Dynamic Canary Randomization for Improved Software Security

William H. Hawkins, Jason D. Hiser, Jack W. Davidson
Computer Science Department, University of Virginia
Charlottesville, VA 22904
{whh8b,jdh8d,jwd}@virginia.edu

ABSTRACT

Stack canaries are a well-known and effective technique for detecting and defeating stack overflow attacks. However, they are not perfect. For programs compiled using *gcc*, the reference canary value is randomly generated at program invocation and fixed throughout execution. Moreover, for software running on the Linux operating system, canary values are inherited from the parent process and only changed if/when the child process `exec()`s a different program. Researchers and others have exploited these behaviors to craft real-world attacks that bypass the protections of stack canaries. This paper describes a moving-target stack canary technique that prevents such attacks. The Dynamic Canary Randomization technique (DCR) rerandomizes stack canaries at runtime. DCR is applied directly to the binary using a static binary rewriter (i.e., it does not require access to the program's source code). DCR operates with minimal overhead and gives the user the flexibility to specify the conditions under which to rerandomize the canary. DCR is an improvement over existing canary rerandomizers because it allows rerandomization to be applied at any point during execution and at any frequency. We show that DCR improves software security by demonstrating its ability to prevent real-world attacks on well-known software (e.g., *nginx*) "protected" by traditional stack canaries.

CCS Concepts

•**Security and privacy → Operating systems security; Software and application security;** *Intrusion detection systems; Vulnerability management; Software security engineering;* •**Software and its engineering →** *Compilers;*

1. INTRODUCTION

Stack canaries are a well-known, effective technique for detecting stack overflow attacks and *gcc* has supported their use for more than ten years [2]. At function invocation, a secret value (the canary) placed on the stack between the caller's saved return address and the space allocated for the callee's local variables. Before a function returns, the canary value on the stack value is compared to the program's reference canary value. A mismatch indicates that the program wrote beyond the allocated boundaries of a buffer on the stack. Upon detection of such a condition, the program halts. If the canary value was modified by an attacker attempting to hijack control of the target program by overwriting the return address, his/her efforts are thwarted. However, if the value of the reference canary is leaked to the attacker, its protection can become ineffective. The attacker can use the leaked canary value to craft their attack command so that the stack always contains a matching canary value.

For programs compiled using *gcc*, the reference canary value is randomly generated at program invocation and fixed throughout execution. Moreover, for software running on the Linux operating system, canary values are inherited from the parent process and only changed if/when the child process `exec()`s a different program. Researchers and others have exploited these behaviors to craft real-world attacks that bypass the protections of stack canaries. For instance, researchers have discovered that it is possible to infer the canary values for every process on a device running the Android operating system [3]. Researchers have also developed the blind return oriented programming (BROP) technique that bypasses the protection offered by canaries in server software that follows the accept-fork paradigm to process multiple clients simultaneously (e.g., *nginx*, *mysql*) [1].

In this work we introduce a moving-target defense technique called Dynamic Canary Randomization (DCR). DCR is a technique for rerandomizing stack canaries at user-defined points during program execution and can be applied directly to binary programs (i.e., it does not require access to the program's source code). DCR is an improvement over existing canary rerandomizers because it allows rerandomization to be applied at any point during execution and at any frequency. We show that DCR prevents the leaked value of a canary from being used to launch an attack against target software and show that it defends against BROP attacks.

2. DESIGN

Dynamic Canary Randomization is applicable to binary programs that are compiled with *gcc* with stack smashing protection (SSP) (`-fstack-protector`). In some modern Linux distributions, *gcc* defaults to this option. *gcc* SSP inserts code into the function prologue and epilogue. In a function's prologue, code is inserted that copies the reference canary value from thread local storage (TLS) to the program

CISRC '16, April 05-07, 2016, Oak Ridge, TN, USA
© 2016 ACM. ISBN 978-1-4503-3752-6/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2897795.2897803

stack. In a function epilogue, code is inserted that compares the stack's canary to the reference canary value. If these values do not match, the program invokes a handler that terminates the program.

