DynaGuard: Armoring Canary-based Protections against Brute-force Attacks

Theofilos Petsios
Columbia University
theofilos@cs.columbia.edu

Vasileios P. Kemerlis
Brown University
vpk@cs.brown.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

ABSTRACT

Over the past decade many exploit mitigation techniques have been introduced to defend against memory corruption attacks. W^X, ASLR, and canary-based protections are nowadays widely deployed and considered standard practice. However, despite the fact that these techniques have evolved over time, they still suffer from limitations that enable skilled adversaries to bypass them.

In this work, we focus on countermeasures against the byte-by-byte discovery of stack canaries in forking programs. This limitation, although known for years, has yet to be addressed effectively, and was recently abused by a series of exploits that allowed for the remote compromise of the popular Nginx web server and a full ASLR bypass in x86-64 Linux. We present DynaGuard, an extension to canary-based protections that further armors hardened applications against brute-force canary attacks. We have implemented DynaGuard in two flavors: a compiler-based version, which incurs an average runtime overhead of 1.2%, and a version based on dynamic binary instrumentation, which can protect binary-only applications without requiring access to source code. We have evaluated both implementations using a set of popular server applications and benchmark suites, and examined how the proposed design overcomes the limitations of previous proposals, ensuring application correctness and seamless integration with third-party software.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Design, Reliability, Security

Keywords

canary-based protection; canary re-randomization

ACSAC '15, December 7–11, 2015, Los Angeles, California, USA
ACM 978-1-4503-3682-6/15/12
DOI: http://dx.doi.org/10.1145/2818000.2818031.

1. INTRODUCTION

Among the many different types of memory corruption vulnerabilities actively exploited throughout the past two decades, stack buffer overflows remain the most frequently encountered [45], and various protection mechanisms have been proposed to prevent adversaries from abusing them. Widely deployed defenses include W^X [31, 36] and non-executable memory [20,30], Address Space Layout Randomization (ASLR) [18,35], and stack canaries [10,13,28]. However, none of these protections has fully eliminated stack smashing attacks.

In fact, although stack buffer overflows are no longer trivially exploitable, they are still present in popular applications, and are often used as a stepping stone in modern, multi-stage exploits that make use of techniques like Return-Oriented Programming (ROP) [40]. For instance, the recently-introduced Blind ROP (BROP) [4] attack requires only a stack-based memory corruption vulnerability and a service that restarts after a crash to automatically construct a ROP payload.

In certain cases, however, attacks can be rendered feasible due to *inherent* limitations of the deployed protection mechanisms. In this work, we examine one such case, related to a limitation of current canary-based protections [10,13], which allows adversaries to guess the value of a canary in forking applications, byte-by-byte, in a brute-force manner, and with a small number of attempts. This is possible due to the underlying process creation mechanism adopted by modern operating systems (OSes): new (child) processes are created by duplicating the memory of a template (parent) process.

This duplication of memory is known to have critical security implications. In the Android OS, it significantly weakens ASLR [23]. In case of stack canaries, it results in identical stack canary values being present in the parent and child process(es), after invoking the `fork` system call. Although this issue has been known since 2006 [16], it remains unaddressed. As a result, a previously-known technique for bypassing SSP (the canary mechanism of GCC [13]) and ASLR in Linux forking servers [38], was recently used in the wild by a series of exploits against the (hardened) Nginx web server [27]. The same canary bypass technique is also pivotal to the BROP attack [4], which automates the creation of ROP payloads given only a stack vulnerability and a service that restarts after a crash. Similarly, a full ASLR bypass in x86-64 Linux was possible [26], facilitated by a bypass of SSP using byte-by-byte canary guessing.

The severity and plethora of these exploits underline the need to revise the design of canary-based protections. To address the aforementioned issue, we present a scheme that armors (stack) canary protectors against attacks that brute-force the canary in forking applications. Specifically, through a lightweight, per-thread bookkeeping mechanism, our design enables the runtime update of the canary value in all protected (active) stack frames of the running thread, so that newly-forked processes get a *fresh* canary, different from the canary of their parent process. Contrary to previous work [25], our approach guarantees correctness and can be used as-is in production software.

We have prototyped our proposed solution in DynaGuard, which can be applied on top of existing stack smashing protection mechanisms to prevent brute-force guessing of canaries in forking programs. DynaGuard provides protection across the whole spectrum of applications, as it comes in two versions: when source code is available, a compiler-level version of DynaGuard, implemented as a GCC plugin, incurs just 1.2% runtime overhead over native execution, and is fully compatible with third-party libraries that are protected with the default canary mechanism; for binary-only executables, for which no source code is available, we have implemented a version of DynaGuard on top of Intel's Pin [24] Dynamic Binary Instrumentation (DBI) framework, which incurs an average slowdown of 2.92% over Pin and 1.7x over the native binary.

More importantly, the DBI-based and compiler-based versions of DynaGuard can be combined, if needed, to achieve full coverage of security-critical applications: parts for which access to source code is available can be compiled using DynaGuard's GCC-based version, while all other components can be protected at runtime using the Pin-based variant (or the inverse, for instance, if one has access to the OS libraries but not to the application's source code), allowing for cost-effective, targeted protection. In addition, the modular design of DynaGuard facilitates its integration to the back-end of popular compilers, such as GCC and LLVM, as it is naturally compatible with other compiler-assisted hardening techniques (e.g., PointGuard [9], CFI [1], CPI [22]).

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

- We present DynaGuard, a robust solution for armoring existing canary-based protections against brute-force attacks in forking applications.
- We have evaluated the correctness of a recently proposed solution [25] to the problem of canary brute force attacks, and demonstrate how DynaGuard overcomes its design limitations.
- We have implemented two versions of DynaGuard: a GCC plugin and a DBI tool built on top of Pin, protecting both binary-only applications and programs whose source code is available. Both implementations are publicly available as open-source projects.
- We demonstrate the practicality of our our approach, which incurs a runtime overhead of 1.2% when applied at the compiler level, and show that it can be easily adopted by popular compiler toolchains to further address security issues arising from the process creation mechanism of modern OSes [2, 5, 23, 32].

