Impact of Software Obfuscation on Susceptibility to Return-Oriented Programming Attacks

Harshvardhan P. Joshi
hpjoshi@ncsu.edu

Aravindhan Dhanasekaran
adhanas@ncsu.edu

Abstract

Software obfuscation is a commonly used technique to protect software especially against white-box attacks, where the adversary controls the host on which the software runs. It is a form of security through obscurity and is commonly used for intellectual property and DRM (Digital Rights Management) protection, defense against reverse-engineering attacks, and even used by malware. Some of the popular obfuscating techniques, including changing the control flow graph and substituting simpler instruction sequences with complex instructions, may however make the obfuscated binary more vulnerable to Return-Oriented Programming (ROP) based attacks. We analyze the ROP gadgets present in both – obfuscated and un-obfuscated versions of well known binaries, to see if obfuscation makes ROP based exploits easier. Based on our study, we identify which particular obfuscation technique(s) results in more probable ROP attack gadgets.

1 Introduction

Software developers have two broad security concerns: (1) vulnerabilities that lead to exploitation of the software (e.g. buffer overflow); (2) reverse engineering (e.g. software or music piracy). The first concern is based on the threat model where the software runs on a benign host, and the goal is to protect the software in order to protect the host system and the information the software has access to. An example of this threat model is an unprivileged attacker exploits vulnerability in software to gain access to restricted information. The second concern however, is based on a very different threat model where the host or the privileged user is malicious and tries subvert the restrictions set by software developers. An example would be reverse engineering DRM protection software for music or movies piracy.

Some developers use static analysis tools to prevent vulnerabilities such as buffer overflow; however, many others rely on OS level protections such as ASLR and W⊕X. Protection against reverse engineering is generally achieved via software obfuscation tools and cannot leverage OS support.

Given the pervasiveness of ASLR and W⊕X in operating systems, attackers have begun using return oriented programming. Return Oriented Programming (ROP) is a technique through which an attacker can introduce changes to a program's control flow using many short code snippets, called gadgets, present in a program's address space. ROP allows attackers to exploit buffer overflow vulnerabilities even when ASLR or W⊕X protection is enabled. The goal of the attacker using ROP is to exploit a vulnerability (buffer overflow) in client software in order to compromise the host system. Thus, this is a technique used by attackers in our first threat model.

Even the software that requires protection against the second threat model should be secure against the first threat model in order to protect the benign hosts it may run on. However, some of the software obfuscation techniques used in practice may impact the vulnerability of the software to the first threat model. Some commonly used obfuscation techniques include adding dead or irrelevant code, and changing control flow graphs by splitting or merging functions and adding opaque predicates. We study the impact of software obfuscation on the program's vulnerability to ROP based attacks. An ROP based attack depends on stringing together gadgets in the code in order to perform arbitrary action. In [17], a catalog of x86 gadgets are identified for a Turing-complete ROP. A software with a larger number of gadgets from this catalog is more easier to exploit using ROP by an attacker, assuming they find an entry through vulnerabilities like buffer overflow. We analyze a set of open source software binaries in terms of the catalog of gadgets they contain, and compare this with the gadgets found in obfuscated versions of the same software. Based on our experiments, we show that obfuscation significantly impacts the number of gadgets in a binary and can potentially make an ROP attack easier.

The rest of the paper is organized as follows. Section 2 explains the background information for this work, including a brief introduction to return oriented programming, current state of defenses against ROP and about popular code obfuscation techniques. Section 3 provides a brief overview of our work, outlines the design and implementation details, followed by evaluation methods and

results in section 4. Section 5 includes related work and finally we conclude our work in section 6.