DCR uses a similar mechanism: It inserts code in the function prologue that places canary values on the stack. In the function prologue, it inserts code that verifies that the value has not changed. DCR and *gcc* SSP are so similar that, in fact, DCR is implemented by using our binary rewriter, Zipr, to replace the canary handling instructions emitted by *gcc* with its own handling instructions and reusing the TLS space *gcc* allocates for the reference canary value. Like *gcc* SSP, DCR keeps the reference canary value in TLS. In DCR, like in *gcc* SSP, an executing program has a single reference canary value throughout program execution.

The difference between DCR and *gcc* SSP is the actual value stored in a canary. Because there is one reference canary during program execution, when DCR rerandomizes the canary value it must update all canaries existing on the program stack. If DCR only used the rerandomized canary value on subsequent function invocations, the program would incorrectly detect a canary mismatch when returning from functions invoked prior to rerandomization.

Consider example.c in Figure 1. The figure shows the contents of the stack when it reaches the statement labeled (2). The user directed canary randomization to occur at the statement labeled (1). The figure shows the problem when all canary values on the stack are not rewritten at the rerandomization point. Although the rerandomized reference canary value is placed on the stack at the invocation of `function2()` (label 5), the canary values already on the stack (labels 3, 4) no longer match the reference canary value. In this case, DCR would detect a stack overflow error in the epilogue to `function1()` where one did not exist and incorrectly terminate the program.

To facilitate rewriting every canary value on the stack at runtime, DCR embeds offsets and addresses in the values of canaries on the stack and stores an additional offset alongside the reference canary value in the TLS space. These values are used by DCR to build a linked list of canaries on the stack at a given point during program execution. The offsets stored in a canary value on the stack point to the previous stack canary. A pointer to the head of the list is embedded in the reference canary value. See Figure 2.

In Figure 2, the stack grows from the top of the figure to the bottom. Time *a* is before time *b*; time *b* is before time *c*; etc. The reference canary value is `0x123456`. The numbers in green in the reference canary value are the offsets to the head of the stack canary linked list. When a function is invoked at time *a*, the reference canary value is copied to the stack at address `0x7777f0f00`. The reference value is updated to "point" to that stack location, `0000` in this case. When a function is invoked at time *b*, the reference canary value is copied from TLS. Its offset is updated to the difference between the current stack address and the offset embedded in the reference canary value. The result is pushed on the stack at address `0x7777f0ee8`. The reference canary value's offset is updated to point to the top of the stack, `00018` in this case. On completion of a function, this process is reversed and the canary value is verified. If DCR detects that a canary value is modified, the program terminates.

Using these calculations, the program can retrieve the head of the stack canary linked list at any point during program execution by accessing the reference canary value and use the embedded offsets in the canaries on the stack to iterate through every canary value. At the point where a user-directed canary rerandomization occurs, DCR iterates through the linked list of canary values and rewrites the entries. Because unsafe values may be latent on the stack (from buffer overflows that occurred in previously invoked functions), DCR verifies the canary values as each one is rewritten. If there is a mismatch, the program immediately terminates. This check is also necessary to verify the integrity of the offsets in the stack canary linked list.

3. IMPLEMENTATION

DCR is applicable to binary programs compiled with *gcc* SSP. Other compilers could easily be accommodated. DCR is implemented in conjunction with a static binary rewriter whose actions are driven by a transformation that describes modifications to make on an input binary program. The static rewriter alters the input program according to this transformation specification and generates an updated binary program. The resulting binary program executes (without any additional runtime support) on exactly the same platforms as the original program.

When DCR is applied to a program, the user specifies a particular function whose invocation will trigger the randomization. For instance, the user may specify that rerandomization occur when every `read()` is called. In the future, we will add support for the user that wants to randomize canary values after every *n* instructions or every *s* seconds.

We have tested our implementation of DCR against SPEC2006 and found that it is robust enough to support the programs in that benchmark. We have also tested the implementation against *nginx*, a well known, open-source webserver. See Section 4 for details.

4. EVALUATION

This section describes the process we used to evaluate the performance impact of DCR and its ability to add additional security to vulnerable software. Performance impact is measured using the industry-standard SPEC2006 benchmark suite. Each experiment is a configuration and execution of the entire suite of SPEC2006 benchmark applications.[1] The security aspect of DCR is demonstration by its ability to thwart a real-world attack.

