HoneyGadget: A Deception Based ROP Detection Scheme

Xin Huang, Fei Yan(✉), Liqiang Zhang, and Kai Wang

Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University, Wuhan 430072, China
yanfei@whu.edu.cn

Abstract. Return-Oriented Programming (ROP) is a robust attack which has been proven to be Turing-complete. ROP reuses code segments named gadget in vulnerable applications and modifies control flow to achieve malicious attacks. Existing defense techniques for code reuse attacks attempt to restrict the policy of control flow transfer (e.g. CFI) or make locating gadgets a hard work (e.g. ASLR). However, decades of the arm race proved the ability to detect up-to-date attacks remains the Achille's heel. In honeypot, a general pattern for operators is spreading honeytokens and hunting spammers by capturing their malicious behavior. In order to capture the attack pattern of code reuse attacks, we present a novel deception based ROP detection model named HoneyGadget. HoneyGadget inserts various types of honey gadgets as tokens to some specific points of binary files where normal control flow would not reach and record their places once the application is loaded. During the execution, HoneyGadget uses Last Branch Record (LBR) to trace execution records. On performing a sensitive function call, HoneyGadget compares LBR records with the maintained address list, and terminates the program immediately if some records match. Since these honey gadgets will not be executed by normal control flow, there must be a ROP attack. We have developed a fully functioning prototype of HoneyGadget. Our evaluation results show that HoneyGadget can (1) capture ROP attacks actively and (2) incurs an acceptable overhead of 7.61%.

Keywords: Return-Oriented Programming · Gadget insertion · Deception · Control flow · Last Branch Record

1 Introduction

Code injection attack was a tricky problem for software security practitioners before non-executable memory was introduced. With the widely deployment of DEP [2] and $W \oplus X$, attackers are forced to reuse existing code segments in binary. Over time, the state-of-art code reuse attacks have dramatically evolved from reusing sensitive system functions in related libraries of victim application (e.g. return-to-libc [33]) to chaining small code segments named gadgets into a

© Springer Nature Switzerland AG 2019
F. Liu et al. (Eds.): SciSec 2019, LNCS 11933, pp. 121–135, 2019.
https://doi.org/10.1007/978-3-030-34637-9_9

gadget chain (e.g. Return-Oriented Programming [5,30]). Triggered by a simple buffer overflow vulnerability, code reuse attack proved that it can perform arbitrary Turing-complete computation without injecting any malicious code [8]. In addition, there are several automated tools or methods available to help attackers to mount ROP attacks [28,29].

On the other hand, attempts to defense ROP attacks never stop. Existing defense techniques can be classified into two categories [24,26], which are randomization and control flow transfer checks respectively. A general purpose of Address Space Layout Randomization (ASLR) is to make data segments of the target application and the exact memory address of gadgets unpredictable. Control Flow Integrity (CFI) introduced by Abadi et al. [1] calls for validation checks for each control flow transfer. By constructing Control Flow Graph (CFG) statically and applying integrity checks at execution, CFI-based defense schemes restrict policies of control flow transfer.

However, existing defense methods are either one-time effort or detecting malicious behavior according to pre-defined policies, the ability to detect up-to-date attacks is outdated. Code reuse attacks via remote code execution is considered the most frequently used attack technique [4,31] in modern application scenarios. The remote adversary manipulates a code pointer to create memory disclosure and locates available gadgets for ROP attack. Equipped with weapons to exploit 0-day vulnerability, these advanced attacks are able to break existing defenses [4,6,7,15,31].

In order to capture these code reuse attacks, we propose HoneyGadget, a deception based defense scheme just like Honey-Patches [3]. HoneyGadget inserts honey gadgets as honeytokens to binary files of the target application and its related libraries, then we can detect attacks at runtime if inserted gadgets are executed. We have implemented a prototype of HoneyGadget on x86-based Linux platform. The experiment results show that the HoneyGadget incurs a modest overhead of 7.61% on average.

In summary, our main contributions of this paper include:

1. We propose HoneyGadget, a deception based ROP detection scheme, which provides a new method to capture ROP attacks.
2. We propose novel techniques combining constructing gadgets, inserting gadgets automatically and runtime ROP gadget chain detection method to achieve a ROP detection scheme.
3. We have implemented a prototype of HoneyGadget, and our evaluation shows that HoneyGadget achieves high accuracy with low overhead, proving our scheme practical.

