FrankenSSL: Recombining Cryptographic Libraries to Create Software Variants

Author: Bheesham Persaud me@bheesham.com
Supervisor: Dr. Anil Somayaji, School of Computer Science
Course: COMP 4905: Computer Science Honours Project

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Computer Science, Honours

at

School of Computer Science
Carleton University
Canada
August 18, 2016

Abstract

The need for diversity in the software we use is on the rise. As methods of how to automatically find and exploit vulnerabilities evolve, so must our defenses.

Methods such as Address Space Layout Randomization and N-version programming are efforts to further software diversity in, an otherwise, monoculture.

FrankenSSL is a case study of if it is possible to recombine different implementations of software in order to gain security through diversity, in which any advantage that an attacker may gain by finding a vulnerability is thwarted by the effort required to actively exploit it.

We will be looking at the OpenSSL project and its forks, BoringSSL and LibreSSL, and will evaluate the feasibility of recombining portions of the three libraries in order to create variants of them which may render some vulnerabilities useless.

Acknowledgements

I would like to thank Dr. Anil Somayaji for supervising and giving much needed guidance on this project; members of the Carleton Computer Security Lab, including Borke Obada-Obieh, Nilofar Mansourzadeh, and Ashley Moni for the seemingly endless hours of help I received while writing a related paper; my friends and family who spent hours proof-reading my many drafts; and my parents, Lelawattie and Shradanand, without whose support this would not be possible.

Additionally, I would like to acknowledge the support of Canada's Natural Sciences and Engineering Research Council (NSERC) through their Discovery grants program.

Contents

List of Tables

Chapter 1

Introduction

The search for vulnerabilities is becoming easier. Recently, the automated search for vulnerabilities in computer software has become viable to use in real-word applications. As recently displayed in the DARPA Cyber Grand Challenge, we now live in an age where servers can automatically find and exploit vulnerabilities in software [1]. As methods improve to find vulnerabilities in computer software, so should our defenses and mitigations. To address the discovery of new vulnerabilities in software, most Internet-connected platforms have extensive resources devoted to developing and deploying patches for software vulnerabilities. Regardless of our best efforts to mitigate the increased discovery of software vulnerabilities, many systems go unprotected, and even "protected" systems fall prey to undiscovered attacks – zero-day vulnerabilities.

A part of why computer security is difficult is because a large percentage of computers are running the same software, creating a monoculture of sorts amongst computers. This creates a situation whereby finding a vulnerability that affects any one host may affect many other hosts, thus making a favourable situation for attackers. This situation puts attackers in a position

where they have a high return on investment, that is to say: attackers gain more resources than what they spent to conduct the attack. One approach to mitigate this is to introduce "randomness", or diversity, into software. Diversity makes software less predictable, and thus harder to deterministically exploit some classes of vulnerabilities.

There are a few notable projects which aim to improve security guarantees, or "harden", software, such as PaX, Grsecurity, SELinux, and various compile-time options in modern compilers. The PaX, Grsecurity, and SELinux projects modify the Linux kernel to employ, amongst other runtime defenses, principles of least privilege policies for processes' memory spaces, and Address Space Layout Randomization (ASLR). ASLR is a method which randomizes a running process' memory layout, making attacks which rely on this knowledge difficult, if not near impossible, to successfully execute. Modern compilers such as the Gnu project C and C++ compiler and the clang LLVM compiler offer features to mitigate against buffer overflow attacks at compile-time.

When ASLR is used in tandem with the stack-smashing protection features offered in modern compilers, buffer overflow attacks and other classes of attacks, which depend on an attacker knowing specific things about a process' address space, can be detected and possibly mitigated. In the worst case scenario, most defenses tell the running process to panic and shut down immediately in an attempt to mitigate malicious access to the systems resources.

However, regardless of our best efforts to defend against the pitfalls of predictability, attackers can, to some degree, reverse any randomization and diversity added to an application applying derandomization attacks [2] to the process' memory layout and spraying the heap [3]. The derandomization attacks described by Shacham et al., "converts any standard buffer-overflow exploit into an exploit that works against systems protected by address-space

randomization." [2] Heap spraying is where an attacker, "attempts to fill up a large fraction of memory in a way that increases the likelihood of reaching a target object." [3] Attackers have the advantage of being able to inspect and study the behaviour of running software, and thus are able to make predictions on how it will behave when given certain inputs. If diversity were achieved at the source or object level, instead of at runtime, attackers would lose the ability to accurately and precisely predict how software behaves since different versions of the software may be running.