All experiments were performed on a server-class host containing two 2.80GHz Xeon processors, each with 10 cores, and a total of 50GB of RAM. The host runs Ubuntu 14.04 LTS with Kernel version 3.13.0-30. With the exceptions noted below to configure stack smashing protection, all executions of SPEC were performed using the default compilation parameters and optimization at level `-O2`. Each application in the benchmark suite was executed once on reference input.

4.1 Steady-State Performance Overhead

To assess the performance, we compared the *steady-state overhead* incurred by stack canaries implemented by the

[1]Two of the 29 benchmark applications are not included because they did not run on the host platform when compiled and linked with the standard toolchain.

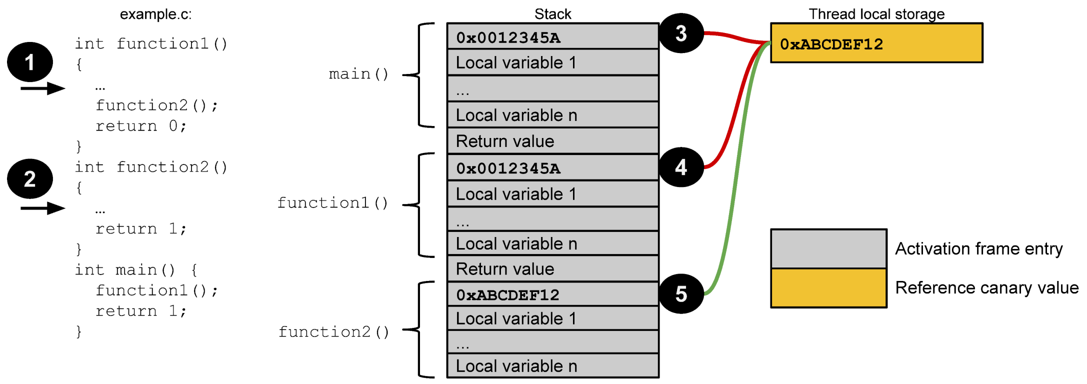

Figure 1: All canary values on the stack must be rewritten when the reference canary value is rewritten.

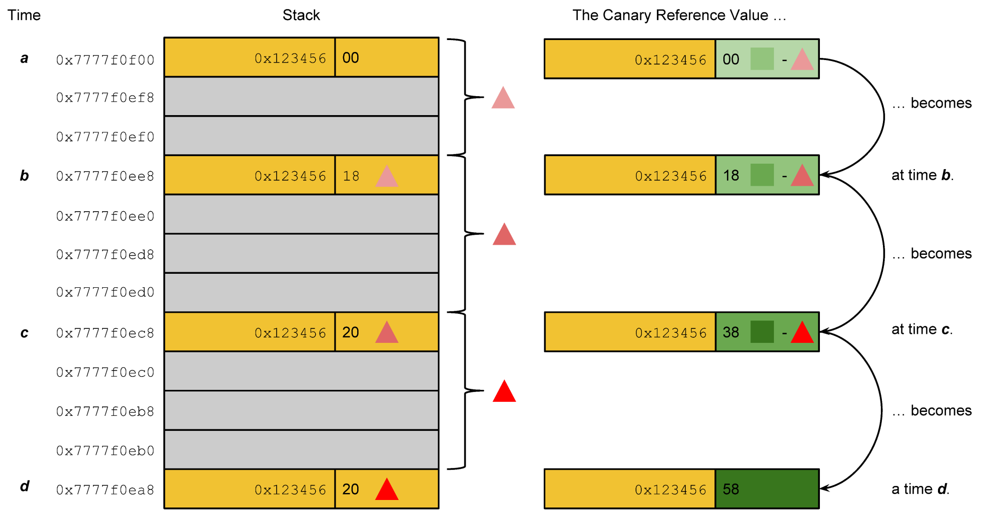

Figure 2: Building the linked list of canary addresses on the stack at runtime.

gcc compiler and DCR. Steady-state overhead is the performance penalty for maintaining canary values at runtime. For *gcc*'s SSP this is the cost of pushing canary values on to the stack in the prologue and checking their values in the epilogue. For DCR, this is the cost of pushing canary values on to the stack in the prologue, checking their values in the epilogue *and* maintaining the linked list. The overhead of rerandomizing the canary values at different intervals is studied in Section 4.2.

