DiffGuard: Obscuring Sensitive Information in Canary Based Protections

Jun Zhu(✉), Weiping Zhou, Zhilong Wang, Dongliang Mu, and Bing Mao

State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China
junzhu0406@gmail.com, zhouweipingcs@163.com, njuwangzhilong@163.com, mudongliangabcd@163.com, maobing@nju.edu.cn

Abstract. Memory Corruption attacks have monopolized the headlines in the security research community for the past two decades. NX/XD, ASLR, and canary-based protections have been introduced to defend effectively against memory corruption attacks. Most of these techniques rely on keeping secret in some key information needed by the attackers to build the exploit. Unfortunately, due to the inherent limitations of these defenses, it is relatively difficult to restrain trained attackers to find those secrets and create effective exploits. Through an information disclosure vulnerability, attackers could leak stack data of the runtime process and scan out canary word without crashing the program. We present DiffGuard, a modification of the canary based protections which eliminates stack sweep attacks against the canary and proposes a more robust countermeasures against the byte-by-byte discovery of stack canaries in forking programs. We have implemented a compiler-based DiffGuard which consists of a plugin for the GCC and a PIC dynamic shared library that gets linked with the running application via LD PRELOAD. DiffGuard incurs an average runtime overhead of 3.2%, meanwhile, ensures application correctness and seamless integration with third-party software.

Keywords: Information leak · Brute-force attacks
Canary-based protection · Canary re-randomization

1 Introduction

Buffer overflows, sensitive data exposure and related memory corruption vulnerabilities constitute an important class of security vulnerabilities. According to the CNNVD Situation Report in 2016 [1], there exists a notable increase in vulnerability number, from 5128 in 2011 to 8336 in 2016. Buffer overflows remain the most frequently encountered [2], which brings a huge threat to network and information security. Over the last years, several techniques have been developed to prevent adversaries from abusing them. Stack canaries [3–5], Address Space Layout Randomization [6] and non-executable stack (NX/XD) [7] are widely deployed due to the low overhead,simplicity and effectiveness. However, none of

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 738–751, 2018.
https://doi.org/10.1007/978-3-319-78813-5_39

these techniques has fully eliminated stack smashing attacks and several attack vectors are still effective under all these protections [8–11].

Stack buffer overflows are often used as a stepping stone in modern, multi-stage exploits like Return-Oriented Programming (ROP) [15]. For instance, Blind ROP (BROP) [13] attack requires only a stack-based memory corruption vulnerability and a service that restarts after a crash to automatically construct a ROP payload. Security researchers believe that only the forked networking servers are prone to brute force attacks, based on the fact that in forking application, all the children processes inherit/share the same memory layout from the parent process. The attacker can try in bounded time all the possible values of canary (for SSP) and memory layouts (for ASLR) until the correct ones are found. There exists a dangerous form of SSP vulnerability, called byte-for-byte, which allows the attacker to try each byte of the canary independently and to find the value of the canary with a little number of attempts. There exists some techniques effectively armoring protections against brute-force attacks on forking program, but they could not guarantees the correctness of child process [14]. The different function frames of the process share the same canary word stored in TLS, which greatly weaken the security of process data. Through CVE-2012-3569 VMware OVF Tool format string vulnerability [12], we successfully leaked the runtime stack data and scanned out the canary without crashing the program.

The severity and plethora of these exploits underline the redesign of canary-based protections. To address the aforementioned issue, we present a modification of the SSP technique, called DiffGuard (Different function frames with different canaries), which consist of assigning different canaries for different function frames and setting a number of new canary words for each child process when the fork() system call is invoked. Specifically, through a lightweight, per-frame randomizing mechanism, our design smashes the consistency issue of traditional canary based protections and enables the runtime update of the canary values in all protected function frames of the running thread, so that newly-forked processes get a number of fresh canaries, different from the canaries of their parent process. Contrary to previous work [14,15], our approach makes the canary of different frames independent of each other in both non-forking and forking programs and guarantees correctness while preventing brute force attacks against stack canary protection on forking programs. DiffGuard provides protection based on source code, which is a compiler-level version of tool, implemented as a GCC plugin, incurs just 3.2% runtime overhead over native execution, and is fully compatible with third-party libraries that are protected with the default canary mechanism.

In summary, the main contributions of this work are the following:

1. We present DiffGuard, a robust solution for obscuring sensitive information(canary word) in Canary based Protections.
2. The SSP byte-by-byte attack in forking applications is no longer applicable to the DiffGuard.

