Smashing the Stack Protector for Fun and Profit

Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, Apostolis Zarras

Smashing the Stack Protector for Fun and Profit

Bruno Bierbaumer[1] (✉), Julian Kirsch[1], Thomas Kittel[1], Aurélien Francillon[2], and Apostolis Zarras[3]

[1] Technical University of Munich, Munich, Germany
bierbaumer@sec.in.tum.de
[2] EURECOM, Sophia Antipolis, France
[3] Maastricht University, Maastricht, Netherlands

Abstract. Software exploitation has been proven to be a lucrative business for cybercriminals. Unfortunately, protecting software against attacks is a long-lasting endeavor that is still under active research. However, certain software-hardening schemes are already incorporated into current compilers and are actively used to make software exploitation a complicated procedure for the adversaries. Stack canaries are such a protection mechanism. Stack canaries aim to prevent control flow hijack by detecting corruption of a specific value on the program's stack. Careful design and implementation of this conceptually straightforward mechanism is crucial to defeat stack-based control flow detours. In this paper, we examine 17 different stack canary implementations across multiple versions of the most popular Operating Systems running on various architectures. We systematically compare critical implementation details and introduce one new generic attack vector which allows bypassing stack canaries on current Linux systems running up-to-date multi-threaded software altogether. We release an open-source framework (*CookieCrumbler*) that identifies the characteristics of stack canaries on any platform it is compiled on and we propose mitigation techniques against stack-based attacks. Although stack canaries may appear obsolete, we show that when they are used correctly, they can prevent intrusions which even the more sophisticated solutions may potentially fail to block.

1 Introduction

Buffer overflow vulnerabilities are as old as the Internet itself. In 1988, the *Morris Worm* was one of the first malware discovered in public that leveraged this vulnerability [1]. Since then, many security breaches can be linked to successful exploitation of buffer overflows, which denotes that the problem is far from being solved. As a matter of fact, the *Mitre Corporation* lists more than eight thousand *Common Vulnerabilities and Exposures* (CVE) entries that contain the keyword "buffer overflow"[4]. A significant portion of these vulnerabilities is comprised by the so-called stack-based buffer overflow bugs [2]. This is due to the application's stack inherent property of mixing user-controlled program data together with

[4] https://cve.mitre.org/cve/search_cve_list.html

control flow related data; thus allowing an attacker to overwrite control flow related parts of the stack.

Due to this well-known weakness, Cowan et al. [3] propose a technique named *Stack Smashing Protection* (SSP). The idea behind SSP is to detect stack-based control flow hijacking attempts by introducing random values (so-called *canaries*) to the stack that serve as a *barrier* between attacker-controlled data and control flow relevant structures. After a function finishes executing, a canary—named after the coal miner's canaries which were used to detect presence of gas—is checked against a known "good" value stored in a safe location. Only if the canary maintains its original value, execution continues. This mitigation technique has been present in compilers for more than 10 years and is now a counter measure supported by major compilers [4–6].

Recently, more advanced techniques have been proposed to prevent buffer overflow attacks including *Code Pointer Integrity* (CPI) [7, 8] and *Control Flow Integrity* (CFI) [9]. Both ideas encircle the concept of protecting the control flow from being hijacked. These advanced techniques have created the illusion that stack canaries are nowadays obsolete. However, both techniques consider non-control-flow diverting attacks to be out of scope. As we discuss later, this is an underestimated attack that can be successfully countered by stack canaries [10]. While introduced almost twenty years ago, stack canaries are still one of the most widely deployed defense mechanisms to date [11] and are, as we will show, a necessary complement to other more recent modern buffer overflow mitigation mechanisms. As a matter of fact, all modern compilers support stack canaries.

In this paper, we show that stack canaries, even in combination with more advanced techniques, are not a silver bullet. We find that due to inconsiderate implementation decisions, stack canaries *themselves* are vulnerable to buffer overflow attacks; ironically the same type of attack that they are supposed to protect against. To demonstrate this, we first implement a framework (*CookieCrumbler*) which is able to identify the characteristics of stack canaries on various modern operating systems, independently from the CPU architecture used. We then run *CookieCrumbler* on 17 different combinations of *Operating Systems* (OSes), *C* standard libraries, and hardware architectures. We run *CookieCrumbler* against both seemingly old, but still widely used and supported, OSes as well as the most recent versions. The extracted *CookieCrumbler* results enable us to introduce a new attack based on the observation that the canary reference values are not always stored at a safe location. This allows an attacker to overwrite the control-flow relevant data structures and the current reference value at the same time and thereby reliably bypass SSP.