As mentioned previously, DCR is implemented using a static binary rewriter. The experimental methodology is designed to isolate the overhead of the static rewriter from the overhead of DCR and contrast that with the overhead introduced by *gcc*'s SSP.

It is possible to enable SSP in *gcc* in several ways. On most modern OS platforms, *gcc* defaults to protecting any function that calls `alloca()` or uses "buffers larger than 8 bytes" [6]. The user can force *gcc* to add SSP to every function no matter the size of the local buffers or the means of memory allocation. We refer to the former as *selective SSP* and the latter as *complete SSP*.

To perform the assessment, we performed six experiments.

Experiment 1 Execution without SSP: An execution of SPEC2006 with SSP disabled (`-fno-stack-protector`). The baseline for Experiments (2) and (3).

Experiment 2 Execution with selective SSP: An execution of SPEC2006 where *gcc* determines the functions to protect (`-fstack-protector`) [6].

Experiment 3 Execution with complete SSP: An execution of SPEC2006 where *gcc* is configured to protect all functions (`-fstack-protector-all`).

Experiment 4 Execution with static rewriter, no transformations with selective SSP: An execution of SPEC2006 where the application binaries are compiled with selective SSP and rewritten with with the static rewriter. Comparing these results to the results of Experiment (2) isolate the overhead of the static rewriter.

Experiment 5 Execution with DCR and selective SSP: An execution of SPEC2006 where the application binaries are rewritten to add DCR. The application binaries themselves are compiled with selective SSP. Comparing these results with Experiments (2) and (4) isolate the steady-state overhead of DCR when SSP is selectively enabled.

Experiment 6 Execution with DCR and complete SSP: An execution of SPEC2006 where the application binaries are rewritten to add DCR. The application binaries themselves are compiled with selective SSP. Comparing these results with Experiments (3) and (4) will isolate the steady state overhead of DCR when SSP is fully enabled.

Experiments (1), (2) and (3) show the overhead of *gcc*'s SSP. Experiment (4) isolates the overhead of the static rewriter. Experiments (5) and (6) show the steady-state overhead of DCR. Results are shown in Figures 3 and 4.

On average, *gcc*'s selective SSP adds negligible (.38%) overhead while *gcc*'s full SSP adds minimal (2.89%) overhead. On average, the static rewriter incurs 24% overhead while DCR's selective and full SSP adds .045% and 13.38%, respectively. While a 24% overhead may seem high, stack protection applied directly to the binary offers several benefits: 1) no source code is required, 2) usable on programs written in different programming language binaries and 3) no changes to fragile and complicated build processes.

4.2 Case Study: Rerandomizing Canaries in bzip2

Besides steady-state performance overhead, DCR introduces runtime overhead every time that the canary value is rerandomized. The more often canaries are randomized, the more overhead imposed by DCR. On the other hand, the more often canaries are rerandomized, the less time is available for an attacker to take advantage of any data leaks from the application. This security versus performance tradeoff is fundamental to the DCR method.

In this section, we study DCR's overhead when protecting a particular Internet-facing web application. We plan on studying the most effective way to balance the performance overhead of DCR with the additional security it provides in the future. See Section 6.

Consider a cloud infrastructure provider whose web-based configuration interface allows the user to upload compressed configuration files. Those files are uncompressed and then parsed and used to configure a virtual host. Assume that the configuration files are compressed and decompressed with a daemonized version *bzip2* that is compiled with SSP. This version of *bzip2* accepts input and commands (i.e., whether to compress or decompress) over a socket and writes output back to the user over the same connection. If there is a stack overflow vulnerability in *bzip2* that leaks the stack canary values, then an intruder could build a specially crafted compressed file to a) retrieve the stack canary value and b) launch an attack on the cloud provider.

For such an attack to succeed, the attacker relies on the fact that, once learned, stack canary values remain consistent throughout program execution. Applying DCR to the *bzip2* program that decompresses the configuration file would prevent such an attack.

