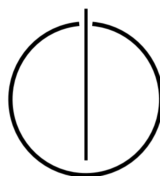
DEPARTMENT OF INFORMATICS
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Improving Application Software Security in Linux

Sebastian Neubauer



DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Verbesserung der Anwendungssicherheit in Linux

Improving Application Software Security in Linux

Author:	Sebastian Neubauer
Supervisor:	Prof. Dr. Claudia Eckert
Advisor:	Julian Kirsch, M.Sc.
	Bruno Bierbaumer, Dipl.-Ing.
Submission Date:	15. July 2017

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Ort, Datum	Sebastian Neubauer

Acknowledgments

I would like to thank Julian Kirsch and Bruno Bierbaumer for giving me this nice topic and having great and interesting ideas while guiding me. All my questions were answered pretty fast.

Abstract

A lot of software that we use today is written in C and C++. Especially these memory unsafe languages induce vulnerabilities in applications. Therefore people developed techniques which make the exploitation of programming faults harder. The goal of this work is to fortify these techniques and introduce new methods to make programs more secure. We analyze the following techniques with regard to their security effect and their impact on performance:

1. Checking the position of the stack pointer in every system call, which showed an overhead of $(2.7 \pm 3.3)\,\%$ in a microbenchmark. The measured overhead shows a large standard error, thus we cannot be sure that our patch actually makes applications slower.

2. Adding random gaps between sequent `mmap` allocations, leading to a maximal speed loss of $(2.8 \pm 0.5)\,\%$.

3. Improving the Stack Smashing Protector (SSP) by clearing the SSP from the stack after checking it (no measureable performance change) and generating a random SSP for every function call ($(265 \pm 4)\,\%$ times slower in a microbenchmark, while more realistic workloads showed a regression of $<2\,\%$).

We created patches for the stack pointer check and the SSP improvements. We consider the security, which is added by the discussed patches, useful enough and the performance overhead low enough, to make use of these techniques.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 2

2 Background **3**
2.1 Process Memory Layout 3
2.2 `mmap` 4
2.3 Shellcode 5
2.4 Stack Smashing Protector 5
2.5 No eXecute bit 6
2.6 Return Oriented Programming 6
2.7 Stack Pivoting 7
2.8 Address Space Layout Randomization 7
2.9 RELocation Read Only 7
2.10 SafeStack 8

3 Design **9**
3.1 Stack Pinning 9
3.2 Randomize `mmap` 9
3.3 Improving the Stack Smashing Protector 10

4 Implementation **12**
4.1 Stack Pinning 12
4.2 Randomize `mmap` 12
4.3 Improving the Stack Smashing Protector 13

5 Evaluation **15**
5.1 Stack Pinning 15
5.2 Randomize `mmap` 17
5.3 Improving the Stack Smashing Protector 19

6 Discussion **22**
6.1 Related Work 22
6.2 Future Work 23
6.3 Conclusion 24

Acronyms **27**

Appendix **28**

1 Introduction

In this section we motivate the introduction of new security mechanisms. Afterwards we list our contributions.

1.1 Motivation

Software, especially bigger projects, contains bugs (the only exception here are formally verified programs, but even there the specification could still contain bugs). If an application does not have the expected behaviour, it makes its users (and authors) unhappy. But it gets more critical if a bug can be abused and becomes a vulnerability, which can be exploited. Exploitation in this context means, an attacker interacts with the application in a way that it to shows unwanted behaviour. Unwanted behaviour can be for example remote code execution, which permits the attacker to execute arbitrary code inside the program.

Memory safe languages like *Java* are used more and more. This can for example be seen in the distribution of languages in the newly created repositories on *GitHub* [4]. From the perspective of security, unfortunately many libraries, existing and new projects are still written in C/C++. Important projects (e. g. operating system kernels like linux and Windows) are often written in low-level languages, which are vulnerable to memory corruption. This can also be observed for Internet Of Things (IoT) projects. According to an *Eclipse* survey from 2016, 48 % of IoT projects involve C, only topped by Java with 52% [25] (more than one language per project was possible in this survey).

The security impact of languages like C and C++ is that they can be more easily exploited than e. g. Java. Kernel exploits are used to elevate privileges and gain power over a system, which makes them a wanted attack target. Taking IoT devices over can lead to big botnets, that are able to flood the internet with traffic and take down servers. This is possible because they are always connected to the internet. For this reason, their security is also important.

As the amount of devices that are connected to the internet and the amount of software that is running on them is increasing, the challenge is to make code less exploitable in a generic way. Additionally that has to happen under the constraint that the performance impact is small enough. Otherwise, even if a lot of security holes were fixed, the technique would not be used.