The need for software diversity has been noted by many researchers before [4, 5]. Research done by Forrest et al. describe methods for building diverse software by adding or removing nonfunctional code; reordering code; and reordering a programs memory layout [6]. Baudry et al. discuss the history of software diversity in great detail, and describe methods of how to achieve it in many different ways [7]. We will be focusing on design diversity, or N-Version programming as defined by Chen et al. [8].

This report presents our findings when recombining libraries which implement SSL/TLS functionality. Specifically, we recombine parts of OpenSSL [9] with two of its forks, BoringSSL [10] and LibreSSL [11, 12]. Our preliminary results indicate that a recombination at the Application Programming and Binary Interface (API and ABI, respectively) may yield some positive results. The research presented here is the precursor to what Persaud et al. had presented at the Annual Symposium on Information Assurance [13].

The rest of this paper proceeds as follows: we first describe background and related work in Section 2; we discuss our implementation details in Section 3; and our preliminary results in Section 4; our limitations and areas for future research in Section 5; and finally, conclude in Section 6.

Chapter 2

Background

2.1 Compilers and Libraries

The tools required to compile C code, the language this project is written in, are a compiler and a linker. For this project we used the GNU project C and C++ compiler due to its versatility.

Additionally, there are several intermediate states that code goes through before it gets executed. When code is compiled, a compiler will organize the output such that any dependent program will know where the code it requires is located and how to call it. This is, of course, a very simplified overview of the compilation and linking process.

Code is compiled into machine code and is placed in object files. The compiler will output, alongside the machine code, extra data such as sections and symbols to help the systems linker figure out how to call each function. A library is a bundle of object files.

We leverage the system's runtime linkers ability to change what each symbol points to at runtime; this lays the basis for the majority of our work.

2.2 Software Diversity for Security

Chen et al. defined N-Versioned programming as, "programing is defined as the independent generation of N $\geq$ 2 functionally equivalent programs from the same initial specification" in 1987 [8]. Since then, there have been numerous evaluations into the feasibility of such a method. In the experiment ran by Knight et al., they found "that the programs were individually extremely reliable but that the number of tests in which more than one program failed was substantially more than expected." [14] However, there is a high cost associated with creating N-versions of software: creating N-versions of software takes at least a factor of N times more than it took to create one version.

Due to the high cost of N-version programming, much of the research in the software diversity field has been led by what Forrest et al. [6] described, where compile-time and runtime diversity are added such that source code semantics of applications will be preserved [7]. Attempts to add diversity to systems have been implemented at many levels runtime, including:

- instruction set: "use a machine instruction set that was both unique and private" [15, 16];
- memory address: "randomizes the location of victim program data and code" [17]; and
- system calls: a combination of instruction set randomization (ISR) and address space layout randomization (ASLR) [18, 19].

2.3 OpenSSL, BoringSSL, and LibreSSL

The security of OpenSSL has been called into question numerous times. Most of the Common Vulnerabilities and Exposures (CVEs) that are disclosed that

have to do with OpenSSL are because of human error. Below are but a few high-profile CVEs that were disclosed.

CVE-2003-0147 14th March 2003. "Researchers have discovered a timing attack on RSA keys, to which OpenSSL is generally vulnerable, unless RSA blinding has been turned on." [20]

CVE-2008-0166 9th January 2008. "A vulnerability in the OpenSSL package included with the Debian GNU/Linux operating system and its derivatives may cause weak cryptographic keys to be generated." [21]

CVE-2014-0160 7th April 2014. "A missing bounds check in the handling of the TLS heartbeat extension can be used to reveal up to 64k of memory to a connected client or server." [22]

Heartbleed, being the last of many straws for a large amount of developers [23], motivated both Google and a group of OpenBSD developers to create their own forks of OpenSSL: BoringSSL [10] and LibreSSL [12], respectively.

BoringSSL diverges more from OpenSSL than LibreSSL, as Google developers are willing to make breaking changes in order to easily accommodate their own requirements. However, the goals of the developers behind LibreSSL are to keep API and ABI compatibility whenever possible while simplifying and massaging the code to meet their standards. Many of the unsecure algorithms and defaults have been deprecated, if not removed completely. There have also been efforts to create a simplified TLS interface, dubbed libtls [24].

Heartbleed did not take away from the popularity of OpenSSL; if anything, it increased the number of people reviewing code and contributing to the project. This is strange, because generally when forks are created the parent project often sees a dip in popularity. This unique situation creates the ideal environment in which to experiment with software recombination.