The rest of this paper is organized as follows. We begin in Sect. 2 by introducing background knowledge on existing ROP attack methods and relative defenses. In Sect. 3, we detail our threat model and assumptions. The basic idea of HoneyGadget and the concrete implementation are illustrated in Sects. 4 and 5 respectively. We evaluate our system in Sect. 6. Related works are given in Sect. 7, and conclude in Sect. 8.

2 Background

2.1 Return-Oriented Programming

Return-Oriented Programming (ROP) is a typical code reuse attack. The main idea of ROP is chaining gadgets in the binary files of the target application as attack payload. Gadgets chosen for gadget chain are usually short with no more than 6 instructions [10] to avoid unplanned adjustment to pointers or registers. Each gadget in the attack payload is responsible for performing one or several steps of computation, such as loading argument from a specific register or performing arithmetic operations [8]. Triggered by an inconspicuous vulnerability, stack buffer overflow for example, control flow of the target application is hijacked. Together with the deployment of ROP defense schemes in modern system, ROP based attack techniques update correspondingly [6,7,15]. These attack techniques utilize flaws in control flow transfer policies, and bring defense schemes false negatives. Flexible and powerful, these features make ROP a state-of-art attack technique.

In modern application situations, except from software on the local host, applications and services provided by remote servers become a growing trend [3,13,27]. Correspondingly, attacks on those remote hosts based on remote code execution and code reuse techniques appear [4]. Based on the feature that servers do not rerandomize the address space layout after a crash under particular circumstances, BROP rewrites every single byte of stack canary after several attempts, and this corrupts stack integrity protection. The adversary then invokes write to dump more available gadgets in process memory. BROP enriches the arsenal of remote attackers and expand the attack surface of code reuse attacks.

2.2 Last Branch Record

Last Branch Record (LBR) provides a way to trace the execution control flow of a program, as it can log the branch information executed in a looped buffer at real-time. CPU can record the execution pace parallel at execution, and it incurs no slowdown. The length of the looped buffer is limited. For an Intel Haswell CPU, the length is set to 16, indicating that LBR can record the past 16 instruction branches executed. For an Intel Skylake CPU, LBR can record the last 32 executed instruction. While the looped buffer of LBR is filled, the newly recorded branches overwrite the old ones [16]. The functionality of LBR is enabled/disabled by certain model-specific registers (MSRs). The access to MSRs requires kernel privilege, which makes the status of LBR transparent to programs running in user space.

3 Threat Model and Assumptions

HoneyGadget aims to capture attack patterns of ROP attacks from both local-host and remote attackers. To ensure that our scheme is practical, we define our

threat model based on strong yet realistic attack assumption. With attack models in previous literature [4,6,7,15] and application scenarios of HoneyGadget, we generate the threat model as follows.

We assume the target application has at least buffer overflow vulnerability and the adversary has ready knowledge to exploit the vulnerability. The adversary is allowed to exploit the vulnerability repeatedly and can use automatic gadget generating tools to locate available gadgets and construct attack payload.

For remote side, we assume servers restart their worker processes after a crash and do not change their address space layout. Currently, servers such as Nginx and Apache are compatible with this feature. We further assume that the adversary is allowed to overwrite a variable length of bytes including a return instruction pointer [4]. These assumptions mean that the adversary can mount BROP attack successfully.

We assume the operating system enables standard defense mechanisms such as $W \oplus X$ and ASLR by default. However, as HoneyGadget focuses on capturing the malicious behavior of adversaries, methods aim to stop unintended control flow transfer such as CFI are disabled.

4 HoneyGadget

In this section, we describe the architecture of HoneyGadget. We first introduce the overview of our scheme, then we give out the detail of each component of HoneyGadget.

4.1 Overview

HoneyGadget owns two main components: *static processing module* and *runtime checking module* (see Fig. 1). The static processing module is responsible for (1) source code iteration and locating places to insert honey gadget as honeytokens; (2) generating gadgets that meet the requirement of potential code reuse attacks and (3) gadget insertion. After processed by the static processing module, the input file together with secured libraries are then taken over by runtime checking module. The runtime checking module of HoneyGadget (1) maintains address list of inserted gadgets and a pre-defined sensitive function list, and (2) performs runtime monitoring of execution. At last the output file is provided to local users and remote users. The output file has no interference on normal operations. However, those inserted honey gadgets are tempting but dangerous traps for attackers.