In summary, we make the following main contributions:

- We propose *CookieCrumbler*, a framework to automate the identification of the characteristics of SSP implementations.
- We evaluate *CookieCrumbler* against state of the art operating systems and libraries, and discover weaknesses in multiple SSP implementations.
- We introduce a novel attack vector to exploit those vulnerabilities.
- We propose mitigations techniques to harden SSP implementations.

2 Background and Related Work

2.1 Stack Smashing Protection

The idea to guard certain parts of the executable's stack dates back to 1998 [3,12]. The concept is to protect control flow related information on the stack using a so-called *stack canary* or *stack cookie*: a random value placed between the user-controllable data and the return pointers on the stack during stack setup phase in the function prologue. The mechanism is implemented synchronously with the control flow: after function execution, once the control flow returns to the caller, the cookie value is checked against a known "good" value. Only if there is a match between the two values, the stack frame is cleaned up and the control flow is allowed to return to the caller. Several years were needed for StackGuard to be integrated in the mainline GCC distribution [13].

Attackers may try to evade StackGuard by embedding the canary in the data used during the overflow (i.e., canary forgery). Cowan et al. [7] propose two methods to prevent such a forgery: *terminator* and *random canaries*. In 32-bit operating systems, a terminator canary is usually constructed using of the char representation of `NULL`, `CR`, `LF`, and `EOF` (`0x000d0aff`). This is because overflows often exploit unsafe string manipulation functions: as the terminator canary includes characters which are used to terminate strings, it is impossible to directly include them in a string without terminating the string operation. However, not all buffer overflows are due to unsafe string manipulation operations (e.g., `read()`) and the fixed terminator canary does not provide any protection in those cases. On the other hand, random canaries cannot be guessed by an attacker and is therefore the most generic approach.

Marco-Gisbert and Ripoll extend the original StackGuard concept by proposing a renewal of the secret stack canary during the `fork` and `clone` system calls [14]. This way, an external attacker is not able to brute-force the stack canary in scenarios where the request handling routine is forked from a server application for each request, as is typically the case for network facing applications. As an alternative, Kuznetsov et al. [8] propose secure code pointers by storing them in a safe memory region. In their work, they assume that the location of the safe region can be hidden.

In essence, in order to be effective, StackGuard relies on the following assumptions:

❶ The cookie value placed on the stack (Can) must be *unknown* to the attacker.
❷ The known *good* value (Ref) is placed at a location in memory that is distinct from the location of Can and ideally mapped read-only.
❸ If a stack cookie value (Can) is corrupted, the program execution terminates immediately without accessing any attacker controlled data.
❹ The overflow is contiguous, starting from a buffer on the stack, and therefore does not permit an attacker to skip certain bytes in memory.

The main focus of this paper is to falsify Assumptions ❶, ❷, and ❸ using contiguous overflows (i.e., adhering to Assumption ❹).

2.2 Function Pointer Protection

Function pointers stored in writable memory at static addresses are common targets to gain control of a vulnerable program's execution. To defend against this threat, [7] introduces code pointer protection. *PointGuard* [15] is the first mechanism capable of encrypting code pointers in memory. For each process, PointGuard generates a random key during process creation. Each pointer in memory is then scrambled in memory by performing a bijective operation on the pointer using the process's specific random key.

Glibc implements this protection mechanism since 2005 [16] by using the `PTR_[DE]MANGLE` macros. On the other hand, the Windows run-time, provides similar functionality with the `Rtl[En|De]codePointer` API call since *XP SP2* (2004). Both implementations use very similar algorithms to encipher pointers: a logical bit rotation combined with an XOR ($\oplus$) involving the per-process random secret (RAND). For instance, the 64-bit Windows run-time implements the following two equations for pointer protection:

$$\text{PTR}_{\text{ENC}} = \text{ror64}(\text{PTR}_{\text{ORIG}} \oplus \text{RAND}, \text{RAND}) \tag{1}$$

$$\text{PTR}_{\text{ORIG}} = \text{rol64}(\text{PTR}_{\text{ENC}}, \text{RAND}) \oplus \text{RAND} \tag{2}$$

On Linux (with *glibc*), the situation is very similar, except that a constant is used as the number of digits to rotate (`0x11` is actually $2 \cdot$ `sizeof(void *)` $+ 1$):

$$\text{PTR}_{\text{ENC}} = \text{ror64}(\text{PTR}_{\text{ORIG}} \oplus \text{RAND}, \texttt{0x11}) \tag{3}$$

$$\text{PTR}_{\text{ORIG}} = \text{rol64}(\text{PTR}_{\text{ENC}}, \texttt{0x11}) \oplus \text{RAND} \tag{4}$$

Most notably, the main difference in the two implementations is that on Windows the `Rtl[En|De]CodePointer` retrieves the value RAND from the kernel, whereas *glibc* on Linux stores the pointer guard in user space in the *Thread Control Block* (TCB). Cowan et al. [15] state that the PointGuard key has to be stored on its own page once it is initialized to protect the key against information leakage. As can be seen, this assumption is not met by all implementations.