2.4 Software Recombination

In order to reap the benefits from the now diverse code of OpenSSL and its related forks, we will focus our efforts on creating diverse implementations of the TLS protocol. Our goal follows the areas for future research as proposed by Foster et al. in his presentation of ObjRecombGA [25]. ObjRecombGA used a genetic algorithm to find code which satisfied various fitness functions. This approach successfully combined functionality between closely related variants of certain software, including Quake, a game; GNU sed, a UNIX command line utility; and Dillo, a web browser. Foster et al.'s method of recombination worked at the the level of generated object files. However, this method required minor modifications to the source code, a measure that we are refraining from in our research.

There are specific versions of OpenSSL which are Federal Information Processing Standard (FIPS) Publication 140-2 accredited. Any changes to the source code would require that new version to be accredited and go through varying degrees of auditing: not a process that is cheap. As such, it is desirable to create recombinations that still retain their FIPS Publication 140-2 accreditation.

We propose to recombine at the API, or function, level because we believe it will have the biggest impact on security. Popular libraries are likely to be used in many applications, and if a vulnerability is found in those libraries then mitigating it will improve the security of any dependent softwares. In Section 3, the next section, we will go into detail the design and implementation details of our project, FrankenSSL.

There are three significant implementation details that must be ironed out when combining software which export overlapping functionality. The first is scaffolding, or how each of the libraries will live on the hosting system. The second is build infrastructure, or how each of the libraries will be compiled

and built. The third is symbol resolution, or how a dependent application will know which function from which implementation to call.

Scaffolding. The implementation libraries must be able to co-exist on the same system. A vanilla installation of either OpenSSL, BoringSSL, or LibreSSL will completely overwrite any existing installation of any other – that is, they cannot co-exist on the same system without some modification to their default installation location.

Build Infrastructure. The implementation libraries must be built similarly. Each library must be built with similar features enabled to ensure that our wrapper library – which will be discussed in the next section – will not depend on any deprecated functions.

Symbol Resolution. The implementation libraries must export similar symbols and the wrapping library must know how to call any one of the implementations.

Chapter 3

Methodology

While our design and methodology are geared towards working with OpenSSL; BoringSSL; and LibreSSL, the same design and method can be generalized and applied to any other library that has at least one fork.

First, each library was configured similarly such that neither will expose features not present in any other. This lays the foundation for the overlapping functionality that we will wrap. This ensures that each library offers the same features and any library or application requiring any offered features is guaranteed to have an implementation.

Second, each library was built in separate subdirectories, without any modifications to their source code, which, as we previously mentioned, was one of our design goals.

Third, wrapper code was then written for each function in each library we wanted to make permutations with. The function we chose were:

- `SSL_load_error_strings()`,
- `SSL_library_init()`,

- `TLS_client_method()`,
- `SSL_CTX_new()`,
- `SSL_CTX_free()`, and
- `SSLeay_version()`.

Fourth, we created a stub library as a placeholder for applications to link with. This supplied dummy implementations of each of the implementations in addition to some debugging features.

Fifth, a test program created in order to evaluate permutations of the combinations of each library. To test we created a program which utilizes each of the symbols we wrote wrapper functions for. From here, we employ runtime linking via the `LD_PRELOAD` environment variable. This allows us to, at runtime, link and test the desired function implementation.

The test program was then ran through the GNU Debugger and valgrind to find any defects. See table 4 for tablularized results.

Chapter 4

Results

We have tested all combinations of the functions we wrote wrapper code for. The results reflect testing on only the production code – that is, the functions that were not used for debugging and analysis purposes. Our results are summarized in Table 4.

Initialization	Deinitialization	Result
OpenSSL	OpenSSL	Works
OpenSSL	BoringSSL	SIGABRT
OpenSSL	LibreSSL	SIGSEGV
BoringSSL	BoringSSL	Works
BoringSSL	LibreSSL	Leak
BoringSSL	OpenSSL	Leak
LibreSSL	LibreSSL	Works
LibreSSL	BoringSSL	SIGABRT
LibreSSL	OpenSSL	SIGSEGV

Table 4.1: Results from our tests.

In addition to our tablular data we have debug logs from running each of the

test program, `bin/init`, under different conditions. We can show what the runtime linker sees is true by making note of which libraries it reports the current process has linked.