4.2 Static Processing Module

As we mentioned, the key idea of HoneyGadget is deception. Based on the observation of attack principle of code reuse attacks, we draw a conclusion that those attacks assemble gadgets into attack payload and hijack the control flow of victim

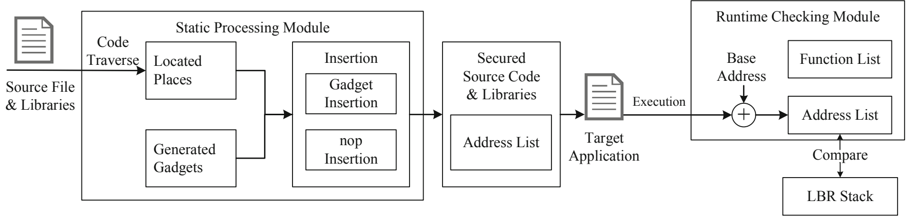


Fig. 1. Overview of HoneyGadget.

program no matter attack trick transforms. Thus, we can insert honey gadgets that meet the requirement of code reuse attacks as honeytokens. In order to avoid potential altering of execution flow caused by our gadgets, HoneyGadget inserts them to places where benign control flow would never reach.

In general, we summarize these places into two categories, which are opaque instructions [23] and function outlet.

Type I: Opaque Instruction: Opaque instructions have been used in software protection extensively. By setting predicates according to the value of invariant, context or execution result, opaque instruction is designed to clutter the control flow graph and it can redirect execution flow to a certain path.

Type II: Function Outlet: Code spaces right after function outlets is another case. The outlet of a function can be identified with the *ret* instruction and normal execution flow would never reach code segments right after *ret* instruction. However, inserting honey gadgets right after *ret* will grant the function with multiple outlets. Automatic gadget generating tool such as ROPEME [21] and ROPgadget [28] will regard the second function outlet as a fake one and discard it. In order to separate honey gadgets from existing function outlets, the static processing module selects instruction *nop* to complete this task. Those inserted *nop* sequences form interspaces between original outlet and the inserted gadget, and it confuses the automatic gadget generating tool.

Due to the poor alignment on x86 platform, unintended gadgets enrich attackers' options on their way to construct gadget chains. In order to eliminate potential unintended gadgets, HoneyGadget randomizes source code layout by randomly inserting *nop* instructions (0x90) before each assembly instruction. Shown in Fig. 2, by inserting a *nop* sequence between instruction "*mov [ecx], edx*" and "*add ebx, ebx*", the unintended gadget disappears. We will introduce the detailed implementation of inserting *nop* instructions and gadgets in Sect. 5.

For each honey gadget inserted, the static processing module records the offset to the start of the source code file in a formulation of address list. The address list is then maintained by runtime checking module during executions. Finally, the static processing module gives out a sensitive function list. The list contains function calls that can elevate privilege or perform arbitrary execution such as *execve()* and *setreuid()*.

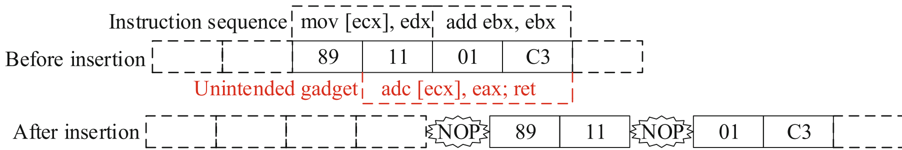


Fig. 2. The layout of instruction sequence after *nop* insertion.

4.3 Runtime Checking Module

Gadgets inserted are independent from existing code segments in source code, and there is no legal control flow transferred to them. Thus, it is most likely triggered by malicious attackers once the inserted gadgets are executed. Runtime checking module is designed to check whether there are inserted gadgets in execution branches of CPU. When loading application, ASLR randomizes the space layout of the application. Thus, in order to have an accurate record of honey gadgets inserted, the runtime checking module updates the saved address list. This module adds the base address of code segments with offsets of each honey gadget when the application is loaded. This maintenance procedure is done in kernel space, which is also transparent to user level applications.

Based on the observation that malicious executing code will eventually need to perform system calls to achieve something meaningful, the static processing module pre-defines a sensitive function call list and saves it in the kernel module together with the address list. While the target application is about to perform a sensitive function call, the runtime checking module pauses the execution of the target application and reads from the looped buffer of LBR. Then the runtime checking module compares the recorded instruction addresses with maintained address list. If one or more record matches, HoneyGadget confirms a ROP attack.