3. We have evaluated the effectiveness of the recently proposed solution [14, 15] to the problem of identical canary stored in each function frames, and demonstrate how DyffGuard overcomes its design limitations.
4. We have implemented a compiler-level DiffGuard, demonstrating the practicality of our approach, which incurs a runtime overhead of 3.2% and shows that it can be easily adopted by popular compiler toolchains to further address security issues arising from the process creation mechanism of modern OSes.

The rest of the paper is organized as follows. We provide a background on the existing defenses and review their weaknesses with respect to canary based protections in Sect. 2. We detail the design of DiffGuard in Sect. 3. We describe the implementation details in Sect. 4. We evaluate our system in Sect. 5, and we cover some related work in Sect. 6, and conclude in Sect. 7.

2 Background

In this section, we first introduce simply the stack smashing attack and canary-based protection. Then we briefly describe the existing work and propose a stack scan algorithm based on the limitations of existing works bypassing canary based protections.

2.1 Canary-based Protection and Brute-force Attacks

The general principle of stack smashing attack is to change the control flow to execute attacker-supplied code. Stack smashing relies on the fact that most C compilers store the saved return address on the same stack used for local variables. The common form of buffer overflow exploitation is to attack buffers allocated on the stack. A well-accepted countermeasure against stack smashing attacks is the Canary-based Stacking Smashing Protection. The basic idea is to place a canary right after the return address in stack frame to detect buffer overflows.

Processes created with fork() are a duplicate of the calling process. Both, father and child have the same canary value. On a forked server, where the service is attended by children of the server process, an attacker can build **brute force attacks** by guessing the value of the canary as many times as needed.

Bit-by-bit Attack: The frame-canary word is overwritten on each trial. If the guessed word is not correct then the child process detects the error and aborts. As consequence, the attacker does not receive a reply, which is interpreted as an incorrect guess. The guessed value is discarded, and attacker proceeds with another value until all the possible values are guessed.

Byte-by-byte Attack: The basic idea in leaking canaries with byte-by-byte attack is to overflow a single byte, overwriting a single byte of the canary with value x. If x was correct, the server does not crash. The algorithm is repeated for all possible 256 byte values until it is found (128 tries on average). The attack continues for the next byte until all 8 canary bytes (on 64-bit) are leaked.

Table 1 shows the complexity of using bit-by-bit versus byte-by-byte attacks. Most canary implementations set to zero one of the canary bytes (the most significant in x86) for preventing the buffer overflow attacks when the overflow is performed by a string copy functions. For this reason, the number of bytes needed to guess is three (for 32-bit systems) or seven bytes (64-bits systems). Statistically, the bit-by-bit attack is described as a "**sampling without replacement**" and since all the values has the same probability ($\frac{1}{c}$) it is modelled by the uniform distribution with a support range of [1, c] and a mean of $\frac{c+1}{2}$ [14]. With the standard SSP, bit-by-bit attack needs at most 2^{24} trails to break the system (and 2^{23} in average) in 32-bit systems. On a byte-by-byte attack, the process of finding each byte is modelled as a uniform distribution whose mean is 256/2 and the support range is [1, 256], the attacker needs at most 768 trails to break the system (and 384 in average) in 32-bit systems. The average requests in 64-bit systems is calculated as above. With this figures, the standard canary technique provides a weak protection for this kind of bugs.

2.2 Previous Works

The basic idea of preventing brute force attacks focuses on re-randomizing the reference-canary of the child right after the **fork()**. RAF-SSP'renew canary at fork strategy consist in renew the value of the reference-canary of the child process right after it is created (forked). The new value is also a random value and every child process have a different reference-canary. However, this partial update will result in an abort if execution reaches the frames inherited from the parent process, as the canary cookies in these frames still hold their old values [15]. RAF-SSP assumes that a child process never reuses inherited frames legitimately. DynaGuard use per-thread bookkeeping mechanism to guarantee program correctness. At a high level, DynaGuard operates as follows: after a fork system call, and right before any instruction has executed in the child process, DynaGuard must update the canaries in both the TLS and all inherited stack frames in the child process.

2.3 Threats

Rather than a detailed explanation on how to bypass the SSP, we will present only the weaknesses of the existing canary based protections that enables the possibility of an attack. Basically, there are three ways to bypass the canary:

1. Overwriting the target data (return address, function pointer, etc.) without needing to overwrite the frame canary.
2. Overwrite the frame-canary with the correct value.
3. Disclosure runtime memory data.

With a view of situation 1, since GCC v4.6.3, local variables are reordered so that buffers are located first (higher addresses) and below them the function pointers and the saved registers. Based on this fact, directly overwriting the target data could not achieve the goal. We had discussed situation 2 in Sect. 2.1 and introduced existing works which prevent brute force attacks against canary based protections in Sect. 2.2. So we will focus on the memory data disclosure against the canary value.