2.3 Attacks Against Stack Canaries

An adversary may attempt to attack the stack canary mechanism itself in order to successfully exploit a program. Strackx et al. [17] analyze the security promises made by randomization based buffer overflow mitigation systems, such as the ones described above. They conclude that a vulnerable program offering both a buffer overread and a buffer overflow can be easily attacked. However, their work misses the experimental evaluation of the success rate of such an attack. Ding et al. [18] reveal weaknesses in the StackGuard implementation used in Android 4.0: the source of randomness used for the stack canaries is only initialized once at OS boot and then used for every application on the system. In addition, the created canary is predictable as the state used to initialize the canary only depends on randomness available at kernel boot-up.

Dynamic Canary Randomization [19] attempts to defend attacks targeting stack canaries. This technique re-randomizes all active stack canaries during run-time so the attackers cannot reuse the knowledge they gained while leaking memory from an earlier execution of the attacked process. While this approach might help against attacks that read the canary and then use the gained knowledge in a separate step, it is ineffective against the attack introduced in this work.

2.4 Thread Control Block

Modern OSes contain a dedicated data structure, called TCB, to store information about the environment that the current thread is executing in. The data stored in the TCB varies depending on OSes and thread library implementations. For instance, on Windows this data structure is named `ThreadInformationtBlock` and contains information about a thread's *Structured Exception Handling* (SEH) chain, its associated *Process Control Block* (PCB), and a pointer to *Thread Local Storage* (TLS). The TCB is accessed either using a library function or a designated register that improves speed. For example, *glibc* on Linux *x86_64* uses the `fs` register as the base address of the TCB. Intel provides *Model Specific Register*s (MSRs) to override `fs` and `gs` segment base addresses, effectively enabling 64-bit OSes to access the TCB in a fast way. This is achieved by prefixing any load or store operation with the `fs` segment register.

Both SSP (with StackGuard) and Function Pointer Protection (with PointGuard) belong to the standard set of defense mechanisms and are highly adopted in practice. However, both mechanisms require the storage of their respective random reference keys (Ref). This is where the TCB becomes relevant, in context of our work: In some versions of the compiler/standard library, it is the TCB that contains the reference keys for both mechanisms. Therefore, both mechanisms can be attacked if the data contained in the TCB can be overwritten, as shown in this paper.

2.5 Modern Defense Mechanisms

In this study, we explicitly concentrate on concrete implementations rather than theoretic contributions. We therefore focus on defense mechanisms that are (*i*) available in current (2018) compilers, (*ii*) production ready, and (*iii*) deployed in current operating systems. This leaves us with a very narrow set of mechanisms. In fact, we rarely find academic publications implementations that reach a mature state PointGuard [15], and StackGuard [12].

There is a trend on maintaining the integrity of an application's control flow at runtime. CFI is achieved by ensuring the integrity of forward and backward edges in the control flow graph. As we focus on stack based exploitation techniques targeting control flow information related to the backward edge, we only consider the backward edge validation relevant. Backward edge validation is typically done using a shadow stack [9]. One production ready implementation is SafeStack [8], which we inspect in Section 6 to understand its relationship to stack canaries.

```
Algorithm collect_emp_data()
    Data: Implicitly: Software architecture of the target system
    Result: Data rows for main- and sub-thread
    main ← measure()
    sub ← run_thread(measure())
    return (main, sub)
Procedure measure()
    loc ← allocate_stack(128)
    tls ← allocate_thread(128)
    glo ← allocate_global(128)
    dyn ← allocate_dynamic(128)
    Δ_LOC ← memory_location(REF) - loc
    Δ_TLS ← memory_location(REF) - tls
    Δ_CLO ← memory_location(REF) - glo
    Δ_DYN ← memory_location(REF) - dyn
    return ( (Δ_LOC, W(Δ_LOC)), (Δ_TLS, W(Δ_TLS)),
             (Δ_CLO, W(Δ_CLO)), (Δ_DYN, W(Δ_DYN)) )
```

Algorithm 1: Algorithm used to measure empirical features.

3 Dissecting Implementation Choices

In this section we define five qualitative and five empiric features which we use to systematically evaluate choices made in canaries implementation.