2 Background

2.1 Return Oriented Programming

Return Oriented Programming [17] is made possible by making use of gadgets. Each gadget ends with a 'return' or 'jump' instruction, which is used to chain together several such gadgets to alter the program's behavior and each gadget might perform a different operation, say a load, adding registers and a jump. On the whole, return oriented programming can be viewed as the generalized notion of return-to-libc [20] types of attacks. Current research is focused on defending, or eliminating, return oriented programming attacks as they are proven [12] to be ineffective against the major defense against execution of malicious code - W$\oplus$X protection model. In W$\oplus$X model, a memory page is either writable or executable, but not both, which prevents all types of code injection attacks. In return oriented programming attacks, the attacker does not inject any code and just alters the program execution by executing already existing code in an arbitrary fashion.

A return oriented program consists of several gadgets arranged carefully to meet the attacker's goal. These gadgets must be in the memory, in the address space of the executing program or in the address space of a library used by the program. Most of the gadgets fall under five broad categories: (1) load/ store (move to/from register/memory), (2) arithmetic and logical (add, sub, and, xor, bitwise and shifts), (3) control flow (jmp and ret), (4) system calls (int 0x80 followed by call, ret) and (5) function calls (call, ret). The instruction sequences that are chosen to be gadgets can be identified by the presence of 'ret' instructions or by identifying unintended instruction sequences. The latter is possible only in case of x86 architecture as it supports unaligned, variable length instructions. This can be exploited by moving the instruction pointer to an offset within the instruction, which changes the interpretation of the current and the following instructions until a 'ret' or a 'jmp' instruction is decoded. The gadgets need not to be contiguous and can be chained by means of return or jump instructions. The first in line gadget should be executed by redirecting the stack pointer, which can be accomplished by a simple buffer overflow.

2.2 Defenses Against ROP

The attacks that fall under return oriented programming paradigm are very broad, but still there are many defense mechanisms to mitigate or to prevent ROP based attacks. This section briefs few such mechanisms.

Code randomization [15] is a technique in which the intended instruction sequences, of a binary, to be used as a gadget, are randomized through in-place, narrow code transformations. The performance overhead is negligible as no new code is inserted. When used along with other code randomization procedures, such as, Address Space Layout Randomization (ASLR) [19], it disrupts 10% and 80%, effectively and probabilistically, of the instruction sequences in a binary. ROPdefender [10] is a tool that adds additional code to a binary, to observe the program's execution behavior, using just-in-time compiler. It can prevent all return instruction based attacks and detect unintended instruction attacks. It doesn't need the program source code as it uses a JIT compiler. The performance overhead of ROPdefender is almost 100%. XFI [11], based on control flow integrity [6], in addition to flexible access control and fundamental integrity guarantees, also offers stack protection with return address safety. XFI can protect return address based ROP attacks using intended instruction sequences.

2.3 Software Obfuscation Techniques

Software obfuscation is achieved by a sequence of transformations on the code in order to obscure the purpose of the code while maintaining its original behavior. There have been several obfuscating transformations identified in literature [14, 9]. These include inserting opaque predicates which are difficult to evaluate at compile time; inserting dead or irrelevant code; cloning, splitting or merging functions; control flow flattening; etc. More details about some of these obfuscation techniques are discussed in section 3.2.

3 Overview and Experimental Design

In this paper, we hypothesize that *obfuscation tools make software more susceptible to ROP attacks*. This is based on our observation that obfuscation tools add redundant code to the software, change control flows, add conditionals, etc. which can increase the number of gadgets in the binary.

In order to study the susceptibility of a software to ROP attacks, we identify the catalog of gadgets in the code. It may seem that a software with a larger number of gadgets would be more susceptible to ROP attacks. However, there is limited advantage of repetitive gadgets, and variety is more important in order to chain them together and accomplish a useful task. Hence, if a software contains a full catalog of gadgets that is Turing-complete, then we consider it to be more susceptible than a software with an incomplete catalog of gadgets.