5 Implementation

In this section, we detail the implementation of our HoneyGadget, and give algorithms on gadget insertion and *nop* insertion.

5.1 Honey Gadget Insertion

Since HoneyGadget is a deception based defense scheme trying to confuse the ROP attacker by inserting honey gadgets. It turns out that the place where the gadgets are inserted, the number of inserted honey gadgets and types of those gadgets are the main factors that affects the effectiveness of HoneyGadget.

Places of Honey Gadget. The place to insert gadgets should be carefully arranged. Inserting gadgets inside normal instruction sequences may conflict with benign execution. For example, the gadget which modifies register *eax* may change the return address of benign execution flow. Consequently, the gadgets

should be placed to unreachable execution paths. However, the diversification of unreachable execution path should be guaranteed to avoid those honey gadgets from identification. We generate those places in two types: opaque instructions and code spaces right after function outlet.

Function Outlet: As mentioned in Sect. 4, inserting gadgets directly after ret instruction will grant a function with multiple outlets. In this case, automatic gadget generating tool will recognize the fake gadget discard it. In HoneyGadget, we insert *nop* instructions to space out the two outlets as a disguise. After analyzing several frequently used dynamic libraries including *glibc* and *ld*, we noticed that there are several *nop* instructions between basic blocks. The number of *nop* instruction is between 5 to 40. Thus, we disguise those inserted gadgets as normal code segments by inserting 5 to 40 *nop* instructions after *ret*.

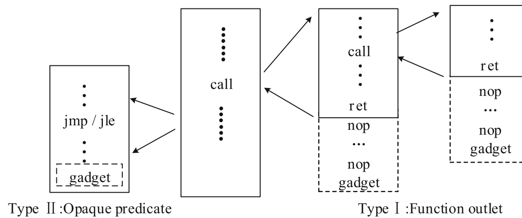


Fig. 3. Layout of code segment after inserting honey gadget.

Opaque Instruction: For opaque instructions, there exists three different types, which are invariant opaque predicates, contextual opaque predicates and dynamic opaque predicates [23]. In HoneyGadget, we focus on locating invariant opaque predicates in the source code. Due to its easy deployment, it is the most frequently leveraged opaque predicate [23]. HoneyGadget uses KLEE [32] to perform symbolic execution. KLEE is built on top of LLVM compiler infrastructure with a symbolic virtual machine engine. During this procedure, KLEE engine is responsible for locating the unreachable path and iterate to the end of this path. Following the end of the path, a gadget is inserted. The layout of code segment after inserting gadgets and *nop* instructions is shown in Fig. 3.

Insertion Algorithm. The honey gadget insertion algorithm is given in Algorithm 1. HoneyGadget randomly inserts *nop* instructions and gadgets after functions in source code at the probability of *pGadget*. For each insertion place, static processing module generates a random number *pRand*. If requirements are met, static processing module first inserts several *nop* instructions, then it randomly chooses a set of operation instructions such as *call*, *mov* or *sub* and an ending instruction to construct a gadget. To be noticed, HoneyGadget is able to generate all types of gadgets. This makes those honey gadgets inserted applicable for constructing a gadget chain. The length of generated honey gadget is no more than 6 instructions.

Algorithm 1. Honey gadget insertion

Input:
(1) The list of functions and opaque instructions, FList;
(2) The probability of insertion, pGadget;
(3) List of candidate operation instruction, operationTypeTable;
Output:
The list with deception gadgets inserted, FList.

```
1: numOperationTypes ← operationTypeTable
2: for F ∈ FList do
3:    pRand = random (0,1)
4:    if pRand ¡ pGadget then
5:       i = the ret instruction of F
6:       numNOP = random (5,40)
7:       insertAfter (i, nop, numNOP)
8:       i ← i.next (numNOP)
9:       nOpt = random (1,5)
10:      for index from 1 to nOpt do
11:         optIndex = random (0, numOperationTypes)
12:         insertAfter (i, operationTypeTable [optIndex])
13:         i ← i.next
14:      end for
15:      insertAfter (i, endRet)
16:   end if
17: end for
18: return FList
```

5.2 Insert nop

As presented in Sect. 4, HoneyGadget randomizes code layout by randomly inserting *nop* before each instruction, this procedure can eliminate potential unintended gadgets.

Similar with gadget insertion procedure, during *nop* insertion procedure, static processing module traverses each instruction from the first line in source code. For each instruction traversed, the module generates a random number *pInsert*. If *pInsert* is less than *pNop* defined previously, static processing module inserts a *nop* ahead of the instruction.