In this example, we show that the `bin/init` program uses all three implementations of the libraries.

```
$ LD_PRELOAD="lib/libhstopen_init.so lib/libhstboring_deinit.so lib/libhstlibre_version.so" ldd ./bin/init
        linux-vdso.so.1 (0x00007ffd6755a000)
        lib/libhstopen_init.so (0x00007f149ba99000)
        lib/libhstboring_deinit.so (0x00007f149b896000)
        lib/libhstlibre_version.so (0x00007f149b694000)
        libhststub.so => /.../hst/libhst/build/lib/libhststub.so (0x00007f149b492000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f149b0a9000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f149aea5000)
        libssl.so.1.1 => /.../hst/libhst/open/../../openssl/libssl.so.1.1 (0x00007f149ac3b000)
        libcrypto.so.1.1 => /.../hst/libhst/open/../../openssl/libcrypto.so.1.1 (0x00007f149a7df000)
        libssl.so => /.../hst/libhst/boring/../../boringssl/build/ssl/libssl.so (0x00007f149a596000)
        libcrypto.so => /.../hst/libhst/boring/../../boringssl/build/crypto/libcrypto.so (0x00007f149a1f7000)
        libcrypto.so.37 => /.../hst/libhst/libre/../../libressl/crypto/.libs/libcrypto.so.37 (0x00007f1499e1d000)
        /lib64/ld-linux-x86-64.so.2 (0x000055562a483000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f1499bff000)
        libresolv.so.2 => /lib64/libresolv.so.2 (0x00007f14999e4000)
```

Additionally, we can show that the program can use functions from the different implementations at runtime. We can see the output of running `bin/init` reporting that it is using LibreSSL, even though it is using an calling `hst_tls_init` and `hst_tls_deinit` which use OpenSSLs implementation to initialize and deinitialize.

```
$ LD_PRELOAD="lib/libhstopen_init.so \
              lib/libhstopen_deinit.so \
              lib/libhstlibre_version.so" \
  ./bin/init

HST: using libre's impl of: hst_tls_version
Using version: LibreSSL 2.3.2
HST: using open's impl of: hst_tls_init
HST: using open's impl of: hst_tls_deinit
```

These results are further discussed in Section 5.

Chapter 5

Discussion

While it was possible to build a framework which allows for the combination of arbitrary, overlapping functionality, we chose, instead, to be surgical in our approach.

Applications which depend on OpenSSL for TLS will generally call the same functions in the same order: library initialization, context initialization, session initialization, read and write functions to communicate securely, session deinitialization, context deinitialization, and library deinitialization. As such, we chose to wrap the functions that would have the greatest effect on any dependent software.

Both LibreSSL and BoringSSL had made major modifications to the way they were each built, namely that they stopped exporting versioned symbols – a required feature if we would like to link at runtime without any library inadvertently overwriting functionality from another.

One interesting point to note, is that at a high-level, BoringSSL has diverged from the OpenSSL source the most. Additionally, all the errors with the recombinations of the libraries arose from the divergence of how each library

stores session state, specifically with code that relates to the `SSL_CTX` data structure. The `SSL_CTX` data structure is at the core of how each of these libraries operate, and thus making any changes to it inherently makes interoperability with a different implementation difficult.

We had numerous options at our disposal for how we would recombine the libraries. Here, we will outline the three main ones we evaluated and why, ultimately, we decided to choose symbol versioning.

objcopy. Objcopy is a part of the GNU Development Tools package, and modifies the innards of object files. It is capable of, amongst other things, renaming and removing symbols in object files. We could have used objcopy to rename the symbols such that none of the symbols from any of the libraries collided with any other. This method, however, would mean that our goal of not modifying the resulting code in a way which would require re-certification would not have been able to met.

Additionally, with each symbol that was renamed, a corresponding header file would have to be created, meaning that the required work to maintain the codebase would be increased as there would be many fragile dependencies. However, this method would be equivalent to writing our own compiler pass, albeit without any of the benefits of having the a compiler and all of its infrastructure at your disposal.

Compiler pass. Modern compilers are complicated beasts. Writing a compiler pass requires deep knowledge of the inner workings of compilers and linkers. Additionally, it raises the entry barrier for anyone looking to reproduce the results of the research presented here, as it would require them to use a modified compiler and linker.

While this option, as restricting as it may be, is not entirely unreasonable, introduces many layers of complexity as there are many moving parts to modern compilers. Added complexity is not a desirable outcome, especially

when working with security-centric software, thus it should be avoided if possible.

Symbol versioning. Symbol versioning offered a solution that was neither as cumbersome as writing a compiler pass nor as pervasive as objcopy.

Using Dreppers, "How To Write Shared Libraries" [26] as a guide for creating shared libraries, we decided that versioned symbols would meet our needs perfectly. Adding symbol versioning to each of the projects was relatively easy, though it did modify the resulting object files. The modifications, however, were limited to extra metadata associated with each of the symbols and did not affect the code in any way.

Exporting the versioned symbols allowed paved the path for us to be able to effectively use our wrapper code, as now we could guarantee that none of the symbols would collide.