To evaluate our hypothesis, we find and compare gadgets in obfuscated and unobfuscated versions of a large number of binaries. We use a set of open source software of different size and type including libraries. We use Linux as both the development and the target platform for the binaries, and use LLVM tool chain for compiler. This is in order to use an LLVM based obfuscation tool called Obfuscator-LLVM [2]. The tool performs obfuscation transformations on LLVM bytecode, and thus can work with a variety of LLVM front-end (e.g. C, C++, etc.) and back-ends (x86, MIPS, ARM, etc.). We build the obfuscated version of the software for Linux using the same clang compiler with the same flags as used to build the unobfuscated binaries. We perform our analysis on both these sets of binaries.

We use an ROP gadget finding tool, ROPgadget [4], to identify potential gadgets in binary. A list of all unique gadgets found in the unobfuscated binary is created, and then compared against the list from obfuscated binary. We have written our own tool to categorize these gadgets based on their functionality. We also look at the total number of gadgets found, and the total number of unique gadgets found in a binary.

3.1 Binaries Selection

We use two sets of software, GNU Coreutils [1] and OpenSSL [3] as target binaries for our evaluation. Since we believe that the increase in size of binary due to obfuscation may impact its ROP susceptibility, we want to evaluate on binaries of different sizes. In order to be realistic, we should use binaries that are likely to be obfuscated. A DRM protection software is a prime candidate for this, however, such software is unlikely to be distributed as open source. We decided instead to use software that is security sensitive but openly available. We use OpenSSL, and specifically its libraries lissl and libcrypto as an example of commonly used security sensitive libraries. We tried to use glibc, which is more commonly used but it does not compile under LLVM toolchain.

GNU Coreutils include utilities like `cp, mv, ls, date`, etc. which are included in nearly all Linux distributions. Some of these utilities can also be considered to be part of these Linux distribution's trusted computing base (TCB) since they get frequently run as root. The Coreutils package includes 106 utilities of varying size and thus provides us with a good base for our evaluation. For more detailed analysis we select a subset of these utilities again based on their security sensitiveness, and their potential to do harm. These include `link, chroot, shred, touch, date, cp`. We select `touch` and `date` due to recent vulnerability CVE-2014-9471 that may allow arbitrary code execution or denial of service attack. We select `shred` due to expectation of secure removal of files, while the others (`link, cp, chroot`) for their potential for misuse.

3.2 Obfuscation Tool

While there are several software obfuscation tools available, we had difficulty finding suitable tools for our purpose. Software obfuscation is particularly popular for Java programs, since they are easy to reverse engineer, and a large number of tools and techniques are designed for Java. Majority of obfuscation tools for C/C++ are neither open-source nor free. The Tigress [5] is one of the free tools available for C, with many obfuscation transforms. However, it is not open source and does not seem to be actively maintained. Obfuscator-LLVM [2] is an open source project to build an obfuscator for LLVM toolchain. It works on LLVM's intermediate representation (IR) level, and so it can take advantage of LLVM's front-end and back-end which support many languages including C and C++, and architectures such as x86, arm, mips, etc. Since, this is a fairly new project, it has limited obfuscating transforms available. These are:

- Instruction Substitution (SUB): This obfuscation technique relies on substituting one set of instructions with another set of instructions while maintaining the same functionality. In Obfuscator-LLVM, this obfuscation substitutes standard binary operators (like addition, subtraction or boolean operators) by functionally equivalent, but more complicated sequences of instructions. This is a basic, straightforward obfuscation and does not add a lot of security.

- Control Flow Flattening (FLA): This obfuscation flattens the control flow graph of the program so that the structure of the program cannot be easily understood by static analysis like disassembly. This is achieved by identifying and moving blocks in a function which are at nested levels, next to each other. The selection of control flow to a particular block is done using a switch statement and a control variable that keeps track of the state of the program.

- Bogus Control Flow (BCF): This obfuscation modifies a function call graph by adding a basic block before the current basic block. This new basic block contains an opaque predicate and then makes a conditional jump to the original basic block.