The different function frame shared the same canary stored in TLS, which greatly weakened the randomness of canary. Through an information disclosure vulnerability, attackers could leak stack data of the runtime process and scan out canary word without crashing the program. We proposed a scanning algorithm to find out canary. The input is runtime-stack data which have been leaked by program vulnerability(format string, dangling pointer, etc.) and platform information. The output is the most possible canary words. The intuition here is that all the function frames of the runtime-stack stored the same canary words, this consistency allows us to find canary from the disclosure data. In this algorithm, we set the size of the sliding window to memory address width (e.g., 4 bytes for a 32-bit operating system). The scan of the runtime stack data starts from the top of the stack indicated by the value of stack pointer ESP plus an offset equal to the memory address width (e.g., $\mathrm{ESP} + 4$ for a 32-bit operating system). We add repeated words which frequency of occurrence is more than three times in the candidate tag. In order to find out canary from the candidate collection, We made the following three rules to determine:

1. Terminator value: most canary implementations set the last byte of canary to the terminator value. the value is composed of different string terminators (CR, LF, NULL and -1).
2. Randomization: canary is a random value generated by reading the device /dev/random. Such as 0xAAA0, 0xABA0, 0xAAB0(A and B represent hexadecimal numbers) could be directly removed.
3. Function prologue: The prologue of each function is fixed. In GCC version's canary based protections, canary is usually stored in the position of $\mathrm{EBP} + 8$. Through this relative offset, we can further screen out the possible canary.

Through the above rules, We can screen out the canary value. Considering XOR canaries implemented in Windows, the canary is generated by $canary_t \otimes EBP_f$, $canary_t$ is the original canary value stored in TLS and EBP_f is the base address of the function frame. Since the upper 2 or 3 bytes are fixed, our scanning algorithm is still working.

Table 1. Comparison of different canary based protections

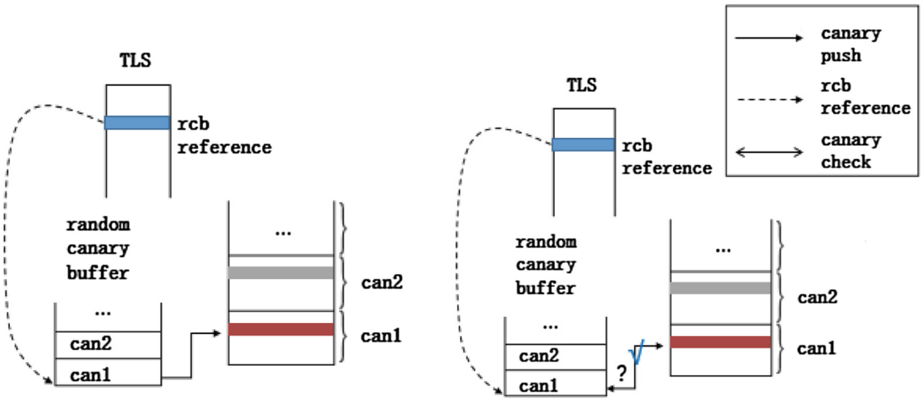
Protextion	Brute force attack	Correctness	Consistency
StackGuard	F	T	F
RAF-SSP	T	F	F
DynaGuard	T	T	F

As shown in Table 1, due to the same canary in both parent and child process, StackGuard [3] is specially prone to brute force attacks in forking applications.

RAF-SSP could prevent brute force attacks against SSP, but could not guarantee correctness of the program. Althrough RAF-SSP and DynaGuard ensure that parent process has a different canary value with the child process, but the different frames still store the same canary in a process. These three protections can not solve the problem of consistency. In the following sections, we discuss how DiffGuard solves the problems discussed above while preserving application correctness and preventing brute force attack against canary based defenses.

3 Design

At a high level, DiffGuard operates as follows: (1) for non-forking program, DiffGuard assigns a separate canary to each stack frame in the process. (2) for forking program, after a fork system call, and right before any instruction executed in the child process, DiffGuard must update canaries inherited from the parent process. Once the canaries have been updated, it can resume the execution of the child. This runtime update renders byte-by-byte brute-force attacks infeasible, since every function frame of forked process has a fresh canary.