Recall that the rate at which *bzip2* rerandomizes its stack canary values determines the period of time that an intruder can use a leaked canary. As the rerandomization rate increases, the period of validity decreases, and vice versa. Again, managing this tradeoff is important and something that we plan on studying in detail in future work (see Section 6).

To secure this hypothetical web application from this particular threat vector, we decided to configure DCR to rerandomize canary values every time that *bzip2* invokes `memcpy()`. To quantify the overhead associated with this choice, we used the version of *bzip2* from SPEC2006.

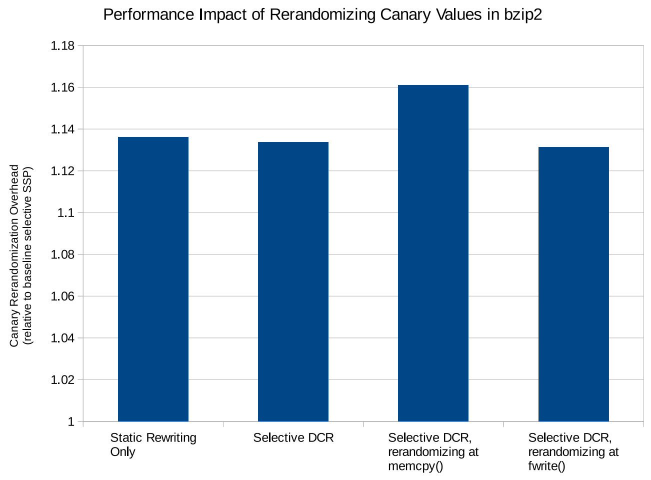

Figure 5: Performance impact of rerandomizing canary values at every `memcpy()` and `fwrite()` during an execution of *bzip2* under SPEC2006.

For the reference test input set to *bzip2* in SPEC2006, there are 1704322 invocations of `memcpy()`. The number of canary values updated at each randomization point depends on a) the depth of the stack of the running program at that time and b) the number of stack frames that contain a canary. By default for the version of the compiler used to generate *bzip2* in our experiment, only certain functions are protected with stack canaries [6]. In other words, DCR may not rewrite canary values at every rerandomization point. In our experiment, there were 1704286 canary values updated, for an average of 0.9999 canary updates per randomization.

`memcpy()` is obviously not the only channel for an information leak from the vulnerable *bzip2* application. Since the adversary could exfiltrate data written to standard output or the filesystem, we chose to rerandomize at every `fwrite()` call and compare the results. For the reference test input set to *bzip2* in SPEC2006, there are 29826 invocations of `fwrite()` and there are 10070 canary values updated which yields 2.96 canary updates per randomization.

In order to isolate the performance overhead of canary rerandomization in DCR, we performed four different executions of the SPEC2006 version of *bzip2*. In the first execution, we measured the speed of the *bzip2* without any modifications. In the second execution, we measured the speed of *bzip2* compiled with selective SSP after it had been statically rewritten (equivalent to Experiment 4 in 4.1). In the third execution, we measured the speed of *bzip2* after it had been statically rewritten and modified to include a configuration of DCR that rewrites canary values at every invocation of `memcpy()`. In the final execution, we measured the speed of *bzip2* after it had been statically rewritten and modified to include a configuration of DCR that rewrites canary values at every invocation of `fwrite()`.

Figure 5 shows the results of these four experiments. The overhead of DCR rerandomizing canary values at every `memcpy()` is 16.08%, with 13.33% of that overhead due to the