4 Results and Discussion

As discussed in section 3, we perform our evaluations on binaries found in the GNU Coreutils package as well as the libraries in OpenSSL. We built GNU Coreutils and

OpenSSL using LLVM 3.4, and also built the same binaries with obfuscations using the same compiler. We built different versions of the obfuscated binaries, for different types of obfuscations supported by Obfuscator-LLVM and discussed in section 3.2, and a binary with all the obfuscations. In our discussion, unless otherwise specified, an obfuscated binary refers to a binary built with all the supported obfuscations.

While the OpenSSL libraries libssl.a and libcrypto.a are a collection of object files, we need an ELF binary in order to find ROP gadgets. We created a small test program and then statically linked the objects from both binaries with this program to create a binary which we can use for evaluation. We refer to this combined binary of objects from libssl and libcrypto as libssl.

Once we identify gadgets in a binary, we also categorize these gadgets based on the instructions they contain and the function they can serve. The categories are

- Memory: These are instructions and gadgets that facilitate load and store operations from/to memory and registers.
- Arithmetic: These are gadgets that contain arithmetic instructions and which can be used to perform operations such as add, sub, neg, etc.
- Logic: These are gadgets that contain instructions or can be used to perform operations such as and, or, xor, shift, rotate, etc.
- Control: These are gadgets that can control the flow, such as conditional or unconditional jumps.
- Other: These are gadgets that we could not categorize.

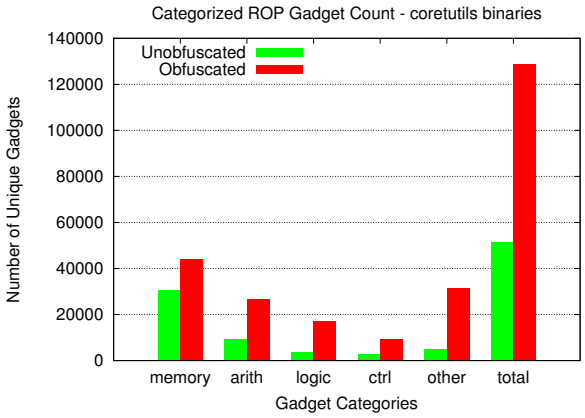

Figure 1: Gadget count for 'coreutils' binaries

Figure 1 shows the categorized as well as the total gadget counts aggregated for all the binaries in Coreutils with

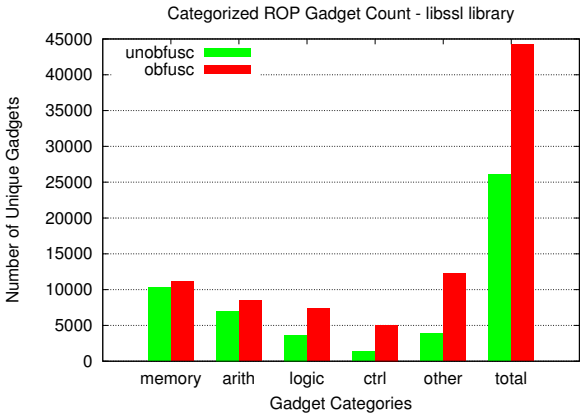

Figure 2: Gadget count for 'libssl' library

and without obfuscation. We can see that obfuscation more than doubles the total gadgets in the binaries, however the impact varies across categories. Similar graph for libssl is shown in Figure 2, where the total increase is more moderate. We believe this was because the obfuscator we are using was not able to handle some of the more complicated and large source files in libssl, and hence abandoned obfuscations on them. However, both these graphs show that obfuscation significantly increases the number of gadgets in a binary.

4.1 Impact of Obfuscation Type

Binaries	unobfusc	sub-obfusc	bcf-obfusc	fla-obfusc	all-obfusc
link	44	48	81	92	149
chroot	59	59	111	131	223
shred	85	97	169	185	313
date	101	113	217	277	473
cp	169	185	381	425	757
coreutils	11264	9318	17408	20480	41984

Table 1: Binaries size (in KBs) with different obfuscation techniques

Table 1 shows the binary sizes for some of the utilities in GNU Coreutils along with their size with different obfuscations. The instruction substitution obfuscation is referred to as sub-obfusc (or SUB), bogus control flow as bcf-obfusc (or BCF) and control flow flattening as fla-obfusc (or FLA), and details of these obfuscations are discussed in section 3.2. As expected, substitution does not have as big an impact on binary size as the other two. The size increases in binaries with full obfuscation ranged from about 3x to 5x.