3.1 Qualitative Features

We identify key features of stack canaries by studying the source code of their implementations—if available—or reverse engineering the functionality in their binary format. As required by Assumption ❶ (*unknown* REF) we investigate the origin of the randomness of the reference canary values. The re-randomization of REF is expected to occur at two points during program execution: (*a*) when a process is duplicated using the `fork` system call on UNIX and (*b*) when a new thread (and hence a new stack) is being created. Similarly, CAN could take different values while a particular thread executes different functions and allocates distinct local stack frames. Information that might be encoded into function local values of CAN might include (*i*) REF, (*ii*) the guarded stack contents or some distinct identifier of the function context, and (*iii*) the thread ID.

Assumption ❸ (immediate termination) is another claim that can only be verified in a qualitative manner. To find the quantity of code executed after the canary corruption is detected, we introduce the notion of *noisiness* of the failure handler. To estimate the NOISE level, we count function invocations that are triggered from the point where execution enters the cookie verification failure handler until the point where the application terminated. We also manually check the number of variables that are read from the corrupted memory region (e.g. the stack), and whether the handler executes in user or kernel mode, which we denote by *Current Privilege Level* (CPL).

3.2 Empirical Features

To reason about potential attack targets, we retrieve basic information about the application's memory layout. For each OS and C library pair, we run a test

program which follows Algorithm 1. The program measures the distance (in terms of their addresses) between each user-controllable types of memory. This distance measurement is an important information. Indeed, the closer the REF value is from a user controllable memory the easier it will be for an attacker to overwrite this reference value, and therefore to be able to corrupt the canary without being detected.

More precisely we measure spatial distances (Δ) between the reference value (REF) and:

1. Δ_{LOC}: a variable allocated on the stack of the function.
2. Δ_{TLS}: a variable allocated in *Thread Local Storage* (TLS).
3. Δ_{GLO}: a global variable allocated in statically allocated memory.
4. Δ_{DYN}: a variable allocated in dynamically allocated memory.

We then compute the range of non-contiguous bytes in Δ_{X}. If this range is not mapped as a contiguous writable memory, an overflow from this variable will trigger a page fault before reaching REF.

5. $\text{W}(\Delta_{\text{X}})$: number of contiguously mapped writable bytes in Δ_{X}.

3.3 CookieCrumbler

We implemented the *CookieCrumbler* framework to evaluate those features. From a high-level perspective, *CookieCrumbler* is a direct implementation of Algorithm 1 in C. When compiled and executed on a system, *CookieCrumbler* will thoroughly analyze the implementation of stack canaries. For this purpose, semantic knowledge about the exact location of REF has to be added to the program. For instance, on x86_64, REF is located within the TCB at offset `0x28`. We include this information for all the environments presented in Section 4.

The core of Algorithm 1 is to retrieve the deltas Δ_{LOC}, Δ_{GLO}, Δ_{DYN}, and Δ_{TLS}. To obtain the respective reference point in memory, we use (*i*) a stack local variable, (*ii*) a variable with the `static` keyword, (*iii*) the pointer value returned by `malloc`, and (*iv*) a variable with the `__thread` keyword (on UNIX) or `__declspec(thread)` (on Windows). Threads are created by calls to the functions `pthread_create` (on UNIX) or `CreateThread` (on Windows). To determine $\text{W}(\Delta_{\text{X}})$, we use signal handling on UNIX (catching `SIGSEGV` on a contiguous byte-by-byte write) and the function `IsBadWritePtr` on Windows.

After successful execution, *CookieCrumbler* generates a set of memory locations, deltas, and number of writable bytes for the main- and the sub-threads of a threaded application, respectively. A thorough analysis of these results can reveal potential vulnerabilities in the implementation of stack canaries. The source code and the measured data can be found online.[5]

4 Smashing the Stack Protector

We run *CookieCrumbler* on various OSes with different C standard libraries. Apart from up-to-date version of the C runtime libraries, we also run *CookieCrumbler*

[5] `https://bierbaumer.net/security/cookie/`

on older `libc` versions that are currently still distributed in the stable branches of commonly used Linux distributions. For more details refer to Table 1.

4.1 Qualitative Results

Surprisingly, the qualitative features we examined look very homogenous. We therefore first explain the most common observations and then discuss special cases. Unexpectedly, we found that almost none of the tested implementations changes CAN across different function invocations within the context of one given thread. The only exception to this rule constitutes the Windows family of operating systems, for which CAN is chosen as REF $\oplus$ rbp when the `rbp` register is used as stack frame pointer and CAN = REF $\oplus$ rsp otherwise.

As indicated by the literature [14] we also observed REF (and consequently CAN for all stack frames) to remain static across `fork` invocations on all UNIX operating systems. On this particular point, comparison with Windows is impossible as the `fork` system call is not supported by Windows operating systems family.