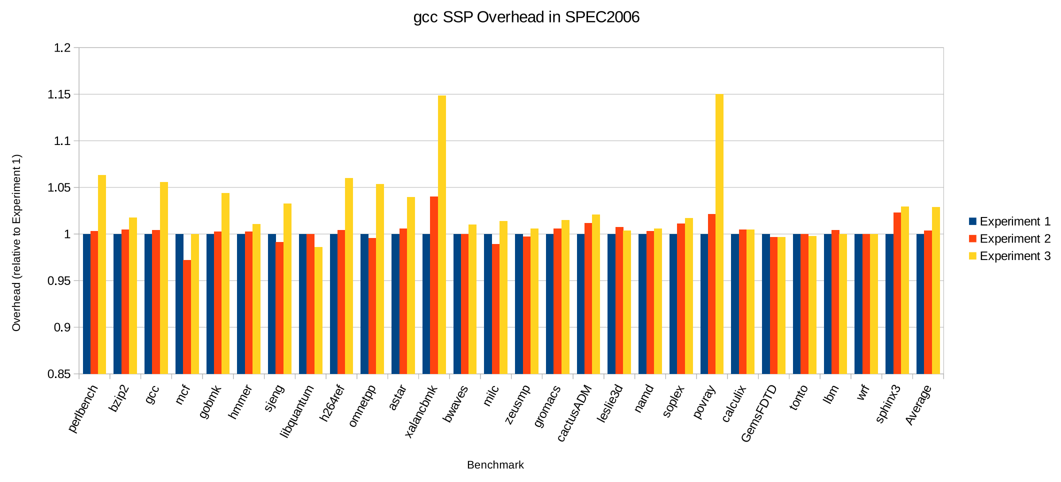

Figure 3: Overhead of *gcc*'s SSP in SPEC2006 benchmark. Experiment (1) is the baseline (no stack protection). Experiment (2) is selective SSP, and Experiment (3) is full SSP.

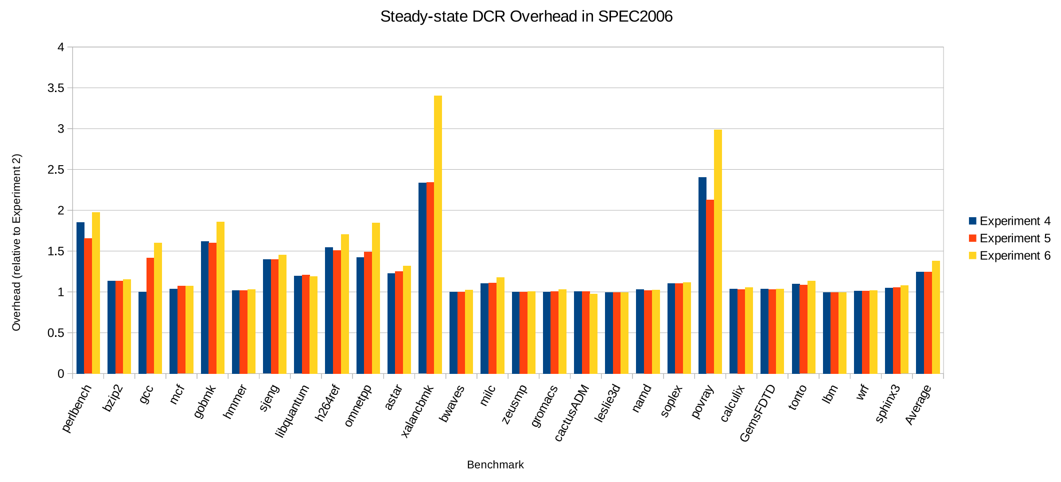

Figure 4: Overhead of DCR's steady-state performance in SPEC2006 benchmark. Experiment (4) is static rewriting only. Experiment 5 is selective DCR applied with static rewriting. Experiment (6) is full DCR applied with static rewriting.

static rewriter itself. By comparison, the overhead of DCR rerandomizing canary values at every `fwrite()` is negligible relative to the overhead of the static rewriter itself. Although there are more updates per randomization in the latter case, there are significantly fewer total randomizations performed when compared to the number of randomizations performed in the former. This accounts for the differences in overhead in the two scenarios. This is a concrete case where the choice of how often to rerandomize affects performance. We plan on studying the tradeoffs between performance and frequency of rerandomization in the general case in future work.

4.3 DCR and BROP

Return oriented programming (ROP) is a new and dangerous form of software attack. In ROP attacks, the attacker finds a list of "gadgets" that he/she can chain together to perform a nefarious task. The attacker places the chain of gadgets on the stack and then transfers program control to that chain. A ROP attack is particularly dangerous because it defeats the protection afforded by a non-executable stack (NX). A complete description of ROP is beyond the scope of this paper. See Shacham for a complete description of ROP [9].