5.3 Trigger Detection

Runtime detection module of HoneyGadget leverages LBR to monitor execution states of instruction branches. Runtime detection module reads LBR buffer by using privilege instruction *rdmsr* and *wrmsr*. For an Intel Skylake CPU, the buffer of LBR can record last 32 executed instructions.

HoneyGadget pre-defines a sensitive function list containing function calls that can elevate privilege or perform arbitrary execution such as *execve*() and

setreuid() during static processing procedure. It will trigger runtime detection mechanism if one of the sensitive functions is called. While the detection mechanism is invoked, HoneyGadget pauses the execution of the target application. Then runtime checking module sends the privilege instruction *rdmsr* to kernel to read LBR buffer. After reading the 32 recorded instructions, the module leverages binary search algorithm to search if there exist one or more recorded instructions match with items in the address list. Since only malicious execution flow can reach inserted gadgets, those addresses in the address list shall never appear in LBR record during normal execution.

6 Evaluation

In this paper, we evaluate the space cost of inserting *nop* instructions and gadgets, effectiveness and performance of HoneyGadget. We implement HoneyGadget on Ubuntu 12.04 with 4 GB available memory. The machine equips an Intel Skylake i5-6500 CPU. And the deployed LLVM and Clang version are both 3.5.2.

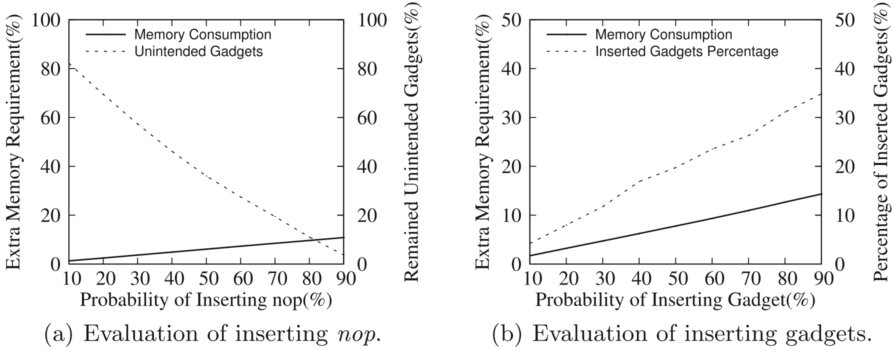
6.1 Effectiveness

In order to evaluate the effectiveness of our scheme, we verify HoneyGadget with real ROP attacks under two real world vulnerabilities. During these tests, *pNop* and *pGadget* are both set to 50%. Results of these tests indicate that HoneyGadget can prevent ROP attacks effectively.

Proof of Concept. In the first test, we test HoneyGadget on a small program containing a stack buffer overflow vulnerability. By inputting long parameters, the vulnerability is triggered and can be then utilized to launch a ROP attack. We use the automatic ROP gadget generating tool ROPGadget [28] to search available gadgets and randomly choose them to construct a ROP gadget chain. We repeat this test 50 times and report the final results. Among the 50 repeated tests, 49 of them used at least one of the inserted gadgets to construct the ROP gadget chain. HoneyGadget captured all the gadget chains containing inserted gadget with no false positive.

No-IP DUC. We also choose No-IP Dynamic Update Client (DUC) version 2.1.9 to conduct the test. The application fails to perform a boundary check while invoke vulnerable function *strcpy*(). The exploit database Exploit-db gives a ROP gadget chain example. We substitute gadgets in the gadget chain with gadgets generated by automatic gadget generating tool. Similar to the first test, we generate 50 gadget chains as ROP payload using different gadgets and 48 out of them contains at least one inserted gadget.

Nginx Web Server. HoneyGadget performs a deception based defense on remote code execution. Nginx web server is one of the most popular web servers in real world application situations. However, the weak security enforcement makes it vulnerable to a couple of attacks [4,14]. We exploit a simple stack vulnerability on Nginx 1.4.0 (64-bit) to launch a BROP attack. We apply HoneyGadget on



(a) Evaluation of inserting *nop*. (b) Evaluation of inserting gadgets.

Fig. 4. Space cost and effectiveness evaluation.

Nginx server, and the scheme inserted honey gadgets that meets the requirement of BROP attack automatically. We repeat BROP attack attempt 50 times, 46 out of them leveraged at least one inserted gadgets during stage 2 or 3 in attack payload. As expected, our HoneyGadget can detect those attacks with no false positive.