2. BACKGROUND

2.1 Canary-based Stack Smashing Protection

The main idea behind canary-based stack protections is to place a *tripwire* right after the return address, in every stack frame, to detect overwrites by buffer overflows. This tripwire, known as *canary* or *canary cookie*, is generated dynamically at the creation of each thread and is typically stored in the Thread Local Storage (TLS) area. During execution, whenever a new frame is created, the canary is pushed from the TLS into the stack. When the function epilogue is reached, and right before the destruction of the stack frame, the canary is compared against the one stored in the TLS. If the two values do not match, the process is terminated, as a stack smashing attack has occurred and has potentially overwritten the return address.

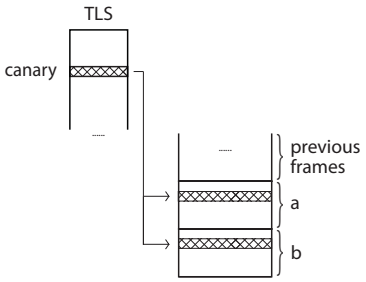
The stack smashing protector of GCC (i.e., SSP [38]) adopts a series of layout transformations to further prevent bypasses. Specifically, SSP creates local copies of the function arguments so that the original arguments are never referenced, and also re-orders local variables in the stack, placing them always behind buffers, in order to prevent pointer variables from being overwritten. Although SSP still faces some limitations [38] (e.g., it does not protect each buffer separately and cannot create argument copies for functions with a variable number of arguments), it hinders stack smashing significantly, as the attacker is unable to read the TLS area and needs to have knowledge of the canary value to successfully corrupt the saved return address.

2.2 Bypassing Canary-based Defenses

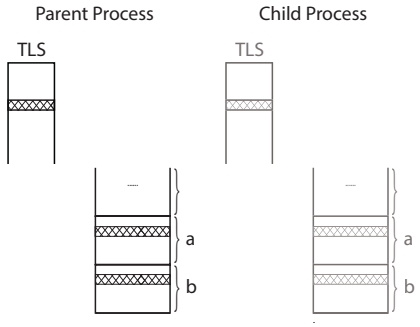
Attackers wishing to bypass the stack canary protection, while lacking any additional information, are restricted to a random-guessing attack, the efficiency of which is bounded by the entropy of the canary. The canary is four bytes long on 32-bit architectures and eight bytes on 64-bit ones, and is, in principle, random. However, there do exist cases [38] where the canary either has a fixed value or always contains a `NULL` terminating byte, reducing its entropy. If we ignore such cases, an attacker will need to perform $2^{32}/2^{64}$ attempts for brute-forcing the canary in the x86/x86-64 architecture. However, under certain conditions, an adversary can brute-force the canary with much fewer attempts, by abusing the process creation mechanism.

Whenever a process is forked, it *inherits* the address space of its parent process, i.e., all the in-memory code and data, including the canaries placed in the stack frames and the TLS. If `execve` is called after `fork`, all memory regions of the calling process are overwritten by the program that gets loaded, whereas, if not, they remain unchanged. Hence, due to this duplication of memory, in canary-protected applications, attackers are able to brute-force the canary, byte-by-byte, as long as they are able to force child processes to be forked from the same parent process and check if any of these child processes has crashed or not (i.e., after successfully triggering a stack buffer overflow on the latter). Unfortunately, the aforementioned conditions are met in several applications, most notably in multi-process, network-facing programs, such as web and database servers, where new processes are forked to service incoming requests.

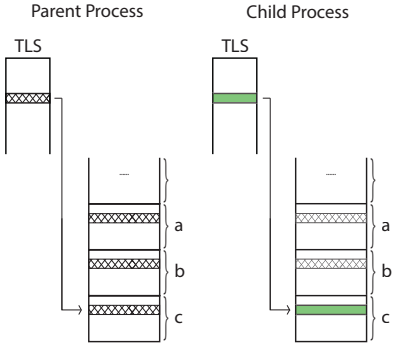
Specifically, the byte-by-byte canary brute-force attack works as follows. Initially, the adversary exploits a stack buffer overflow vulnerability and overwrites only the first



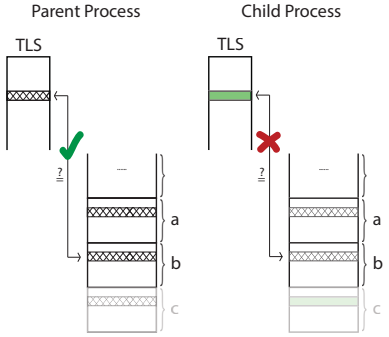
(a) Parent process before fork. Frames a and b are canary-protected.



(b) The forked (child) process is an exact copy of the parent.



(c) Updating the canary in the TLS of the child will result in two canary values being present its stack frames.



(d) Epilogue check for function c succeeds, but will fail once execution reaches frames inherited from the parent.

Figure 1: Renewing the canary in the TLS alone will result in an abort as soon as execution reaches a stack frame inheritted from the parent process.

byte of the canary. If the respective process aborts, the attacker knows that the canary check in the function epilogue failed. Subsequently, the attacker forces the forking of a new (child) process and tries with a different byte value. As new processes are forked from the same parent, the attacker only needs 256 requests (128 on average) to brute-force the first byte of the canary. Armed with the correct byte value, the attacker can then brute-force the second byte using the same approach, totalling $4 * 256 = 1024$ tries, at most, for the full discovery of the canary in 32-bit architectures, and $8 * 256 = 2048$ in 64-bit architectures.