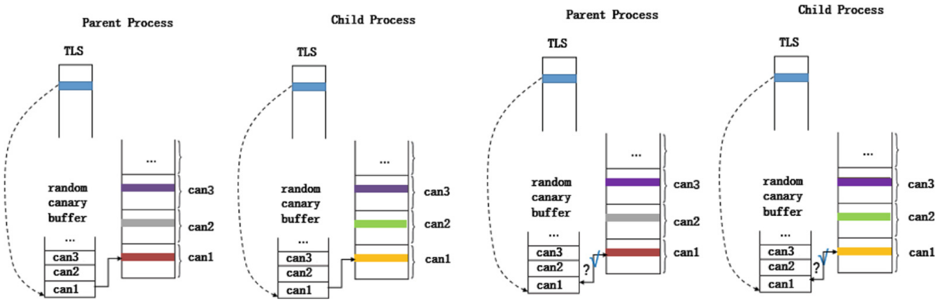
(a) the canaries stored in random canary buffer(RCB) are pushed on the stack.

(b) Epilogue check for function can1 succeeds.

Fig. 1. The design of DiffGuard allows for a complete independence of all canaries in the process.

To the best of our knowledge, current canary protections do not provide multiple different canaries for different function frames in the process. Therefore, DiffGuard's design should allow each running process to generate, access and modify all of its stack canaries at runtime. To achieve this goal, DiffGuard performs a per-thread runtime randomization of all the canaries that will be pushed in the stack during execution, using a lightweight buffer allocated dynamically upon each thread's creation (this buffer is stored in the heap). Figures 1 and 2 illustrate this scheme in more detail.

DiffGuard's random canary buffer (RCB, Fig. 1a) holds all the canaries of the runtime process. When a function is called, DiffGuard takes a canary word from the RCB and pushes it on the function frame. As the function execution is finished, DiffGuard detects the change of the canary word before the function returns (Fig. 1b). To ensure DiffGuard could prevent brute force attacks against canary based protections, we refresh the contents of the RCB. When a child process is forked, the RCB of the parent process is copied to the child process (Fig. 2a). Before execution starts in the child context, DiffGuard modifies all the canary values of the RCB excepted the canaries inherited stack frames in child process (can3). Likewise, whenever a canary-protected frame is pushed onto the stack, the canary is token from the RCB and, once a canary-protected function returns, the respective RCB index is diminished (Fig. 2b). The aforementioned design allows DiffGuard to successfully provide multiple different canaries for different function frames in the process and to modify the canary values for newly-created processes (child process). Specially, it allows for a seamless integration with third-party software and libraries that only support the existing stack protection mechanisms. In addition, the proposed architecture allows for the effective handling of stack unwinding, irrespectively of whether the latter occurs in the context of an exception, due to a signal, or setjmp/longjmp: as the canary saved in the function frame corresponds to the one in the RCB, We can determine the position of the canary in the RCB. Thus, DiffGuard can hook any stack unwinding operation and modify the RCB index accordingly. In this manner, application correctness is preserved. Apart from ensuring correctness, the proposed design has the added benefit of not breaking compatibility with legacy software or current canary protections. Compilers only need to add this bookkeeping mechanism on top of their current stack canary implementations, without altering the well-established conventions on the format of the canary check or a function's prologue and epilogue.



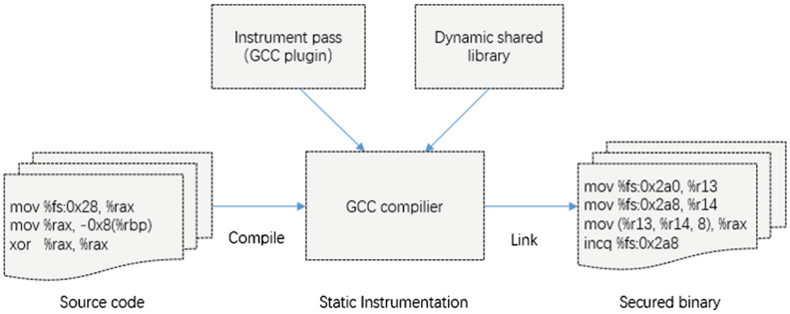
(a) The per-thread RCB is updated. (b) Epilogue check for function can1 succeeds.

Fig. 2. The design of DiffGuard modifies all the canary values of the RCB excepted the canaries inherited stack frames in child process.