In nearly all cases the failure handler executes in user space (the privilege level CPL is 3). The only exceptions to this rule are Windows 8 and newer, which implement the special interrupt number `0x29` (`_KiRaiseSecurityCheckFailure`) for this purpose. When this interrupt handler is called, the program is terminated without accessing any of the potentially corrupted memory in user-space. Windows can fall back to the old user-space failure routine if a call to `IsProcessorFeaturePresent(PF_FASTFAIL_AVAILABLE)` returns zero.

On Windows OS versions newer than 7, the NOISE level is the lowest, as they support an interrupt specifically designed for this purpose. Older versions call 8 functions in `kernel32.dll` and collect information about the current register state before terminating (`TerminateProcess`) the application with return code `0xc0000409` (Security check failure or stack buffer overrun). OpenBSD, when detecting a corrupt stack canary, infers the program's name from a (safe) location in the global variable section of the currently loaded standard library and prints one line of information into the system log. Linux's C standard libraries implement `__stack_chk_fail` in different ways: *musl libc* does not provide any output and terminates execution using a `hlt` instruction, accounting for a minimal NOISE level. *diet libc* prints a static error message and terminates the program with an `exit` syscall. *Bionic* logs a static message, which required to allocate dynamic memory, and finally terminates the program via a `SIGABRT`. The NOISE level culminates on Linux with *glibc* prior to version `2.26`: where we measured that the `__stack_chk_fail` function performs as many as 69 calls to other functions, dispatching at least three calls using (PointGuard protected) writable global static function pointers to create a stack trace by unwinding the attacker controlled stack before exiting the process. More importantly, *glibc* prints the program name fetched from the `argv` array on the stack, which is a potentially attacker-controlled location creating an arbitrary memory leak primitive. This behavior (assigned CVE-2010-3192) was finally fixed in *glibc* version `2.26` in August 2017.

Table 1. Summary of problems found with *CookieCrumbler*.

	Operating System	Architecture	C Standard Library	LOC main	LOC sub	TLS main	TLS sub	GLO main	GLO sub	DYN main	DYN sub
1	Android 7.0	ARM	Bionic	✓	✓	✓	✓	✓	✓	✓	✓
2	Android 7.0	x86_64	Bionic	✓	✗	✓	✓	✓	✓	✓	✓
3	macOS 10.12.1	x86_64	libSystem.dylib	✓	✓	✓	✓	✓	✓	✓	✓
4	FreeBSD 11.00	x86_64	libc.so.7	✓	✓	✓	✓	✘	✘	✓	✓
5	OpenBSD 6.0	x86_64	libc.so.88.0	✓	✓	-	-	✓	✓	✓	✓
6	Windows 10	x86	msvcr1400.dll	✓	✓	✓	✓	✘	✘	✓	✓
7	Windows 10	x86_64	msvcr1400.dll	✓	✓	✓	✓	✘	✘	✓	✓
8	Windows 7	x86	msvcr1400.dll	✓	✓	✓	✓	✘	✘	✓	✓
9	Windows 7	x86_64	msvcr1400.dll	✓	✓	✓	✓	✘	✘	✓	✓
10	Arch Linux	x86_64	libc-2.26.so	✓	✗	✘	✘	✓	✓	✓	✓
11	Debian Jessie	x86	libc-2.19.so	✓	✗	✘	✘	✓	✓	✓	✓
12	Debian Jessie	ARM	libc-2.19.so	✓	✓	✓	✓	✘	✘	✓	✓
13	Debian Jessie	PowerPC	libc-2.19.so	✓	✗	✘	✘	✓	✓	✓	✓
14	Debian Jessie	s390x	libc-2.19.so	✓	✗	✘	✘	✓	✓	✓	✓
15	Debian Stretch	x86_64	dietlibc 0.33	✗	✗	✘	✘	✓	✓	✓	✘
16	Debian Stretch	x86_64	musl-libc 1.1.16	✓	✗	✘	✘	✘	✓	✘	✓
17	Ubuntu 14.04 LTS	x86_64	EGLIBC 2.15	✓	✗	✘	✘	✓	✓	✓	✓

4.2 Empirical Results

We classify our data points into three categories:

1. The vulnerable implementations satisfying $\Delta_{\text{LOC}} > 0$ and $W(\Delta_{\text{LOC}}) = 100.0\%$ are marked in ✗. Here, a long buffer overflow on the stack allows for a complete stack canary bypass as CAN and REF can be overwritten at the same time.
2. The weak implementations satisfying $W(\Delta') = 100.0\%$ with $\Delta' \neq \Delta_{\text{LOC}}$ are marked in ✘. This requires an attacker to not only overflow a data structure located in the memory segment next to REF (maybe even in reverse direction), but also to get control of the execution flow by overwriting a buffer on the stack before the function containing the first vulnerability returns.
3. The secure implementations satisfying $W(\Delta) \neq 100.0\%$ are marked as ✓ in Table 1. These implementations do not offer the possibility to overwrite REF in memory and therefore are secure against the attack presented in this work.