Chapter 6

Conclusion

FrankenSSL is an effort to create permutations of OpenSSL, BoringSSL, and LibreSSL in order to create a library which exposes the same Application Programming Interface as OpenSSL, yet varies between implementations. In doing so we aim to reap the benefits of both compile-time and runtime diversity.

It is likely that we would have found more variants if we had automated the process. We could create a program to pick out colliding symbols, wrap them, and then try to recombine function calls from the different implementations automatically. This would greatly reduce the effort required as a computer can try many more combinations than a human can given the same amount of time.

Library-level recombination may hold promise in the future.

Bibliography

[1] DARPA, "Cyber grand challenge," https://s3.amazonaws.com/cgcdist/cfe/cgc-final_event-cfe-brochure.pdf, 2016, accessed August 17, 2016.

[2] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2004, pp. 298–307.

[3] J. K. David Evans, Anh Nguyen-Tuong, *Moving Target Defense*. New York: Springer New York, 2011, ch. Effectiveness of Moving Target Defenses.

[4] F. B. Cohen, "Operating system protection through program evolution," *Computers and Security*, vol. 12, no. 6, pp. 565–584, 1993.

[5] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier, "Cyberinsecurity: The cost of monopoly," in *Computer and Communications Industry Association Report*, 2003.

[6] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *the Sixth Workshop on Hot Topics in Operating Systems*. IEEE, 1997, pp. 67–72.

[7] B. Baudry and M. Monperrus, "The multiple facets of software diversity: Recent developments in year 2000 and beyond," *ACM Computing*

Surveys (CSUR), vol. 48, no. 1, p. 16, 2015.

[8] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, 1978, pp. 3–9.

[9] OpenSSL Software Foundation, "OpenSSL: Cryptography and SSL/TLS toolkit," https://www.openssl.org/, accessed May 17, 2016.

[10] Google, "BoringSSL," https://boringssl.googlesource.com/boringssl/, accessed May 17, 2016.

[11] OpenBSD, "LibreSSL," http://www.libressl.org/, accessed May 17, 2016.

[12] J. Wagnon, "Security sidebar: LibreSSL is forking OpenSSL," https://devcentral.f5.com/articles/security-sidebar-libressl-is-forking-openssl, May 5, 2014, accessed May 17, 2016.

[13] B. Persaud, B. Obada-Obieh, N. Mansourzadeh, A. Moni, and A. Somayaji, "Frankenssl: Recombining cryptographic libraries for software diversity," in *Proceedings of the 11th Annual Symposium On Information Assurance*. NYS Cyber Security Conference, 2016, pp. 19–25.

[14] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *Software Engineering, IEEE Transactions*, vol. SE-12, no. 1, pp. 96–109, 1986.

[15] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2003, pp. 281–289.

[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of*

the 10th ACM Conference on Computer and Communications Security. ACM, 2003, pp. 272–280.

[17] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." in *USENIX Security*, vol. 3, 2003, pp. 105–120.

[18] X. Jiang, H. J. Wang, D. Xu, and Y.-M. Wang, "Randsys: Thwarting code injection attacks with system service interface randomization," in *26th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 2007, pp. 209–218.

[19] Z. Liang, B. Liang, and L. Li, "A system call randomization based method for countering code-injection attacks," *International Journal of Information Technology and Computer Science (IJITCS)*, vol. 1, no. 1, p. 1, 2009.

[20] OpenSSL, "OpenSSL Security Advisory [17 March 2003]," https://www.openssl.org/news/secadv/20030317.txt, March 2003, accessed August 17, 2016.

[21] The CERT Division of the Software Engineering Institute, "Vulnerability Note VU 925211 - Debian and Ubuntu OpenSSL packages contain a predictable random number generator," https://www.kb.cert.org/vuls/id/925211, May 2008, accessed August 17, 2016.

[22] OpenSSL, "OpenSSL Security Advisory [07 Apr 2014]," https://www.openssl.org/news/secadv/20140407.txt, April 2014, accessed August 17, 2016.

[23] Codenomicon, "The heartbleed bug," http://heartbleed.com/, April 2014, accessed May 25, 2016.

[24] OpenBSD, "TLS_INIT (3)," http://man.openbsd.org/OpenBSD-current/man3/tls_init.3, accessed May 17, 2016.

[25] B. Foster and A. Somayaji, "Object-level recombination of commodity applications," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. ACM, 2010, pp. 957–964.

[26] U. Drepper, "How to write shared libraries," *Retrieved Jul*, vol. 16, p. 2009, 2006.

Chapter 7

Appendix

7.1 Source code

The source code for this project can be downloaded from the following GitHub repository: `https://github.com/bheesham/frankenssl`.