3. CHALLENGES AND REQUIREMENTS

Had the stack canary been different for the newly forked processes, attacks that brute-force the canary value byte-by-byte would have been infeasible. Since the canary is pushed from the TLS to each newly created stack frame, to modify the canary upon a `fork` system call we must modify both the canary *in the TLS*, as well as the canaries *in the stack frames* that the forked process inherited.

However, this is not feasible in current canary-based protectors, as hardened programs do not store any information regarding *where* (in their address space) their canaries are stored. Thus, once a child process is forked, there is no way for it to access the canaries in the frames it inherited from its parent process, as it cannot differentiate canaries from random data that may be residing in the stack. As a result, under the current design of the stack canary mechanism, the only value that can be updated at runtime upon a `fork` is that of the canary in the TLS. 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.

To demonstrate this point, let us consider the example shown in Fig. 1, where a process has two canary-protected frames (`a` and `b`) in its call stack the moment it invokes `fork`. Initially, all frames have the same canary value, copied from the TLS into the stack at the creation of each frame (Fig. 1a). Once the child process is forked, it inherits its parent's frames in their exact state, as expected by the process creation mechanism (Fig. 1b). In the child process' context, before the process starts executing, we modify the canary cookie in the TLS, thus the newly created frames in the child now have a different canary value, as is the case for frame `c` (Fig. 1c).

The canary checks for the newly created frames will succeed, since the canaries in each frame have the same value with the (new) canary stored in the TLS. Thus, frame `c` will be successfully destroyed both for the parent and the child process. However, once execution reaches one of the inherited frames in the child process (Fig. 1d), the canary in the TLS will have a new value and the canaries in the stack will still have their old value, causing the child process to terminate. In addition, the stack smashing check in the function epilogue may similarly fail if, at some point, during the execution of the child process, either an exception is triggered or `setjmp/longjmp` is invoked, forcing the stack to unwind.

Successfully updating the canary in child process after a `fork` would require modifying the current implementations to incorporate a bookkeeping mechanism that allows for a runtime update of the inherited stack canaries. However, incorporating such a mechanism to production systems proves to be a challenging task. First, new protections should be made modular so as to be compatible with third-party software. Therefore, a solution that requires a custom version of system or third-party libraries (e.g., `glibc`) would not be easy to deploy at a large scale. Similarly, the incremental performance overhead of any new canary protection design should be within acceptable limits for production systems and respect existing micro-architectural hardware optimizations (e.g., hardware prediction schemes), without requiring major changes to current implementations.

To the best of our knowledge, a robust and readily deployable mechanism for armoring canary-based defenses against brute-force attacks has not yet been proposed. As a result, attacks like those described in Section 2 are still present, exploiting a limitation in a widely deployed defense. The variety of applications and OSes that are protected with some form of stack canaries is a major obstacle towards the adoption of new countermeasures, as backward compatibility and testing of new designs is not trivial. In the following sections, we discuss how DynaGuard solves the problems discussed above while preserving application correctness.

4. DESIGN

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. 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 forked process has a fresh canary.

As we discussed in Section 3, current stack canary protectors do not keep any information regarding where the canaries are located within the stack of each thread. Therefore, DynaGuard's design should allow each running process to access and modify all of its stack canaries at runtime. To achieve this goal, DynaGuard performs a per-thread runtime bookkeeping of all the canaries that are pushed in the stack during execution, using a lightweight buffer allocated dynamically upon each thread's creation (this buffer is stored in the heap). Figure 2 illustrates this scheme in more detail.

DynaGuard's *canary address buffer* (CAB; Fig. 2a) holds references to all the canaries stored in the stack of the running process. When a child process is forked, the CAB of the parent process is copied to the child process (Fig. 2b). Before execution starts in the child context, DynaGuard modifies the canary value in the TLS, as well as in all the stack addresses referenced by the entries in the thread's CAB. Likewise, whenever a canary-protected frame is pushed onto the stack, the address of the canary is stored in CAB (Fig. 2c) and, once a canary-protected function returns, the respective address is removed (Fig. 2d).

The aforementioned design allows DynaGuard to successfully modify the canary values for newly-created processes, without facing any of the limitations described in Section 3. Specifically, it allows for a seamless integration with third-party software and with 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 because of `setjmp/longjmp`: as the stack always grows towards lower addresses, the addresses that were last saved in CAB must always be lower than the current value of the stack pointer. Thus, DynaGuard can hook any stack unwinding operation and modify the canary address buffer accordingly, so that the latter is always consistent with the program stack. In this manner, and contrary to recently proposed solutions [25], application correctness is preserved even when all frames are canary-protected.

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. Finally, due to the small number of canary-protected frames that (on average) are active at runtime, and since DynaGuard only needs to store one address per protected frame, this design is very efficient with respect to memory and CPU pressure.

5. IMPLEMENTATION

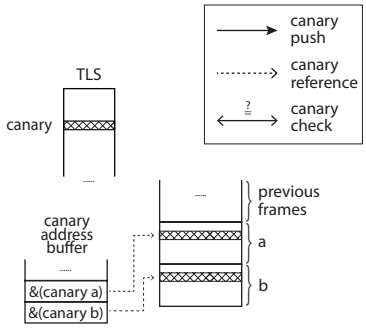
5.1 Compiler-based DynaGuard

The compiler-based version of DynaGuard 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 ∼1250 lines of C++ code.