4 Implementation

The compiler-based DiffGuard consists of a plugin for the GNU Compiler Collection (GCC) and a position independent (PIC) dynamic shared library that gets linked with the running application via LD_PRELOAD. Combined, they consist of more than 2500 lines of C++ code. Several requirements must be accomplished to implementing DiffGuard at the compiler level, while maintaining compatibility with third-party software at the same time:

1. DiffGuard must instrument all the canary push/pop events and perform its randomization on a per-thread basis;
2. DiffGuard must hook each fork system call and update the canaries in the child process' RCB as described in Sect. 3;
3. DiffGuard must intercept all calls related to stack unwinding and ensure that the RCB index gets updated accordingly.



```
mov %fs:0x28, %rax
mov %rax, -0x8(%rbp)
xor   %rax, %rax
```

```
mov %fs:0x2a0, %r13
mov %fs:0x2a8, %r14
mov (%r13, %r14, 8), %rax
incq %fs:0x2a8
```

Fig. 3. Overview of system architecture.

The first requirement is handled by DiffGuard's GCC plugin. All other requirements are handled by DiffGuard's dynamic shared library (runtime), which ensures the proper management of the RCB for every thread.

The overview of DiffGuard architecture is shown in Fig. 3. To generate a binary secured against information disclosure vulnerabilities and brute force attacks, developers should compile the source code of the target program with DiffGuard. Given the source code, DiffGuard first identifies instructions that push/pop canary events and then inserts a call to the routine (a static instrumentation in Sect. 4.1). At runtime, with the help of instrumented instructions, DiffGuard initializes a number of random canaries which are stored in RCB for the function frame being created. On every fork system call, DiffGuard updates the per-thread RCB (a runtime library in Sect. 4.2). Later in this section, we describe each component of DiffGuard (the static instrumentation and the runtime library), and explain how we maintain RCB.

4.1 Static Instrumentation

The static instrumentation of DiffGuard is performed at the GCC IR [17] level, registered as an RTL optimization pass and loaded by GCC right after the vartrack pass. The first reason for placing DiffGuard late in the RTL optimization pipeline is to ensure that most of the important optimizations have already been performed, and, as a result, DiffGuard's instrumentation is never added to irrelevant code. In addition, in this manner, we ensure that all injected instructions, which performs the necessary randomization, will remain at their proper locations and will not be optimized by later passes.

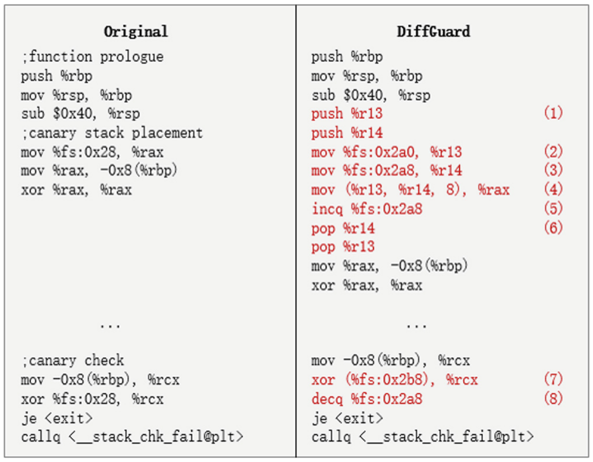
The DiffGuard GCC plugin must modify the canary setup and check inside each canary-protected frame, to prevent the DiffGuard-protected application from using the standard libc canaries. This is necessary to allow the modification of the canary at runtime without affecting any checks in libraries that are not complied with DiffGuard. The canary initialization that occurs during the creation of threads and processes is exactly the same in DiffGuard and in glibc, with the only difference being that the DiffGuard canaries are stored at RCB and the reference to the RCB is stored at a different location in the TLS area. Therefore, the entropy of canaries is not affected, but now the TLS holds two different types of canaries: the standard glibc canary and the DiffGuard canary. Upon a fork, all DiffGuard canaries excepted the canaries inherited stack frames in child process get updated without affecting any checks in modules or libraries that use the legacy glibc canaries.

DiffGuard stores the starting address of RCB, its total size, and its index, in the TLS. In x86-64, the reserved TLS offsets range from 0x2a0 to 0x2b8. In particular, %fs:0x2a0 holds the base address of RCB, %fs:0x2a8 keeps the current index in the RCB (i.e.,how many function frames are created), and finally, %fs:0x2b8 stores the reference to the DiffGuard canary which belongs to the function frame that is currently executing.

Figure 4 shows the canary push/pop instructions inserted by the DiffGuard GCC plugin. Right after the function prologue, before the canary gets pushed to the stack, the reference to the starting address of RCB must be read. Initially, DiffGuard retrieves the address of the RCB from the TLS (1) and the index of the next element to be written (3). Next, it reads canary from RCB (4) and increments the buffer index (5). Finally, the canary is fetched from the RCB and saved onto the stack. For this purpose, if no registers are free, DiffGuard needs to spill two registers for its push/pop canary events ((1),(6)). Likewise, the canary check in the function epilogue is modified to check against the DiffGuard canary instead of the glibc canary (7) and decrease the index in RCB (8).