To launch a ROP attack, the attacker must be able to find the addresses of gadgets in the target program. If the attacker cannot examine the object code of a process offline (if, for instance, the target is a server running on a remote host), then the attacker's task of locating gadgets becomes very difficult. The blind ROP (BROP) attack, presented in 2014 by Bittau et al., is designed to find and exploit gadgets in just this scenario [1]. Their attack carefully probes a remote target until it is able to find a set of gadgets that it can use to download a copy of the program's instructions. The attack proceeds by, first, finding a gadget that will invoke the write system call. Then, it finds gadgets that will control the parameters to that function. Finally, it invokes the write gadget repeatedly to download the entire contents of the program. After that, the attacker can analyze the program's instructions offline, find the locations of other gadgets and inflict further damage using a traditional ROP attack.

The BROP attack relies on several very important assumptions. The authors rely on the fact that the remote target software automatically restarts after a crash and does not re-randomize its address space between executions. This assumption holds for servers that spawn separate child processes to handle clients (the accept-fork paradigm for handling multiple clients simultaneously). In such a framework, the address space of the processes that handle client requests are not re-randomized, even if address space layout randomization (ASLR) is enabled. In other words, the information gained with every attack probe is valid on subsequent attack attempts.

To demonstrate that these assumptions are not unreasonable, consider that *nginx* meets these criteria. *nginx* is a web server that, as of September 2015, powers more than 15% of the top 1 million busiest Internet sites [7].

To summarize, BROP attacks are widely applicable and defeat NX and ASLR. However, their deployment still requires a way of controlling the target. To do this, Bittau et al.'s BROP attack uses the traditional stack overflow. Recall, however, that stack canaries explicitly detect this type of attack. Therefore, the value of the stack canary must be leaked for BROP attacks to succeed.

The authors use byte-for-byte stack reading to leak the canary values from their remote target. First introduced by Zabrocki [10], byte-for-byte stack reading is an iterative process: it overwrites the stack byte-by-byte and observes program behavior. If the program crashes when byte b is written to the stack at position a, then something about the stack is wrong. However, if the program does not crash, it is reasonable to assume that the target program's stack does hold b at a. On the next probe, the attack will maintain the previous byte in it's position and overwrite stack position $a + 1$ until it determines the value b'. Because the values of the canary are static during the course of execution, the attacker is eventually able to learn the canary values and defeat their protections.

The developers of the BROP attack call this generalized stack reading (GSR). It is fundamental to the attack but easy to prevent with DCR. To demonstrate the effectiveness of DCR at thwarting BROP attacks, we started by reproducing the author's results. The authors developed an automated tool, braille, to perform BROP attacks. They used braille to attack *nginx*, *mysql* and *ali*[2] We successfully replicated their attacks on *nginx* and *ali* using braille[3]. For an additional data point, we wrote a program with a seeded buffer overflow vulnerability, *nslr*, compiled it with *gcc* and SSP and attacked it with braille. Their automated attack succeeded against *nslr*.

We then applied DCR to each of these vulnerable binary programs. We specified rerandomization at every `fork()`.

[2] *ali* is a custom server written by an independent researcher at the author's university. In an attempt to simulate attacking a completely "closed" server over the network, Bittau et al. were not given offline access to the program source code *or* binary.

[3] Bittau et al. did not provide enough information in their paper to recreate their tests on *mysql*.

Program	*gcc* and SSP	*gcc* and DCR
nginx	V	P
ali	V	P
nslr	V	P

V: vulnerable, P: protected

Table 1: Results of BROP attacks against software protected with traditional canaries (SSP) and with Dynamic Canary Randomization (DCR)

We then attacked the modified programs using the same attack tool. After applying DCR, the attacks were unsuccessful in every case. See Table 1.

5. RELATED WORK

Gisbert et al. have also implemented canary value randomization [4]. Their technique, known as renew-after-fork stack smashing protection (SSP) or RenewSSP, randomizes stack canary values every time the target software invokes `fork()`. They implemented RenewSSP by preloading a library that contains an alternate version of `fork()`. Their implementation choice makes RenewSSP inapplicable to statically linked programs and it restricts the user to rerandomizing the stack only when a process invokes `fork()`. DCR is applicable to statically and dynamically linked programs and it allows the user to specify arbitrary places for reference canary rerandomization.