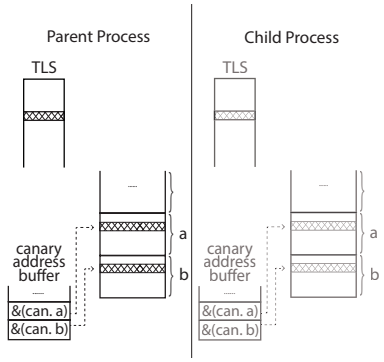
Several requirements must be accomplished to implementing DynaGuard at the compiler level, while at the same time maintaining compatibility with third-party software: *(a.)* DynaGuard must instrument all the canary push/pop events and perform its bookkeeping on a per-thread basis; *(b.)* DynaGuard must hook each `fork` system call and update the canaries in the child process as described in Section 4; *(c.)* DynaGuard must intercept all calls related to stack unwinding and ensure that the CAB gets updated accordingly. The first requirement is handled by DynaGuard's GCC plugin. All other requirements are handled by DynaGuard's dynamic shared library (runtime), which ensures the proper management of the CAB for every thread.

5.1.1 GCC Plugin Implementation

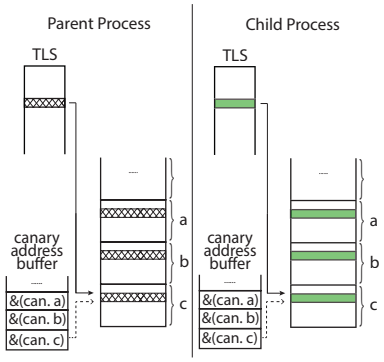
Beginning with v4.5.1, GCC added support for extending the compilation pipeline via plugins that operate on top of the various intermediate languages (ILs) used throughout the translation process. The GCC pipeline consists of three distinct components, namely the front-end, middle-end, and back-end, which transform the input into the GENERIC, GIMPLE, and RTL ILs [43]. DynaGuard is registered as an RTL optimization pass and loaded by GCC right after the `vartrack` pass. The first reason for placing DynaGuard late in the RTL optimization pipeline is to ensure that most of the important optimizations have already been performed, and, as a result, DynaGuard's instrumentation is never added to irrelevant code. In addition, in this manner, we ensure that all injected instructions, which perform the necessary bookkeeping, will remain at their proper locations and will not be optimized by later passes.



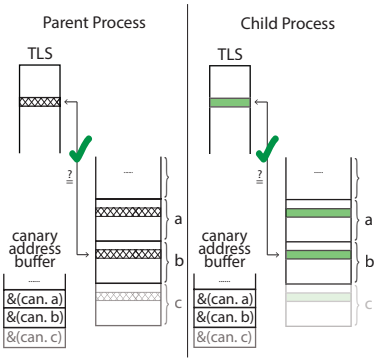
(a) Parent before fork: the canary address buffer (CAB) contains the addresses of all canaries in the process' stack frames.



(b) After forking, the canary address buffer of the parent is copied to the child. All canary addresses are now accessible by the child.



(c) The per-thread CAB is updated upon frame creation/destruction, to be kept consistent with the stack.



(d) Epilogue checks work as expected since all stack canaries are consistent with the canary in the TLS.

Figure 2: The design of DynaGuard allows for a complete update of all canaries in the child process.

Apart from inserting all stack canary addresses to CAB, the DynaGuard GCC plugin must also modify the canary setup and check inside each canary-protected frame, to prevent the DynaGuard-protected application from using the standard `(g)libc` canaries. This is necessary to allow the modification of the canary at runtime without affecting any checks in libraries that are not compiled with DynaGuard. The canary initialization that occurs during the creation of threads and processes is exactly the same in DynaGuard and in `glibc`, with the only difference being that the DynaGuard canary 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 DynaGuard canary. Upon a `fork`, all DynaGuard canaries get updated without affecting any checks in modules or libraries that use the legacy `glibc` canaries.

DynaGuard stores the starting address of CAB, its total size, and its current size, in the TLS, together with the DynaGuard canary. To ensure compatibility with current versions of `glibc`, we reserve 4 out of the 8 free `__padding` elements of the `tcbhead_t` data structure for that purpose. In x86-64, the reserved TLS offsets range from `0x2a0` to `0x2b8`. In particular, `%fs:0x2a0` holds the base address of CAB, `%fs:0x2a8` keeps the current index in the CAB (i.e., how many canary addresses are stored), `%fs:0x2b0` holds the total size of the buffer, and finally, `%fs:0x2b8` stores the DynaGuard canary.

Original

```
;function prologue
push   %rbp
mov    %rsp,%rbp
sub    $0x40,%rsp
;canary stack placement
mov    %fs:0x28,%rax
mov    %rax,-0x8(%rbp)
xor    %eax,%eax

          ...

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

DynaGuard