6.2 Memory Cost Evaluation

On loading the application, those inserted gadgets and *nop* instructions are loaded into the memory together with the application. Consequently, memory requirement of the target application inevitably increases. In our experiment, we evaluate the extra memory requirement of *nop* insertion and gadget insertion respectively. The insertion procedure increases the program binary size. We set *pNop* and *pGadget* 50% as benchmark, the average increase in binary size is 8.41%. Increasement on binary file size has a positive relationship with insertion probability.

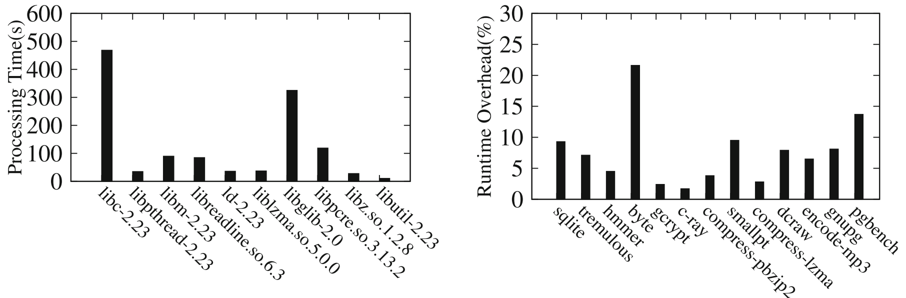
Space Cost and Effectiveness Evaluation of Inserting nop. We use HoneyGadget to process different applications and evaluate the space cost and effectiveness of *nop* insertion. In this test, we set *pGadget* 50%. Inserting *nop* instructions into source code of the target application inevitably increases its size, and the extra memory requirement has a linear positive relationship with nop insertion probability. Figure 4(a) shows that it takes 1.31% extra memory space while *pNop* is set to 0.1, and 10.84% extra memory cost while *pNop* is set to 0.9. On the other hand, along with the increase of *pNop*, the possibility of corrupting an unintended gadget raises. The dashed line in Fig. 4(a) gives the remained unintended gadgets percentage. The percentage of remained unintended gadgets drops from 82.13% to 3.72%.

Space Cost and Effectiveness Evaluation of Inserting Gadgets. Similar to the evaluation on *nop* insertion, in this evaluation procedure, we leverage same applications to perform the evaluation, and *pNop* is set to 50% as benchmark. It then takes about 1.71% extra memory when *pGagdet* is set 0.1, and the memory consumption raises to 14.33% while *pGadget* is 0.9. Together with the increment of *pGadget*, the scale of inserted gadget increases. The results of the experiment show that with *pGadget* of 0.1, only 4.61% of gadgets are inserted. The ratio increases to 19.74% for *pGadget* of 0.5, and to 34.82% for *pGagdet* of 0.9. Figure 4(b) shows the results.

6.3 Performance Overhead

To evaluate the overhead brought by HoneyGadget, we divide the evaluation into 2 phases. Corresponding to the architecture of HoneyGadget, the first phase is static processing, and another one is runtime checking.

We set *pGadget* and *pNop* to 50% and evaluate performance overhead of static processing phase by adding *-time-passes* argument. During the traverse procedure, the module identifies all instructions, basic blocks and functions. Thus, the larger the library size is, the longer time for static processing is needed. Time for processing frequently-used libraries are shown in Fig. 5(a). The experiment results meet this idea. As the results show, except from some huge libraries, it takes about 30 s to process a dynamic library. For example, it takes 35.96 s to process ld-2.23.so and 36.86 s to process liblzma.so. As for libraries with a huge quantity of basic blocks and functions, processing these libraries requires much time. Taking the library libc-2.23.so for an example, the time consumption increases to 468 s. Although it does take some time to do the static processing work, fortunately, operations in static processing phase is mostly a one-time effort, for libraries can be shared by different applications.



(a) Time consumption for processing different libraries.

(b) Runtime overhead of HoneyGadget.

Fig. 5. Performance overhead of HoneyGadget.