In essence, Categories 1 and 2 violate the Assumption ❷.

4.3 Introduced Attack Vectors

We now discuss the practical implications for application security. For clarity we omit the discussion of weak implementations, as an attacker would always need more than one buffer-overflow in a vulnerable application to gain advantage of the situation. For this, we assume an adversary who is capable of triggering a buffer-overflow of suitable size on the stack. As such, we discuss possible attack vectors in two different scenarios depending on the threading model the target executable uses.

Forking. In a forking environment, the whole address space of the target binary is duplicated, including all CAN and REF values contained in memory. When an attacker is able to obtain information about one of the forked processes this renders randomness based countermeasures ineffective as all forked applications share the same randomness: ASLR becomes predictable [20] as well as all cookie values. Assuming an attacker is allowed to restart communication with the vulnerable application an oracle can be created as follows: the attacker overwrites a stack canary byte by byte and observes whether the application at the other end crashes. Only one out of 2^8 possible byte values will allow the application to continue execution. This effectively increases the chance of guessing the stack canary from $(2^8)^8 = 2^{64}$ to $(2^8) \cdot 8 = 2^{11}$ in the worst case—implying a more than significant difference in both attack duration as well as probability of success. This attack vector has already been discussed by researchers [14, 19, 21]. Note that a similar technique can be used to infer certain pointer values residing in the attacked application's stack frame.

Threading. On multi-threaded applications the insights from *CookieCrumbler* can be used in two ways. (1) *If the attacker can write null bytes and the application is mapped at a static address in memory:* all vulnerable implementations stack canaries can be completely bypassed by overwriting CAN and REF with the same value chosen by the attacker. As all program addresses are known, this case directly reduces to an ordinary *Return Oriented Programming* (ROP) attack. (2) *If the attacker is not allowed to write null bytes or the application's code section is not mapped at a static address (e.g.,* Position Independent Executable *(PIE)):* the attack can still succeed on Linux with *glibc*. The attacker will target the PointGuard value, which is also stored in the TCB, directly following REF. Equations 3 and 4 show that in PointGuard any protected pointer is first rotated by a fixed number of digits and then XORed with the PointGuard value (i.e., an attacker controlled number) in the considered setting.

The function in charge of terminating the program after a failed stack cookie check in *glibc* eventually ends up demangling a pointer to `pthread_once`. It is obvious that by the simple arithmetic used during pointer demangling, the attacker can detour the execution flow by a fixed offset to this function. From here on, no generic attack vector exists, but we want to point out that there are code paths in *glibc* that execute the assembly-equivalent of `execve("/bin/sh")` [22], which constitute valuable attack targets in our case. The likelihood of this attack succeeding heavily depends on the memory layout imposed by the dynamic loader on libraries. In our experiments we never observed a distance greater than 2^{24} between `pthread_once` and a gadget that eventually lead to remote code execution.

4.4 Impact

The tested Linux-based platforms (Android, Arch Linux, Debian, and Ubuntu) can be clustered into two different categories. Architectures which have dedicated TLS access registers (x86, x86_64, s390x, and PowerPC) that store the REF in

the TCB and architectures without a direct register access to the TLS (ARM). We have also analyzed the source code of *glibc* and categorized further architectures as TLS-based stack canary implementations: IA64, SPARC, and TILE. While we expect that our results can be extended to those architectures we did not had access to such hardware and did not include them in Table 1.

The TLS-based SSP for all the *libc* implementations are vulnerable to our attack in a multi-threaded environment by overwriting the REF via a stack-based buffer overflow[6]. SSP implementations where the REF is located in the Global section are more robust as it cannot be modified by an buffers overflow on the stack. This result can be seen in the LOC column in Table 1. Our evaluation also shows that most implementations fail to separate other data regions from the location of REF. This might be exploitable if the program uses thread-local variables. If one of the variables can be overflown, an attacker may overwrite the reference canary REF. In this case, the attacker needs two overflows (to change both REF and CAN). This is a difficult attack which also affects single-threaded applications, and therefore less critical issue.

Interestingly, *diet libc* defaults to storing the reference canary in the TLS, even if the application is not multi-threaded. Thus also the main thread stack is adjacent to the used TLS. Note that the main thread's stack and its TLS region are separated in the other implementations. This effectively breaks SSP for *diet libc*. Also, we point out that SSP can be bypassed for multi-threaded applications in all *libc* implementations.