4.2 Runtime Library

The runtime library of DiffGuard maintains the RCB setup and update, as well as the hooking of fork system calls and stack unwinding routines. The library (PIC module) implementing that runtime is loaded via the LD_PRELOAD mechanism into the address space of the runtime application.



```
         Original                              DiffGuard
;function prologue                  push %rbp
push %rbp                           mov %rsp, %rbp
mov %rsp, %rbp                      sub $0x40, %rsp
sub $0x40, %rsp                     push %r13                        (1)
;canary stack placement             push %r14
mov %fs:0x28, %rax                  mov %fs:0x2a0, %r13              (2)
mov %rax, -0x8(%rbp)                mov %fs:0x2a8, %r14              (3)
xor %rax, %rax                      mov (%r13, %r14, 8), %rax        (4)
                                    incq %fs:0x2a8                   (5)
                                    pop %r14                         (6)
                                    pop %r13
                                    mov %rax, -0x8(%rbp)
                                    xor %rax, %rax

          ...                                 ...

;canary check                       mov -0x8(%rbp), %rcx
mov -0x8(%rbp), %rcx                xor (%fs:0x2b8), %rcx            (7)
xor %fs:0x28, %rcx                  decq %fs:0x2a8                   (8)
je <exit>                           je <exit>
callq <__stack_chk_fail@plt>        callq <__stack_chk_fail@plt>
```

Fig. 4. Assembly excerpt for a binary compiled with -fstack-protector, with and without DiffGuard. The canary randomizing code added by the DiffGuard plugin is shown on the right (highlighted).

The RCB is allocated in the heap for each thread of the running program. In order to allocate the RCB before the main thread starts executing, we register-in the DiffGuard runtime-a constructor routine to be called before the main function of the application. This routine performs the RCB allocation, generates a number of random words by reading the device /dev/random and places them in the RCB. Finally, it sets the reference, size and index of RCB in the main thread' s TLS. For all other threads that get created, DiffGuard hooks the pthread_create call and sets the respective TLS entries prior to calling the start_routine of each thread. Finally, a routine to free the allocated RCB for each thread that finishes execution is registered via the pthread_cleanup_push(/pop) mechanism.

To ensure that the canaries in RCB of each thread are sufficient to use, DiffGuard marks the final page in the RCB as write-only and registers a signal handler for the SIGSEGV signal. Inside the signal handler, DiffGuard detects whether the fault is due to DiffGuard's instrumentation (i.e., when DiffGuard tries to read a canary out the boundary of the RCB) and allocates additional memory for the RCB if necessary.

As there may be multiple running threads, and the exception handler may execute in the context of a different thread than the one that generated the SIGSEGV, DiffGuard maintains a hashmap of all the running threads and their TLS entries. Inside the signal handler, DiffGuard iterates through all the threads in the hashmap and examines whether the memory location that caused the fault falls within an allocated RCB.

Lastly, in order to ensure that the RCB' index will correspond to active frame, DiffGuard checks for any stack unwinding and revises the index of RCB. This is based on the simple observation that, as the canary saved in the function frame corresponds to the one in the RCB, We can determine the position of the canary in the RCB. DiffGuard hooks the following calls that result in stack unwinding: __cxxabiv1::__cxa_end_catch and (sig)longjmp. In the cases of siglongjmp and longjmp, the new value of the stack pointer is retrieved from the contents of the __jmpbuf entry of the jump buffer that is passed to the calls, and we adjust the RCB index according to the canary in the stack frame pointed to by ESP.

Once all the components for ensuring the correctness of the canary randomizing are in place, DiffGuard provides different canaries for different function frames in the process. Meanwhile, DiffGuard registers a hook for the fork system call. Once fork is executed, in the context of the child process, and before fork returns, DiffGuard updates the canaries stored in child process' RCB except for the canaries of the function frames inherited from the parent process.

5 Evaluation

In this section we evaluate the performance overhead of DiffGuard and its effectiveness in protecting against byte-by-byte canary brute-force attacks. For our measurements we use the SPEC CPU2006 benchmark suite [18], as well as a series of popular (open-source) server applications. Overall, our GCC-based implementation of DiffGuard incurs an overhead ranging from 0.454% to 11.746%, with an average of 3.2%.