For runtime checking phase, we evaluate the performance overhead by running a benchmark test using *phoronix test suite* [20] with optimization level *-O2*. As we have introduced, those inserted gadgets are unreachable for benign control flow, and they introduce no runtime overhead during execution. The main performance overhead incured by honeygadget is to compare LBR records when handling sensitive system calls. On the other hand, the *nop* instructions we inserted will be executed. With more *nop* instructions inserted, the normalized runtime overhead increases accordingly. In our experiment, we set *pNop* 50%, the evaluation results are shown in Fig. 5(b). HoneyGadget introduces an average overhead of 7.61% which is less than Readactor++ (8.4%) and other fine-grained CFI solutions.

7 Related Work

Address Space Layout Randomization (ASLR) is a representative mechanism to defend ROP attacks. By re-allocating the space layout, ASLR changes the base address of the application and its related libraries. However, a single memory leakage vulnerability is enough to de-randomize the whole memory space.

Enhancement on ASLR is mainly on re-randomization and applying fine granularity. For example, ASLP [19] randomizes the target application at the function level, Remix [9] randomizes the address space at basic block level, and ILR [17] realizes randomization at instruction level. Bigelow et al. promoted a timely randomization scheme to re-randomizes address layout during execution [21]. Although these fine-grained ASLR significantly increase the difficulty for attackers to locate useful information in memory, it also brings extra time consumption and memory allocation.

Inserting some instructions in the program that do not affect the execution of the program can also increase the difficulty for the attacker to obtain internal information of the program. kGuard [18] uses a *nop* sled to change address locations, but they only do this to protect and diversify the kernel. HoneyGadget randomly inserts *nop* instructions and gadgets to source code of the target application and its related libraries. As mentioned in Sect. 5, the diversity of gadget types and inserted places makes attackers hard to distinguish inserted gadgets from original ones. Moreover, the maintained address list is in kernel space, this makes the address list transparent to adversaries and immune to information leakages in application layer.

Though the strict control flow transfer check mechanism is able to mitigate potential control flow hijacking, CFI poses an unacceptable overhead of more than 20%. In order to make CFI practical, a few coarse-grained mechanisms based on CFI are proposed. Coarse-grained CFI mechanisms relax the limitation of legal indirect control flow transfers, and simplify the checking method. Compared with fine-grained CFI, coarse-grained CFI mechanisms such as CCFIR [34] and binCFI [35] loose the indirect control flow checking policy and reduce overhead to an acceptable level. However, the loose checking policy brings potential vulnerabilities.

Another way to reduce overhead of checking the validity of control flow transfer is based on hardware. Liu et al. introduced a CFI enforcement using Intel Processor Trace [22]. Compared to using IPT to trace the execution path, CPU is able to read LBR registers parallel at execution. This feature makes LBR a more efficient way to log instruction branches of the application. Kbouncer [25] uses LBR to detect ROP attacks. ROPecker [10] also leverages LBR to optimize performance overhead. During offline processing procedure, ROPecker identifies potential gadgets and saves them in Instruction & Gadget database (IG). ROPecker reads LBR buffer and analysis executed gadgets in IG, then it indicates following instructions by simulating execution. If the number of gadgets reaches the limit, ROPecker warns user of ROP attack. HoneyGadget also uses LBR to record execution branch of the target application. However, different from these two approaches, the main idea of HoneyGadget is tempting adversaries to launch attacks by inserting gadgets to binary code. The behavior of the attacker is then captured and logged by host.

Booby trap [11] is a mechanism to actively detect and respond to attacks against a target application proposed by Crane et al. The main idea of booby traps is as follows: in a diversified application, code sequences (the actual booby traps) are added that trigger an active response, such as terminating the program or generating an alert. Readactor++ [12] inserts booby traps in both PLT and vtables to mitigate blind probing of table entries. HoneyGadget inserts *nop* instructions and honey gadgets to confuses adversary with traps. Compare to Readactor++, our HaneyGadget is more active and has a greater chance of getting attackers into the traps.

8 Conclusion

In this paper, we present a deception based ROP defense scheme named HoneyGadget. By inserting *nop* instructions and honey gadgets, our HoneyGadget confuses adversary with traps. HoneyGadget maintains an address list recording addresses of inserted gadgets in kernel space and defines a set of sensitive function calls. Once executing the sensitive function call, HoneyGadget pauses execution of the target application and reads LBR buffer to check if recorded instruction branches match with addresses in address list. If the record matches, HoneyGadget alarms a potential ROP attack. Our evaluation shows that HoneyGadget incurs an acceptable runtime overhead of about 7%. Compared to other ROP defense mechanisms, the key idea of HoneyGadget is deception, which is a brand-new method to detect code reuse attacks.