In addition to implementing a type of canary value randomization, Gisbert et al. analyze mathematically the effectiveness of stack canary value randomization [5]. Their results are encouraging and reinforce our empirical findings that canary rerandomization improves software security.

More recently, Petsios et al. proposed and implemented DynaGuard, a system similar to DCR [8]. Like DCR, DynaGuard is implemented as a linked list of stack canaries maintained during execution and imposes no restrictions on where those values can be rerandomized. Unlike DCR, however, DynaGuard adds canary randomization support in one of two ways: during compilation with a compiler extension (for cases where source code is available) or at runtime using PIN for dynamic instrumentation (for cases where source code is unavailable). DynaGuard incurs 1.2% overhead when deployed on source code and 70% overhead when added to software through PIN. One of the advantages of DCR's use of static binary rewriter is that rerandomization points can be chosen per application, while a compiler-based implementation does not offer that flexibility. We cannot contrast or compare DCR and DynaGuard with respect to overhead associated with the choice of the frequency of rerandomization in applications — they do not delineate the costs of rerandomization from steady-state overhead in their results.

Finally, DynaGuard and DCR differ in their means of maintaining the linked list of canary locations at runtime. DynaGuard allocates a canary address buffer to store the linked list whereas DCR utilizes the existing canary reference storage space. Overall, their results are further validation of our belief that canary randomization is an important tool for the improvement of software security.

6. FUTURE WORK

In the future, we plan to optimize the runtime efficiency of DCR and to study the tradeoffs between randomization frequency and performance. Because DCR relies on a static rewriter for implementation, it is important that we investigate performance improvements for the static rewriter itself in addition to the DCR technique *per se*.

We will continue to add additional features to the DCR implementation. For instance, we plan on adding support for the user who wants to randomize canary values after every n instructions or every s seconds.

We also plan on understanding the other threat vectors where DCR would improve the security of software. We know already that DCR is applicable to multi-threaded programs but it is important to understand how this technique can be applied to single threaded, single process programs that coordinate handling multiple tasks simultaneously through a workqueue, a technique used by servers like ntpd and olsrd.

7. CONCLUSION

Stack canaries are a well-known and effective technique for detecting and defeating stack overflow attacks. However, they are not perfect. In this work we introduced Dynamic Canary Randomization (DCR), a technique for rerandomizing stack canaries that is applicable to binary programs (i.e., it does not require access to the program's source code). We showed that DCR operates with minimal overhead and gives the user the flexibility to specify the conditions under which to rerandomize the canary. We compared DCR to existing canary rerandomizers and showed how it improves on the state of the art. Most importantly, we showed that DCR improves software security by demonstrating its ability to prevent real-world attacks on well-known software (e.g., *nginx*) "protected" by traditional stack canaries.

8. REFERENCES

[1] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. *2014 IEEE Symposium on Security and Privacy*, pages 227–242, 2014.

[2] 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 *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[3] D. Kaplan, S. Kedmi, R. Hay, and A. Dayan. Attacking the linux prng on android: Weaknesses in seeding of entropic pools and low boot-time entropy. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, San Diego, CA, Aug. 2014. USENIX Association.

[4] H. Marco-Gisbert and I. Ripoll. Preventing brute force attacks against stack canary protection on networking servers. In *12th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 243–250, Aug 2013.

[5] H. Marco-Gisbert and I. Ripoll. On the effectiveness of NX, SSP, RenewSSP, and ASLR against stack buffer overflows. *2013 IEEE 12th International Symposium on Network Computing and Applications*, pages 145–152, 2014.

[6] n/a. Instrumentation options - using the gnu compiler collection (gcc).

[7] Netcraft. September 2015 web server survey, Sep 2015.

[8] T. Petsios, V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. Dynaguard: Armoring canary-based protections against brute-force attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 351–360, New York, NY, USA, 2015. ACM.

[9] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[10] A. Zabrocki. Scraps of notes on remote stack overflow exploitation, Nov 2010.