The impact of different obfuscation types on gadgets is shown in Figure 3. It is clear that while overall the impact of BCF and FLA are higher than SUB, their impact within specific categories of gadets varies. The percentage increase for different types of obfuscations across categories is shown in Table 2. Since aggregating across

Categories	sub-obfusc	bcf-obfusc	fla-obfusc	all-obfusc
memory	4.70	17.80	12.87	43.86
arith	32.16	180.85	137.09	189.01
logic	56.82	85.52	112.43	352.41
ctrl	19.12	85.73	111.81	219.62
other	98.96	117.09	258.99	552.37
total	23.08	65.20	71.11	150.16

Table 2: Increase in gadgets found in Coreutils as a percentage of gadgets in unobfuscated binaries

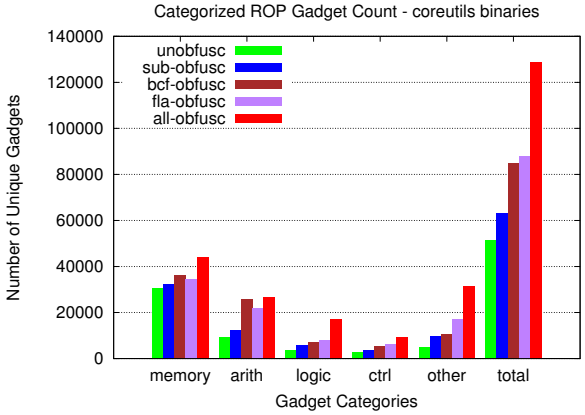

Figure 3: Different types of obfuscation for all 'coreutils' binaries

all of Coreutils binaries is difficult to analyze, we do more detailed analysis on a few selected binaries in Coreutils as identified in section 3.1. The graphs for `cp` and `link` are shown in Figures 4 and 5. We can see that on certain binaries like `link` BCF obfuscation has significantly more gadgets than FLA, but overall both types of obfuscation have similar impact on increase in gadgets.