Acknowledgement. This work was supported in part by the National Natural Science Foundation of China under Grant No. 61272452 and the National Basic Research Program of China (973 Program) under Grant No. 2014CB340601.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, pp. 340–353. ACM (2005)
2. Andersen, S., Abella, V.: Data execution prevention. Changes to functionality in Microsoft Windows XP Service Pack 2, Part 3: memory protection technologies (2004)
3. Araujo, F., Hamlen, K.W., Biedermann, S., Katzenbeisser, S.: From patches to honey-patches: lightweight attacker misdirection, deception, and disinformation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 942–953. ACM (2014)
4. Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D.: Hacking blind. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 227–242. IEEE (2014)
5. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to RISC. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 27–38. ACM (2008)
6. Carlini, N., Barresi, A., Payer, M., Wagner, D., Gross, T.R.: Control-flow bending: on the effectiveness of control-flow integrity. In: USENIX Security Symposium, pp. 161–176 (2015)
7. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses. In: USENIX Security Symposium, pp. 385–399 (2014)
8. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 559–572. ACM (2010)
9. Chen, Y., Wang, Z., Whalley, D., Lu, L.: Remix: on-demand live randomization. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, pp. 50–61. ACM (2016)
10. Cheng, Y., Zhou, Z., Miao, Y., Ding, X., Deng, H., et al.: ROPecker: a generic and practical approach for defending against ROP attack (2014)
11. Crane, S., Larsen, P., Brunthaler, S., Franz, M.: Booby trapping software. In: Proceedings of the 2013 New Security Paradigms Workshop, pp. 95–106. ACM (2013)
12. Crane, S.J., et al.: It's a trap: table randomization and protection against function-reuse attacks. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 243–255. ACM (2015)
13. Durumeric, Z., Bailey, M., Halderman, J.A.: An internet-wide view of internet-wide scanning. In: USENIX Security Symposium, pp. 65–78 (2014)
14. Evans, I., et al.: Missing the point (ER): On the effectiveness of code pointer integrity. In: 2015 IEEE Symposium on Security and Privacy (SP), pp. 781–796. IEEE (2015)
15. Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 575–589. IEEE (2014)
16. Guide, P.: Intel® 64 and ia-32 architectures software developer's manual. Volume 3B: System programming Guide, Part 2 (2011)
17. Hiser, J., Nguyen-Tuong, A. Co, M., Hall, M., Davidson, J.W.: ILR: where'd my gadgets go? In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 571–585. IEEE (2012)

18. Kemerlis, V.P., Portokalidis, G., Keromytis, A.D.: kGuard: lightweight kernel protection against return-to-user attacks. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 2012), pp. 459–474 (2012)
19. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In: 22nd Annual Computer Security Applications Conference, ACSAC 2006, pp. 339–348. IEEE (2006)
20. Larabel, M., Tippett, M.: Phoronix test suite. Phoronix Media (2011). http://www.phoronix-test-suite.com/. Accessed June 2018
21. Le, L.: Payload already inside: datafire-use for ROP exploits. Black Hat USA (2010)
22. Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., Guan, H.: Transparent and efficient CFI enforcement with Intel processor trace. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 529–540. IEEE (2017)
23. Ming, J., Xu, D., Wang, L., Wu, D.: Loop: logic-oriented opaque predicate detection in obfuscated binary code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 757–768. ACM (2015)
24. Pappas, V.: Defending against return-oriented programming. Columbia University (2015)
25. Pappas, V.: kBouncer: efficient and transparent ROP mitigation, pp. 1–2, 1 April 2012 (2012)
26. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: USENIX Security Symposium, pp. 447–462 (2013)
27. Riden, J., McGeehan, R., Engert, B., Mueter, M.: Know your enemy: web application threats, using honeypots to learn about http-based attacks (2007)
28. Salwan, J.: ROPgadget-Gadgets finder and auto-roper (2011)
29. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: USENIX Security Symposium, pp. 25–41 (2011)
30. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the X86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552–561. ACM (2007)
31. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 574–588. IEEE (2013)
32. Team, K.: KLEE LLVM execution engine. http://klee.github.io/
33. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 121–141. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23644-0_7
34. Zhang, C., et al.: Practical control flow integrity and randomization for binary executables. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 559–573. IEEE (2013)
35. Zhang, M., Sekar, R.: Control flow integrity for COTS binaries. In: USENIX Security Symposium, pp. 337–352 (2013)