Windows, macOS, and BSD derivatives store the reference cookie in the `.bss` section. Hence, they are not vulnerable to our overflow attack. However, column GLO in Table 1 shows that storing the reference stack cookie in the `.bss` region might open up a vulnerability. On Windows and FreeBSD, the stack canary is located in front of the global variables. Thus, the value might get overwritten by an overflow running towards lower addresses, which is less common yet not impossible. Only macOS, OpenBSD, and Android (on architectures without TLS based cookies, e.g., ARM) succeed to separate the reference cookie from all other memory regions. As OpenBSD, at the time of writing, is lacking a compiler with support for thread-local variables, it is not conducted in our experiments.

To get an overview how realistic the described attack is, we analyze the binaries installed on a vanilla Debian Jessie installation. About 40% of those programs depend on *pthreads*, which leaves them potentially vulnerable to our attack. Server applications, like web servers, often rely on threading to handle multiple clients at the same time and are particularly subject to such attacks.

5 Attack Mitigations

Re-randomizing REF on process creation (e.g., after forking) is a promising idea to increase canary entropy, as demonstrated by *RenewSSP* [14]. This approach mitigates our attack partially, but we also propose to modify the thread library to randomize REF for each thread.

[6] During the paper's review this issue was independently discovered by Ilya Smith: `https://github.com/blackzert/aslur/`

Frantzen et al. [23] argue to relocate REF to the PCB data structure, but unfortunately this introduced more deficiencies. We extend this idea by proposing the generation of per-function stack cookie values by XORing the static canary with the current stack pointer value to *borrow* randomness from `mmap`. Similarly, we can XOR REF with the return address of the protected function. However, this mitigation is only effective for scenarios where the code segment of the protected function is mapped at randomized addresses.[7]

Handlers running in a corrupted program context should strive to quit execution as fast as possible. *Glibc*'s `__stack_chk_fail` handler is a bad counter-example. It passes control through several layers of code which uses attacker controlled values from the stack. This opens the possibility for further exploitation. Clearly, the approaches taken by *Microsoft Visual C* (MSVC) and *musl libc* are preferable—the handler quits as fast as possible and, in case of MSVC, any reasoning about the crashed program's state (if at all) is performed using run-time data from the OS's kernel only.

Finally, the TCB must not be mapped adjacently to any memory structure that contains user-controllable buffers. The most direct way to achieve this is the introduction of a mandatory guard page mapped with *no access* protection at the *bottom* of the stack. Note that even though *glibc*'s pthread implementation apparently offers such functionality (`pthread_attr_setguardsize`), it is not automatically turned on by software intending to use threads and even more importantly only offers a mechanism to map a guard page on the *top* of the stack.

6 Improving Sophisticated Protection Mechanisms

To highlight how stack canaries can improve application security, we consider the *C* program in Figure 1. Depending on the mitigation mechanisms added when compiling this program, the authentication bypass can be trivially triggered. We consider this example in the context of two software protection mechanisms:
SafeStack: SafeStack [8] is a State-of-the-Art CPI implementation that logically separates the architectural stack into a safe and unsafe region. The safe region contains all control-flow related data while the unsafe stack contains user-controlled data (e.g., arrays).
Stack Canaries: We use the standard implementation employed by *LLVM*.
When compiling with SafeStack enabled, the variables `password` and `admin_hash` are allocated in the unsafe stack, whereas the return addresses are in the safe stack. The stack-based buffer overflow in the `auth` function make bypassing the security check trivial: an attacker first overflows the `password` buffer and then overwrites the `admin_hash` with the hash matching the provided password. When stack canaries are enabled the attack is no longer trivial. After filling the `password` buffer, an attacker has to overwrite CAN to reach the `admin_hash`. Once the `auth` function returns, the canary corruption will be detected and the program will terminate.

[7] OpenBSD very recently added "RETGUARD" which is similar to our proposition `https://marc.info/?l=openbsd-cvs&m=152824407931917&w=2`.

```
int auth(char * valid) {
    char password[32];
    gets(password);
    return strcmp(valid, crypt(password, valid)) == 0;
}
int main(void) {
    char admin_hash[] = "$1$01234567$b5lh2mHyD2PdJjFfALlEz1";
    if (auth(admin_hash)) puts("Welcome to the Admin Area");
}
```

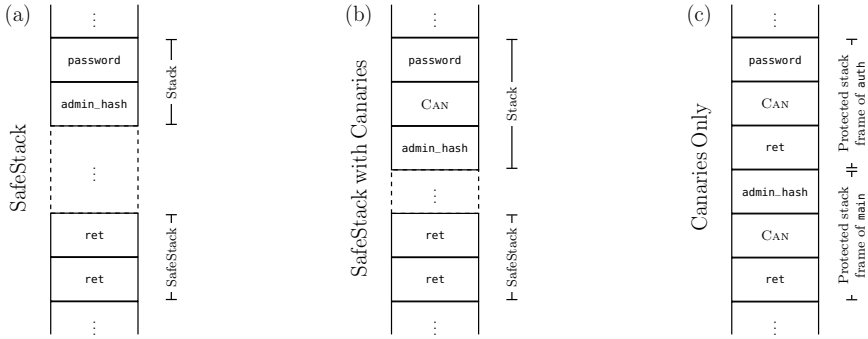

Fig. 1. Different stack layouts of a *C* program exposing an authentication bypass vulnerability when using (a) SafeStack (b) SafeStack with Canaries, (c) Canaries only.