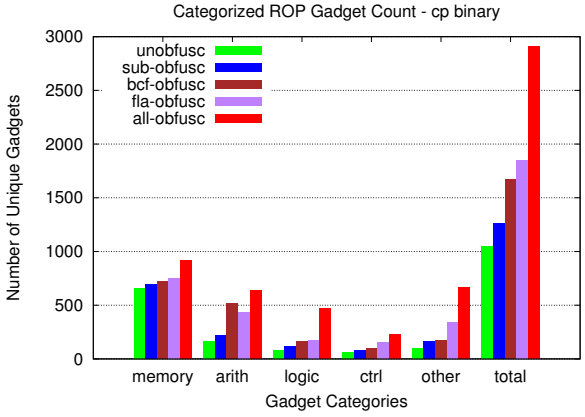

Figure 4: Different types of obfuscation for 'cp' binary

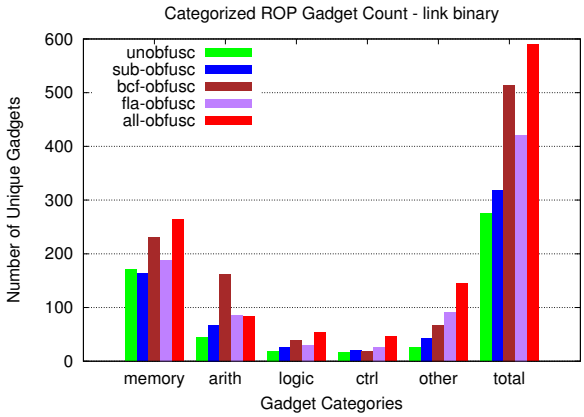

Figure 5: Different types of obfuscation for 'link' binary

4.2 Impact on Gadget Categories

The impact of obfuscations on different types of gadgets is not evenly distributed. We have observed that in general binaries have plenty of memory gadgets but relatively few logic and control gadgets. The impact of obfuscation is more pronounced on these fewer logic and control gadgets than on memory gadgets. This is not only because of the smaller baseline, but also because of the characteristics of the obfuscations. Both, bogus control flow and control flow flattening obfuscations significantly increase control flow and logic related instructions like conditional and unconditional jumps. For memory gadgets, the increase with obfuscations is comparatively small as can be seen in Figure 6. However, for logic and control gadgets, the increase in numbers is quite high and can be more than 6x the number of gadgets in unobfuscated binaries. As we can see in Figures 7 and 8, small binaries like `link` have very few (nearly single digit) logic and control gadgets and obfuscation can push the number up 4 to 5 times. This can potentially impact the feasibility of ROP attack on that binary.

In addition to analyzing the number of gadgets, we looked at the actual gadgets for these binaries in order to do a qualitative analysis. We compared the gadgets found in each utility against their obfuscated binaries. There were some common gadgets between these binaries, usually at the lower memory address where the common preambles in ELF can be expected. However, the majority of the gadgets were very different in both sets of binaries and it was difficult to make any generalizations about them.

4.3 Limitations and Future Work

We discuss the limitations of our work here and also how it can be extended in future.

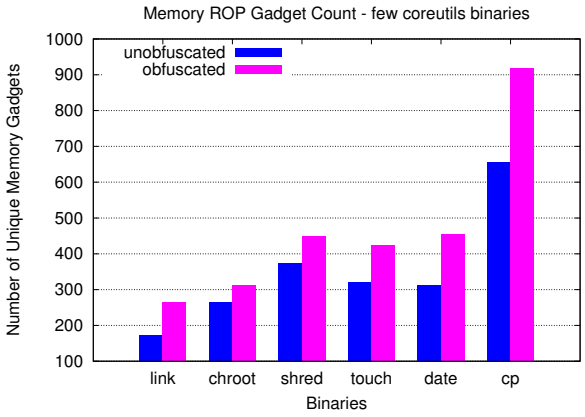

Figure 6: Memory gadget count for few 'coreutils' binaries

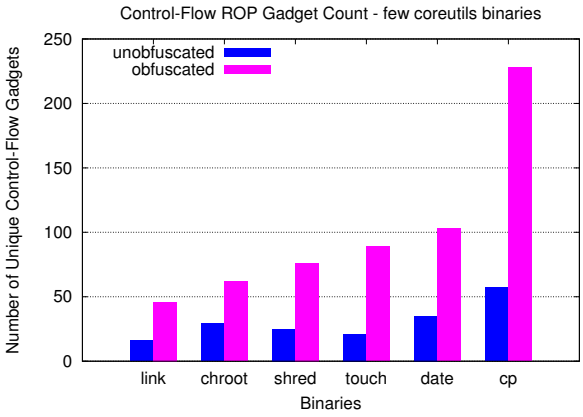

Figure 8: Control-flow gadget count for few 'coreutils' binaries

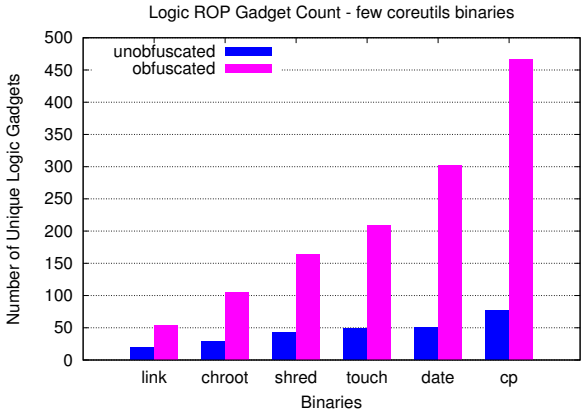

Figure 7: Logic gadget count for few 'coreutils' binaries

- We have selected our sample binaries for evaluation based on a combination of ease of availability and security sensitivity. Evaluation on binaries that are actually likely to be obfuscated would have been more interesting. Such binaries, like DRM protection software, are difficult to acquire in unobfuscated form since that will completely defeat the purpose of obfuscation.