5.1 Effectiveness

We evaluate the effectiveness of DiffGuard from the following two aspects:

1. The identity of each function frame: DiffGuard ensure that each stack frame has its own canary. In order to verify the independence of canary in different stack frames, we instrument the SPEC CPU2006 benchmark suite, the purpose is to create a scan routine which is responsible for disclosing runtime stack data. Through stack sweeping algorithm introduced in Sect. 2, we confirmed that DiffGuard defends against the canaries disclosure attacks perfectly. In the contray, the existing canary based protections are prone to canaries disclosure attacks, and we have more than 90% probability to find canary when the number of function frames on runtime stack is greater than 10.
2. Preventing brute force attacks against canary based protections: We confirmed that DiffGuard defends against a set of publicly-available exploits [13,19] targeting the Nginx web server, which rely on brute-forcing stack canaries using the technique outlined in Sect. 2.

To verify that DiffGuard does not affect software correctness, we evaluated it over the SPEC CPU2006 benchmark suite, and also applied it to a variety of popular forking applications, such as the Apache and Nginx web servers, and the MySQL database servers. We observed no incompatibilities or any altered program functionality. As a final step of our correctness evaluation, we manually stress-tested DiffGuard over a series of scenarios that included combinations of multi-threaded and forking programs that executed setjmp/longjmp and triggered exceptions. In all cases we verified that DiffGuard successfully randomized the stack canaries (RCB) for all newly-created processes without causing any unexcepted behavior.

5.2 Performance

To obtain an estimate of DiffGuard's overhead on CPU intensive applications, we utilized the SPEC CPU2006 benchmark suite. The applications were compiled with the -fstack-protector option enabled. All experiments were performed on a virtual machine running Debian GNU/Linux v8, equipped with two 3.50 GHz four-core CPUs and 8 GB of R. Figure 5 summarizes the performance overhead of our GCC-based implementation of DiffGuard. All binaries were compiled with the DiffGuard plugin and had the -fno-omitframe-pointer compiler option asserted. DiffGuard incurs an average slowdown of 3.2% on the SPEC CPU2006 benchmarks. In all cases, the overhead of the GCC implementation of DiffGuard is below 11.74% for the SPEC CPU2006 benchmarks.

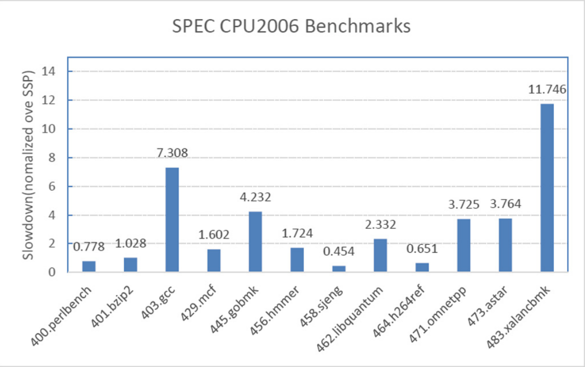


Fig. 5. The runtime overhead of DiffGuard (normalized over native execution).

6 Related Work

Canary-based stack protections were popularized by StackGuard [3]. Subsequently, ProPolice [20] introduced a series of GCC patches for StackGuard, which, among others, reordered the local variables in the stack, placing buffers after (local) pointers and function arguments in the stack frame. ProPolice was subsequently integrated in GCC, by RedHat, as the Stack Smashing Protector (SSP). As modern stack protectors follow a design similar to that of SSP, DiffGuard's architecture can be (easily) adopted by popular compilers due to its low performance overhead. With respect to preventing canary brute-force attacks, RAF-SSP [14] and DynaGuard [15], similarly to DiffGuard, aim to refresh stack-based canaries in networking servers. However, upon a fork system call, RAF SSP only updates the canary in the TLS area, ignoring the frames inherited by the parent process. This design fails to guarantee program correctness. DynaGuard use per-thread bookkeeping mechanism to guarantee program correctness, but the function frames of per-thread shared the identical canary word. This kind of identity makes it possible to be leaked.

A series of mechanisms have been proposed to protect the integrity of return addresses. RAD [21] is implemented as a compiler patch and creates a safe area where a copy of the return address is stored. Similar defenses have been implemented at the micro-architectural level [22], using binary rewriting [22], or by utilizing a shadow stack [23]. Apart from the fact that the previous mechanisms do not tackle the same problem as DiffGuard, they have not gained traction, mainly due to compatibility and performance issues (e.g., such mechanisms nullify several micro-architectural optimizations, like return address prediction) [25]. On the contrary, DiffGuard enhances a mechanism that has already seen wide adoption, without breaking accepted conventions around the format of the function prologue and epilogue, or the stack layout.