```
push   %rbp
mov    %rsp,%rbp
sub    $0x40,%rsp
push   %r14                    (1)
push   %r15
lea    -0x8(%rbp),%rax         (2)
mov    %fs:0x2a0,%r14          (3)
mov    %fs:0x2a8,%r15          (4)
mov    %rax,(%r14,%r15,8)      (5)
incq   %fs:0x2a8               (6)
pop    %r15                    (7)
pop    %r14
mov    %fs:0x2b8,%rax          (8)
mov    %rax,-0x8(%rbp)
xor    %eax,%eax
          ...
decq   %fs:0x2a8               (9)
mov    -0x8(%rbp),%rcx
xor    %fs:0x2b8,%rcx          (10)
je     <exit>
callq  <__stack_chk_fail@plt>
```

Figure 3: Assembly excerpt for a binary compiled with `-fstack-protector`, with and without DynaGuard. The canary bookkeeping code added by the DynaGuard plugin is shown on the right (highlighted).

Figure 3 shows the bookkeeping instructions inserted by the DynaGuard GCC plugin. Right after the function prologue, before the canary gets pushed to the stack, the address in which the canary will be stored must be saved in the CAB. Initially, the address is loaded in the *clobbered* register used for the canary stack placement (2). Subsequently, DynaGuard retrieves the address of the CAB from the TLS (3)

and the index of the next element to be written (4). Next, it stores the canary address in the CAB (5) and increments the buffer index (6). Finally, the canary is fetched from the TLS (8) and saved onto the stack. For this purpose, if no registers are free, DynaGuard needs to spill two registers for its bookkeeping ((1), (7)).[1] Likewise, the canary check in the function epilogue is modified to decrease the index in CAB (9), and check against the DynaGuard canary instead of the `glibc` canary (10).

5.1.2 Runtime Implementation

The code added by the DynaGuard GCC plugin assumes that the respective entries in the TLS are properly initialized, and that a canary address buffer with available space exists. The logic for the CAB setup and update, as well as the hooking of `fork` system calls and stack unwinding routines, is handled by DynaGuard's runtime environment. The library (PIC module) implementing that runtime is loaded via the `LD_PRELOAD` mechanism into the address space of the running application.

The CAB is allocated in the heap for each thread of the running program. In order to allocate the CAB before the main thread starts executing, we register—in the DynaGuard runtime—a *constructor* routine to be called before the `main` function of the application. This routine performs the CAB allocation and sets the appropriate values in the main thread's TLS. For all other threads that get created, DynaGuard 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 CAB for each thread that finishes execution is registered via the `pthread_cleanup_push(/pop)` mechanism.

To ensure that the CAB of each thread is never full, DynaGuard marks the final page in the CAB as read-only and registers a signal handler for the `SIGSEGV` signal. Inside the signal handler, DynaGuard detects whether the fault is due to DynaGuard's instrumentation (i.e., when DynaGuard tries to push an address for a canary in the read-only page of the CAB) and allocates additional memory for the CAB 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`, DynaGuard maintains a hashmap of all the running threads and their TLS entries. Inside the signal handler, DynaGuard iterates through all the threads in the hashmap and examines whether the memory location that caused the fault falls within an allocated CAB.[2]

Since registering a signal handler may overwrite existing handlers of the application, DynaGuard also hooks all `signal` and `sigaction` calls. If the signal is different than `SIGSEGV`, DynaGuard does not alter the application handler. Otherwise, DynaGuard saves the application's handler and then overwrites it with its own handler. If the fault did not occur due to a write into one of the protected pages of the allocated CABs, DynaGuard passes the signal to the saved application handler.

Lastly, in order to ensure that the CAB contains canary addresses only for *active* frames, DynaGuard checks for any stack unwinding and removes the entries corresponding to destroyed frames from the CAB. This is based on the simple observation that, at any point during execution, since the stack always grows towards lower addresses, all addresses stored in a CAB should be higher than the current stack pointer. DynaGuard hooks the following calls that result in stack unwinding: `__cxxabiv1::__cxa_end_catch` and `(sig)longjmp`. In the case of `__cxa_end_catch`, the stack pointer has already been updated to its new value and DynaGuard can check if the CAB is consistent with the unwound stack. 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 the check is performed accordingly.

Once all the components for ensuring the correctness of the canary bookkeeping are in place, DynaGuard registers a hook for the `fork` system call. Once `fork` is executed, in the context of the child process, and before `fork` returns, DynaGuard sets a new canary in the process' TLS and updates all the canaries inherited by the parent process.

5.2 DBI-based DynaGuard

If source code is not available, or when re-compiling a program is not an option, we can still protect security-critical binaries using the DBI-based flavor of DynaGuard, implemented over Intel's Pin [24] dynamic binary instrumentation framework. Whenever a binary is instrumented with Pin, execution occurs within three distinct contexts: the context of the instrumented *application*, the context of the *pintool* which guides the instrumentation process, and finally, within the context of *Pin* itself, which controls the context-switching between the pintool and the application. From the perspective of the underlying OS, only one process is running. In reality though, `glibc` and other libraries are loaded multiple times, and the executing code is either code of the native application, instrumentation code inserted by the pintool, or code belonging to Pin itself.

Before executing any application code, during the *instrumentation phase*, Pin instruments the native binary with new code, or *analysis code* as it is colloquially known, specified by the pintool in use. DynaGuard's instrumentation routines define *where* in the binary the analysis code will be inserted (e.g., before or after a particular instruction, system call, or library load), which routines will be called when the new code is triggered, and what arguments will be passed to them. Note that instrumentation happens only once.

