 'soon' in the original paper regarding Egalito and system loader symbol resolution orders still needs to be rectified, as well as show the requirement of debug symbols to be present for certain shared libraries in order to be able to successfully disassemble them. Lastly we mention the possibility of furthering this work by running the remainder of skipped tests, which require system root permissions to run.

References

[1] Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/.

[2] Coreutils Github mirror. https://github.com/coreutils.

[3] glibc change workaround patch for m4 v1.4.18. http://git.openembedded.org/openembedded-core/plain/meta/recipes-devtools/m4/m4/m4-1.4.18-glibc-change-work-around.patch.

[4] GNU M4. https://www.gnu.org/software/m4/m4.html.

[5] GNU M4 FTP Directory. http://ftp.gnu.org/gnu/m4.

[6] Nginx Github mirror. https://github.com/nginx/nginx.git.

[7] Nginx News. https://nginx.org/.

[8] Nginx-Tests Github mirror. https://github.com/nginx/nginx-tests.

[9] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated system call filtering for commodity software.

In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), Donostia/San Sebastian, Spain*, 2020.

[10] Benjamin Schteinfeld. *LibFilter: Debloating Dynamically-Linked Libraries through Binary Recompilation*. Brown University, 2019.

[11] D. Williams-King, K. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. Wu, J. Yang, and V. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020. http://www.cs.columbia.edu/~junfeng/papers/egalito-asplos20.pdf.