7 Conclusion

In this paper, we address a limitation of the current canary based protection mechanisms, which allows for brute-forcing the canary, byte-by-byte, in forking applications and stack-sweeping the canary, via information disclosure in non-forking applications. We resolve this issue by providing different canaries for different frames and proposing the dynamic update of the canaries in forked processes upon their creation. We present a design that utilizes a per-process, in-memory data structure to update the stack canaries at runtime, and we prototype the proposed architecture in DiffGuard, which is a compiler-based tool operating at the source code level. We evaluate that DiffGuard incurs an average overhead of 3.2% and can be easily integrated to modern compiler toolchains.

Acknowledgments. We would like to thank Theofilos Petsios et al. for their open source implementation of DynaGuard which helps ours quickly getting start of out work. When we have trouble in using SPEC CPU2006, Theofilos Petsios give us some advice. This work was supported in part by grants from the Chinese National Natural Science Foundation (61272078).

References

1. China National Vulnerability Database of Information Security(CNNVD)[Z/OL]. http://www.cnnvd.org.cn/
2. van der Veen, V., dutt-Sharma, N., Cavallaro, L., Bos, H.: Memory errors: the past, the present, and the future. In: Balzarotti, D., Stolfo, S.J., Cova, M. (eds.) RAID 2012. LNCS, vol. 7462, pp. 86–106. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33338-5_5
3. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: automatic adaptive detection and prevention of buffer overflow attacks

4. Etoh, H.: GCC extension for protecting applications from stack-smashing attacks
5. Microsoft.GS (Buffer Security Check) (2002). https://msdn.microsoft.com/en-us/library/8dbf701c.aspx
6. PaX Team: Address Space Layout Randomization (2003). https://pax.grsecurity.net/docs/aslr.txt
7. PaX Team: Non-executable pages design & implementation (2003). https://pax.grsecurity.net/docs/noexec.txt
8. Bulba and Kil3r: Bypassing stackguard and stackshield. Phrack, 56 (2002)
9. Richarte, G.: Four different tricks to bypass stackshield and stackguard protection, World Wide Web, **1** (2002)
10. Shacham, H., et al.: On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM Conference on Computer and Communications Security. ACM (2004)
11. Buchanan, E., et al.: When good instructions go bad: generalizing return-oriented programming to RISC. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. ACM (2008)
12. CVE-2012-3569. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-3569
13. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., Boneh, D.: Hacking blind. In: 2014 IEEE Symposium on Security and Privacy, pp. 227–242 (2014)
14. Marco-Gisbert, H., Ripoll, I.: Preventing brute force attacks against stack canary protection on networking servers. In: 12th IEEE International Symposium on Network Computing and Applications (NCA), pp. 243–250, August 2013
15. Petsios, T., Kemerlis, V.P., Polychronakis, M., Keromytis, A.D.: Dynaguard: armoring canary-based protections against brute-force attacks. In: Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC 2015, pp. 351–360. ACM, New York (2015)
16. Bryant, R., David Richard, O.H., David Richard, O.H.: Computer Systems: A Programmer's Perspective, vol. 2. Prentice Hall, Upper Saddle River (2003)
17. Stallman, R.M.: The GCC Developer Community: GNU Compiler Collection Internals (2017). https://gcc.gnu.org/onlinedocs/gccint/
18. Henning, J.L.: SPEC CPU2006 benchmark descriptions. ACM SIGARCH Comput. Archit. News **34**(4), 1–17 (2006)
19. Metasploit. Nginx HTTP Server 1.3.9-1.4.0 - Chuncked Encoding Stack Buffer Overflow (2013). http://www.exploit-db.com/exploits/25775/
20. Etoh, H.: GCC extension for protecting applications from stack-smashing attacks (2005). http://goo.gl/Tioc4C
21. Chiueh, T.-C., Hsu, F.-H.: RAD: a compile-time solution to buffer overflow attacks. In: Proceedings of ICDCS, pp. 409–417 (2001)
22. Park, Y.-J., Lee, G.: Repairing return address stack for buffer overflow protection. In: Proceedings of CF, pp. 335–342 (2004)
23. Corliss, M.L., Lewis, E.C., Roth, A.: Using DISE to protect return addresses from attack. ACM SIGARCH Comput. Archit. News **33**(1), 65–72 (2005)
24. Sinnadurai, S., Zhao, Q., fai Wong, W.: Transparent runtime shadow stack: protection against malicious return address modifications (2008). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702
25. Dang, T.H., Maniatis, P., Wagner, D.: The performance cost of shadow stacks and stack canaries. In: Proceedings of ASIACCS, pp. 555–566 (2015)