To minimize DynaGuard's runtime performance overhead, our goal is to minimize the instrumentation code, and more importantly, to optimize the analyses routines, as it is the analysis code that dominates the performance overhead. Due to the DynaGuard pintool's model of execution, updating the canary in the TLS of the instrumented process with a system call like `prctl`, would result in a TLS update within the context of Pin, instead of that of the instrumented application, since libraries like `glibc` are duplicated and the execution of `prctl` occurs under Pin. For this reason, Pin exposes an API call for getting the base address of the TLS area of the instrumented program. DynaGuard registers a callback routine to be executed in the child process when-

[1] *Def-use* analysis, as well as checks for *leaf* functions that will never call another protected function can be used to further improve the performance of DynaGuard. However, in our measurements, we considered the worst-case scenario and always spilled registers.

[2] The memory address that caused the fault is accessible through the `si_addr` field of the `siginfo_t` data structure that is passed to DynaGuard's signal handler.

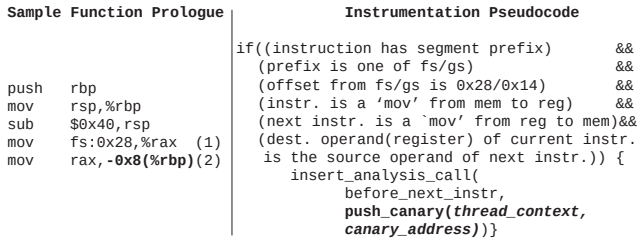


Figure 4: Pinpointing the canary push operation inside the function prologue. The instrumentation code selects instruction (1) and inserts the analysis routine `push_canary` between instructions (1) and (2).

ever a `fork` system call executes; this callback calculates the base address of the TLS segment and is responsible for the canary update (both in the TLS and in all active stack frames).

To compute the new canary value, DynaGuard utilizes the kernel's random number generator (`/dev/urandom`) and, if such a device is not available, falls back to `arc4random`.

5.2.1 Canary Bookkeeping

DynaGuard utilizes a lightweight CAB implementation inspired by `libdft` [21]. In particular, DynaGuard allocates a per-thread CAB in the process' heap, and then uses one of Pin's scratch registers as a pointer to this buffer. This optimization has the benefit of minimizing the instrumentation code and eliminating the unnecessary locking logic of the built-in trace buffer [19]. In order to correctly update the canary on each newly-forked process, CAB holds the address of every stack canary in the active frames, as described in Section 4. CAB is updated upon the following events: (a) canary push (function prologue), (b) canary pop (function epilogue), and (c) stack unwinding (because of exception handling or `setjmp/longjmp`).

In the following, we examine how DynaGuard handles each of the previous scenarios. As mentioned in Section 2, the canary is originally stored in the TLS. Upon a *canary push* from the TLS to a stack frame, the address at which the canary is stored is saved in the CAB, as illustrated in Fig. 4: DynaGuard inserts a call to `push_canary` before `mov rax,−0x8(%rbp)` executes. The arguments to this analysis function are the thread context (holding a reference to the process' CAB through Pin's scratch register), as well as the address in which the canary will be stored in the stack (which is known at runtime as instruction (2) is about to execute).

All the analysis routine needs to perform at this point is to store the canary address into the CAB and increment the buffer index. In addition, since the number of canary addresses that are present in the CAB at any given time equals the number of canary-protected frames that are present in the process' stack, DynaGuard will be able to successfully update all canaries upon a `fork`.

Likewise, whenever a protected frame is destroyed, the respective canary address is removed from the CAB. In order to provision for stack unwinding, the DynaGuard pintool examines whether there is any modification of the stack pointer during runtime. As dynamic binary instrumentation enables the monitoring of all executed instructions, we do not need to perform any hooking of `longjmp` or exception handling calls, as we did in the GCC implementation; instead, DynaGuard checks for changes in the stack pointer value and then updates the CAB accordingly.

6. DISCUSSION

In this section we discuss alternative DynaGuard designs and elaborate on their drawbacks and benefits. Subsequently, we describe how the proposed architecture can serve as the basis for resolving other security problems arising from the (current) OS process creation mechanism.

As an alternative design for DynaGuard, one could consider implementing a stack frame chain, using a mechanism similar to the one used by exception handlers. In such a design, all canary-protected frames would be chained together, with each protected frame holding a pointer to the previous frame that is canary-protected. This eliminates the need for a CAB, as the linked list would allow for "unwinding" the stack and updating all canaries directly at runtime.

Unfortunately, this approach has several drawbacks, which arise mainly from the fact that the pointer to the previous frame that is canary-protected should *itself* be protected (by a canary), to prevent it from being modifiable—otherwise, an attacker could tamper with the stack unwinding process. This would require the modification of current implementations in a non-transparent manner, and would break compatibility with legacy software and third-party libraries. On the contrary, our proposed design can be *transparently* applied to production systems.

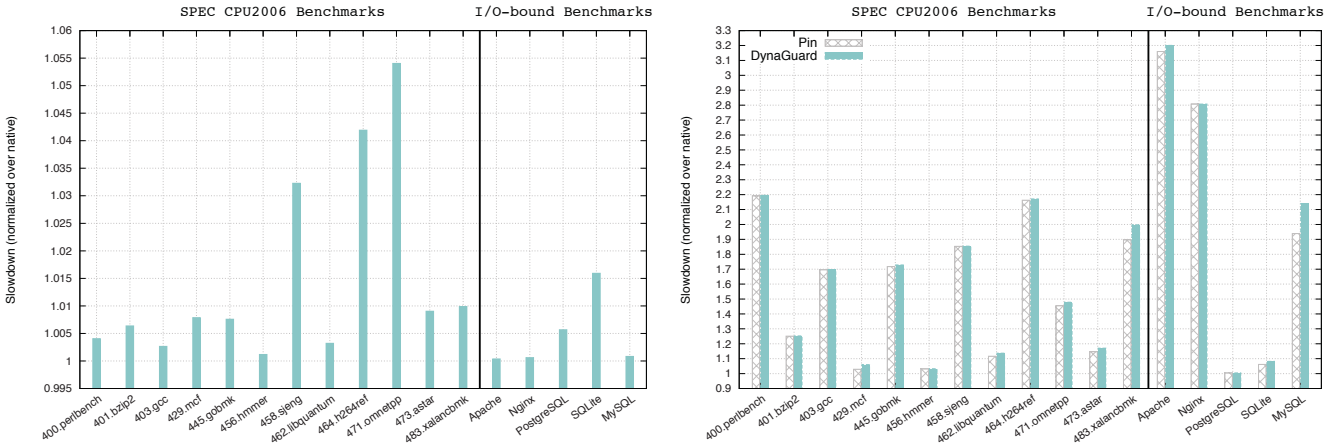
More importantly, DynaGuard's architecture offers the foundation for a compiler-level solution to several other problems arising from the current OS process creation model. For instance, the current process creation mechanism may affect the trustworthiness of cryptographic Pseudo-Random Number Generators (PRNGs), as was the case with OpenSSL [5, 32] and LibreSSL [2]. The PRNGs of these libraries produced the same random number chain in parent and child processes. Modern compilers could adopt DynaGuard's design to support per-process data bookkeeping that would enable entropy gathering and transparent updating of the state of PRNGs, similarly to previously-proposed compiler schemes that extract entropy from the OS at boot time [37].