The same security properties (protecting buffers of adjacent stack frames) are achieved regardless of the usage of SafeStack. To reach the `admin_hash` buffer, an attacker has to overwrite the CAN value. As corrupting CAN should result in program termination—the fact that the return address `ret` is also reachable by the overflow becomes irrelevant. While SafeStack threat model does not include the corruption of non-control-flow-related data structures we argue that stack canaries can improve resistance of CPI against *non-control-flow targeting attacks*.

7 Conclusion

In this work we presented *CookieCrumbler*, a multi-platform framework to systematically study stack canary implementations. We discovered scenarios which are prone to a novel attack that allows bypassing State-of-the-Art stack protection mechanisms in threaded environments. In addition, we introduced new ideas for a more advanced attack that abuses the way exception routines and pointer mangling mechanisms work together. Finally, we believe this work provides systematic insight into the qualitative implementation details of stack canaries used by modern OSes and can serve as a basis for future explorations of security critical parts of the OSes and C standard libraries in use today.

Acknowledgments

The research was supported by the German Federal Ministry of Education and Research under grant 16KIS0327 (IUNO) as well as the SeCiF project within the French-German Academy for the Industry of the future

References

1. P. Streak, The Morris Worm: A Fifteen-Year Perspective (2003).
2. Aleph One, Smashing the Stack for Fun and Profit, Phrack 7 (49).
3. C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, H. Hinton, StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, in: USENIX Security Symposium, 1998.
4. Free Software Foundation, Using the GNU Compiler Collection, `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html` (Feb. 2018).
5. The Clang Team, Clang's Documentation, `https://clang.llvm.org/docs/` (Feb. 2018).
6. Microsoft Developer Network, Compiling a C/C++ Program, `https://msdn.microsoft.com/en-us/en-en/library/8dbf701c.aspx` (Feb. 2018).
7. C. Cowan, F. Wagle, C. Pu, S. Beattie, J. Walpole, Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, in: DARPA Information Survivability Conference and Exposition (DISCEX), 2000.
8. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, D. Song, Code-Pointer Integrity, in: Symposium on Operating System Design and Implementation, 2014.
9. M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, Control-Flow Integrity, in: Conference on Computer and Communications Security (CCS), 2005.
10. S. Chen, J. Xu, E. C. Sezer, Non-Control-Data Attacks Are Realistic Threats, in: USENIX Security Symposium, 2005.
11. L. Szekeres, M. Payer, T. Wei, R. Sekar, Eternal War in Memory, IEEE Security & Privacy 12 (3) (2014) 45–53.
12. C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, E. Walthinsen, Protecting Systems From Stack Smashing Attacks With StackGuard, in: In Linux Expo, 1999.
13. P. Wagle, C. Cowan, et al., Stackguard: Simple Stack Smash Protection for Gcc, in: GCC Developers Summit, 2003.
14. H. Marco-Gisbert, I. Ripoll, Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers, in: Network Computing and Applications, 2013.
15. C. Cowan, S. Beattie, J. Johansen, P. Wagle, Pointguard: Protecting Pointers From Buffer Overflow Vulnerabilities, in: USENIX Security Symposium, 2003.
16. U. Drepper, Pointer Encryption, `http://udrepper.livejournal.com/13393.html` (Feb. 2018).
17. R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, T. Walter, Breaking the Memory Secrecy Assumption, in: European Workshop on System Security (EUROSEC), 2009.
18. Y. Ding, Z. Peng, Y. Zhou, C. Zhang, Android Low Entropy Demystified, in: IEEE International Conference on Communications (ICC), 2014.
19. W. H. Hawkins, J. D. Hiser, J. W. Davidson, Dynamic Canary Randomization for Improved Software Security, in: Annual Cyber and Information Security Research Conference, 2016.
20. H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, D. Boneh, On the effectiveness of address-space randomization, ACM CCS 2004.
21. A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, D. Boneh, Hacking blind, 2014 IEEE Symposium on Security and Privacy.
22. david942j, One Gadget, `https://github.com/david942j/one_gadget` (Feb. 2018).
23. M. Frantzen, M. Shuey, StackGhost: Hardware Facilitated Stack Protection, in: USENIX Security Symposium, 2001.