- We have shown that for certain types of gadgets (logic, control), obfuscation can significantly increase the number of gadgets available. We have also shown that for small binaries, such increase can potentially affect the feasibility of ROP attack. Due to the limited time available to complete this project, we have not been able to show that obfuscating a binary changes gadgets found from non-Turing-complete to Turing-complete.

- It would be interesting if we can devise an actual attack on an obfuscated binary and show that it is more difficult or impossible on un-obfuscated binary. We have not attempted this, again due to limited time.

- We have evaluated only one obfuscation tool, with limited set of obfuscation transforms. It would be interesting to compare the results we have obtained against other obfuscators. However, there are very few open source or free obfuscators available for C, and they are either rudimentary or old and not currently supported or maintained.

5 Related Work

Studying the effects of software obfuscation in increasing the changes for ROP exploits is new and no previous work has been done in the areas involving both ROP exploitation and code obfuscation.

Pappas et. el, has proposed various ROP defenses: (1) kBouncer [16], a light weight tool to prevent certain ROP attacks using hardware features and without requiring any modifications of the binary code; (2) [15], showed, selective type of gadgets could be prevented using narrow code transformations of intended gadget instruction sequences. Another hardware based approach, ROPecker [8], prevents control flow based ROP attacks by examining the last branch taken registers, found in commodity processors. Abadi et. al., [6] used control-flow integrity (CFI) to prevent stack based ROP attacks. Carilini et. al., [7] discuss three new ROP attacks that breaks existing CFI based defense mechanisms such as kBouncer and ROPecker. Schuster et. al., analyze different defense mechanisms in [18] and show that with a little extra effort, it is possible to break ROPecker, kBouncer and ROPGuard.

In [13], the return-oriented programming is used for program steganography. This is a form of software obfuscation using return-oriented programming.

6 Conclusion

Software developers concerned about reverse-engineering attacks and piracy commonly deploy software obfuscation as a defense. The users of software however are more concerned about vulnerabilities in the software that may compromise their system. We show that there is a possibility of conflict between these two security goals if software obfuscation is used. We have shown that software obfuscation significantly increases (1.5x to 3x) the number of gadgets in a binary. We have also shown that for logic and control flow related gadgets, the increase is much higher (up to 6x). For certain, especially smaller, binaries which have very small number of logic and control gadgets this increase can potentially make ROP attacks feasible, or at best, easier.

References

[1] GNU Coreutils. `https://www.gnu.org/software/coreutils/`.

[2] Obfuscator LLVM. `https://github.com/obfuscator-llvm/obfuscator/wiki`.

[3] OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`.

[4] ROPgadget. `http://shell-storm.org/project/ROPgadget/`.

[5] The Tigress C Diversifier/Obfuscator. `http://tigress.cs.arizona.edu/index.html`.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[7] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[8] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[9] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *Software Engineering, IEEE Transactions on*, 28(8):735–746, 2002.

[10] L. Davi, A.-R. Sadeghi, and M. Winandy. Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM, 2011.

[11] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[12] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005.

[13] K. Lu, S. Xiong, and D. Gao. Ropsteg: program steganography with return oriented programming. In *Proceedings of the 4th ACM conference on Data and application security and privacy*, pages 265–272. ACM, 2014.

[14] J. Nagra and C. Collberg. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[15] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615. IEEE, 2012.

[16] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *USENIX Security*, pages 447–462, 2013.

[17] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[18] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses*, pages 88–108. Springer, 2014.

[19] 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*, pages 298–307. ACM, 2004.

[20] R. Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.