7. EVALUATION

In this section we evaluate the performance overhead of DynaGuard and its effectiveness in protecting against byte-by-byte canary brute-force attacks. For our measurements we use the SPEC CPU2006 benchmark suite [17], as well as a series of popular (open-source) server applications. Overall, our GCC-based implementation of DynaGuard incurs an overhead ranging from 0.03% to 5.4%, with an average of 1.2%. The Pin-based version of DynaGuard incurs an average overhead of 170.66%. However, this overhead is dominated by the native DBI framework (Pin), with DynaGuard adding only 2.92% on top of that, on average.

7.1 Effectiveness

We confirmed that DynaGuard defends against a set of publicly-available exploits [4, 27] targeting the Nginx web server, which rely on brute-forcing stack canaries using the technique outlined in Section 2. To verify that DynaGuard does not affect software correctness, we evaluated it over the SPEC CPU2006 benchmark suite, and also applied it to a



(a) GCC-based version. (b) Pin-based version.

Figure 5: The runtime overhead of DynaGuard (normalized over native execution).

variety of popular forking applications, such as the Apache and Nginx web servers, and the PostreSQL, MySQL, and SQLite database servers. We observed no incompatibilities or any altered program functionality.

As a final step of our correctness evaluation, we manually stress-tested DynaGuard 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 DynaGuard successfully randomized the stack canaries for all newly-created processes without causing any unwanted behavior.

7.2 Performance

To obtain an estimate of DynaGuard's overhead on CPU-intensive applications, we utilized the SPEC CPU2006 benchmark suite, whereas in order to examine how it performs on I/O-bound programs, we used a series of popular web and database servers: Apache, Nginx, PostgreSQL, MySQL, and SQLite. For all the server applications, except MySQL, we used the Phoronix [39] benchmark suite and maintained its default configuration, modifying only the compilation stage. For benchmarking MySQL we used the SysBench benchmark tool [33]. Note that in all cases, applications were compiled with the `-fstack-protector` option enabled. All experiments were performed on a system running Debian GNU/Linux v8, equipped with two 2.40GHz six-core Intel Xeon E5645 CPUs and 48GB of RAM.

Figure 5a summarizes the performance overhead of our GCC-based implementation of DynaGuard. All binaries were compiled with the DynaGuard plugin and had the `-fno-omitframe-pointer` compiler option asserted. DynaGuard incurs an average slowdown of 1.5% on the SPEC CPU2006 benchmarks, and 0.46% on the server applications. In all cases, the overhead of the GCC implementation of DynaGuard is below 5.4% for the SPEC CPU2006 benchmarks and below 1.5% for the I/O-bound applications, with the overhead being negligible ($< 0.5\%$) for the Apache and Nginx web servers.

Figure 5b shows the performance overhead of the Pin-based version of DynaGuard. Specifically, the incremental overhead over the native DBI framework is less than 2.92% for all the tested applications. The overall slowdown over the native binary ranges from 0.4% to 3.2x. In particular, the slowdown for the SPEC CPU2006 applications ranges from 3.2% to 2.19x, with an average slowdown of 1.56x. For the database servers, the overhead is 0.4%, 8.19%, 214% for the PostreSQL, SQLite, and MySQL servers respectively, while for the Apache and Nginx web servers the overhead is 3.2x and 2.8x, respectively. As mentioned earlier, this high overhead is mostly due to the underlying dynamic binary instrumentation framework.

8. RELATED WORK

Canary-based stack protections were popularized by StackGuard [10]. Subsequently, ProPolice [13] 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) [38]. Parallel to the evolution of SSP, Microsoft introduced the GuardStack (`/GS`) [28] compiler flag, which also places a canary in the stack using heuristics similar to those of SSP. As modern stack protectors follow a design similar to that of SSP, DynaGuard's architecture can be (easily) adopted by popular compilers, as it incurs a negligible overhead. Moreover, other canary-based defenses, like ValueGuard [44], which aim to protect against data-only attacks can also benefit from DynaGuard's dynamic update of canaries.

With respect to preventing canary brute-force attacks, RAF SSP [25], similarly to DynaGuard, aims 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, in the general case, as it assumes that a child process never reuses inherited frames legitimately, and checks for the `noreturn` attribute to avoid scenarios similar to those shown in Fig. 1. In addition, RAF SSP does not handle exceptions, a vital application component for avoiding the ungraceful termination of server applications.

A series of mechanisms have been proposed to protect the integrity of return addresses. RAD [7] is implemented as a compiler patch and creates a safe area where a copy of the return address is stored. Similar defenses have been im-

plemented at the micro-architectural level [34], using binary rewriting [8], or by utilizing a shadow stack [41]. Apart from the fact that the previous mechanisms do not tackle the same problem as DynaGuard, 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) [11]. On the contrary, DynaGuard 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.

Lastly, several protections have been proposed against memory corruption attacks, and, as such, are beyond the scope of the current work. ASLR-based defenses randomize, among others, the base address of the stack [35] or introduce variable spacing between stack frames [3, 15] to protect against (stack) object corruption. Protection mechanisms like W^X [12] and DEP [30] prevent the execution of injected code, by ensuring that memory is never both writable and executable, whereas defenses like SafeSEH [29] and SEHOP [42] attempt to prevent exploits that abuse the exception handling mechanism to execute arbitrary code. Finally, several protection mechanisms abandon the current stack organization completely. StackArmor [6] operates at the binary level and relies on a combination of randomization and isolation to make the stack objects appear as if drawn from a fully randomized space. SafeStack [22] splits the stack into safe and unsafe regions, and enforces code pointer integrity to prevent control-flow hijacking attacks. However, such mechanisms also have limitations, in terms of both performance and effectiveness [11, 14].

9. 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. We resolve this issue by 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 DynaGuard, which comes in two flavors: a compiler-based one operating at the source code level and a DBI-based one that operates at the binary level. The compiler-based version of DynaGuard incurs an average overhead of 1.2% and can be easily integrated to modern compiler toolchains.

Availability

Our prototype implementation of DynaGuard is available at: `https://github.com/nettrino/dynaguard`

Acknowledgments

We are grateful to George Kontaxis and George Argyros for their valuable feedback on earlier versions of this paper. This work was supported by the Office of Naval Research (ONR) through contracts N00014-12-1-0166 and N00014-15-1-2378. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government or ONR.

10. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proc. of CCS*, pages 340–353, 2005.

[2] A. Ayer. LibreSSL's PRNG is Unsafe on Linux. `https://www.agwa.name/blog/post/libressls_prng_is_unsafe_on_linux`, 2014.

[3] S. Bhatkar, D. C. DuVarney, and S. R. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proc. of USENIX Sec*, pages 271–286, 2005.

[4] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking Blind. In *Proc. of IEEE S&P*, pages 227–242, 2014.

[5] M. Boßlet. OpenSSL PRNG Is Not (Really) Fork-safe. `https://goo.gl/sZuopi`, 2013.

[6] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries. In *Proc. of NDSS*, 2015.

[7] T.-c. Chiueh and F.-H. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proc. of ICDCS*, pages 409–417, 2001.

[8] M. L. Corliss, E. C. Lewis, and A. Roth. Using DISE to Protect Return Addresses from Attack. *ACM SIGARCH Computer Architecture News*, 33(1):65–72, 2005.

[9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proc. of USENIX Sec*, pages 91–104, 2003.

[10] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of USENIX Sec*, pages 63–78, 1997.

[11] T. H. Dang, P. Maniatis, and D. Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proc. of ASIACCS*, pages 555–566, 2015.

[12] T. de Raadt. Advances in OpenBSD. `http://www.openbsd.org/papers/csw03/index.html`, 2003.

[13] H. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://goo.gl/Tioc4C`, 2005.

[14] I. Evans, S. Fingeret, J. González, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proc. of IEEE S&P*, 2015.

[15] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proc. of USENIX Sec*, pages 475–490, 2012.

[16] B. Hawkes. Exploiting OpenBSD. `http://inertiawar.com/openbsd/`, 2006.

[17] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[18] M. Howard, M. Miller, J. Lambert, and M. Thomlinson. Windows ISV Software Security

Defenses. https://msdn.microsoft.com/en-us/library/bb430720.aspx, 2010.

[19] Intel. Pin: Instrumentation API. http://www.cs.virginia.edu/kim/publicity/pin/docs/31933/Pin/html/group__INS__INST__API.html, 2009.

[20] Jonathan Corbet. x86 NX support. https://lwn.net/Articles/87814/, 2003.

[21] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proc. of VEE*, pages 121–132, 2012.

[22] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proc. of OSDI*, pages 147–163, 2014.

[23] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *Proc. of IEEE S&P*, pages 424–439, 2014.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of PLDI*, pages 190–200, 2005.

[25] H. Marco-Gisbert and I. Ripoll. Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers. In *Proc. of NCA*, pages 243–250, 2013.

[26] H. Marco-Gisbert and I. Ripoll. On the Effectiveness of Full-ASLR on 64-bit Linux. In *DeepSeC*, 2014.

[27] Metasploit. Nginx HTTP Server 1.3.9-1.4.0 - Chuncked Encoding Stack Buffer Overflow. http://www.exploit-db.com/exploits/25775/, 2013.

[28] Microsoft. /GS (Buffer Security Check). https://msdn.microsoft.com/en-us/library/8dbf701c.aspx, 2002.

[29] Microsoft. /SAFESEH (Image has Safe Exception Handlers). https://msdn.microsoft.com/en-us/library/9a89h429.aspx, 2003.

[30] Microsoft. A detailed description of the Data Execution Prevention (DEP) feature. http://support.microsoft.com/kb/875352, 2013.

[31] OpenBSD. i386 W^X. https://marc.info/?l=openbsd-misc&m=105056000801065, 2003.

[32] OpenSSL. Random fork-safety. https://wiki.openssl.org/index.php/Random_fork-safety, 2014.

[33] Oracle. MySQL Benchmark Tool. https://dev.mysql.com/downloads/benchmarks.html, 2015.

[34] Y.-J. Park and G. Lee. Repairing Return Address Stack for Buffer Overflow Protection. In *Proc. of CF*, pages 335–342, 2004.

[35] PaX Team. Address Space Layout Randomization. https://pax.grsecurity.net/docs/aslr.txt, 2003.

[36] PaX Team. Non-executable pages design & implementation. https://pax.grsecurity.net/docs/noexec.txt, 2003.

[37] PaX Team. new gcc plugin: latent entropy extraction. https://grsecurity.net/pipermail/grsecurity/2012-July/001093.html, 2012.

[38] A. 'pi3' Zabrocki. Scraps of notes on remote stack overflow exploitation. http://phrack.org/issues/67/13.html, 2010.

[39] PTS. Phoronix Test Suite, June 2015. http://www.phoronix-test-suite.com.

[40] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. of CCS*, pages 552–561, 2007.

[41] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent Runtime Shadow Stack: Protection against malicious return address modifications, 2008. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702.

[42] skape. Preventing the Exploitation of SEH Overwrites. http://www.uninformed.org/?v=5&a=2&t=txt, 2006.

[43] R. M. Stallman and the GCC Developer Community. GNU Compiler Collection Internals. https://gcc.gnu.org/onlinedocs/gccint/, 2015.

[44] S. Van Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens. ValueGuard: Protection of native applications against data-only buffer overflows. In *Proc. of ICISS*, pages 156–170. 2010.

[45] V. van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In *Proc. of RAID*, pages 86–106, 2012.