Many techniques trying to improve security have already been invented. AdressSanitizer (ASan) [9] was built to dynamically detect memory errors but it is seldom activated in production environments. Even if ASan if fast compared to other tools, it slows the application down by a factor of about two. For the reason of speed, techniques that are actually used in production do not incur a big performance penalty but also cannot detect as many exploits. Some widely used hardening methods will be covered in section 2.

1.2 Contributions

Our goal is to improve the security of applications by developing security mechanisms that do not cost much performance but make bugs harder to exploit.

One of our contributions is a patch for the linux kernel that checks in every system call if the stack pointer actually contains an address which points to the stack area.

Another patch, that already exists, adds random gaps between `mmap` allocations. We show that this change comes with a maximal performance overhead of $(2.8 \pm 0.5)\,\%$.

Finally we improve the hardening done by the Stack Smashing Protector (SSP) in two ways, which we implemented for the *llvm* compiler framework. The first part is to remove unnecessary copies of the SSP on the stack so that they cannot be leaked anymore. The second part of the fortification is done in conjunction with *SafeStack*. We introduce a method to randomize the used SSPs. The security and performance impact are also examined for these changes.

Because this work is focused on linux, we do not create an implementation or measure the performance on other operating systems. We do not provide perfect security or control flow integrity with these changes but we focus on the improvement of exploit mitigation while still maintaining reasonable performance results.

2 Background

This part gives a general introduction into the basics of binary exploitation and security mechanisms which are important for the understanding of the newly developed methods.

The focus lies on the `amd64` architectures and on the linux operating system. Many parts though are applicable to other architectures and systems as well.

2.1 Process Memory Layout

This section explains the rough memory layout at the start of a process, which is divided into multiple segments. These parts and their differences are listed in Table 1.

There exists only one heap per process, thus it is possible to allocate an object in one thread and free it in another thread. For performance reasons, each thread has its own area on the heap, so locks are not always needed for an allocation.

On popular architectures, the stack grows from high to low addresses, which is – as we will see later – beneficial for an attacker. The used stack size grows and shrinks with each function call and return. The area that is used for one function call is called a *stack frame*. A stack frame stores the return address of a function and its local variables. Often local variables are not addressed relative to the stack pointer, which always points to the last element on the stack, but relative to the *frame pointer* (also called *base pointer*), that is stored in another register and points to the start of the current stack frame. These pointers are useful for analysing the stack, because the contents of this register will be saved onto the stack before the base pointer for the current function is set. This leads to a linked list on the stack, as it can be seen in Figure 1, which is based on the code in Listing 1 (the `rbp` register contains the base pointer and the `rsp` register the stack pointer).

Section	**Content**	**Rights**	**Occurrence**
program code	machine code, constant data	`r-x`	1 for the main executable and 1 for each library
read-only data	constant data	`r--`	
read-write data	static variables	`r-w`	
heap	runtime allocations	`r-w`	1 per process
stack	return addresses, local variables	`r-w`	1 per thread

Table 1: *Memory layout of a process*

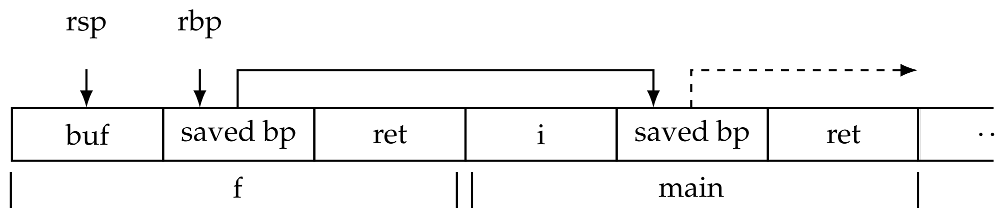


Figure 1: *Stack layout*

```c
void f() {
    /* A char array of size 50 on the
        stack. */
    char buf[50];
    /* Read an input line without bounds
        checking. */
    fgets(buf);
}

int main() {
    int i;
    for (i = 0; i < 5; i++)
        f();
    return 0;
}
```

Listing 1: *Sample C code for Figure 1*

```asm
enter
; is equivalent to
push rbp
mov rbp, rsp

leave
; is equivalent to
mov rsp, rbp
pop rbp
```

Listing 2: *x86_64 assembler instructions `enter` and `leave`*

The machine code, that a compiler emits to create and rewind the frame pointer, consists of an `enter` and a `leave` instruction. They are actually slower than a stack and a `mov` operation as shown in Listing 2. For this reason, optimized binaries will use the latter representation instead of `enter` and `leave`.

2.2 `mmap`

In the following section, we describe a linux system call, which is important to understand the developed security mechanisms later.

The `mmap` function of the C library is used in applications to allocate new user space memory. To accomplish this, it needs to use the `mmap` system call, to get into linux kernel code. The kernel then allocates one or more *pages* (a page is usually 4 KiB of memory) and returns the address of the new memory area. The system call gives the opportunity to specify an address at which the memory should be placed. If this address is set to zero, the kernel will choose an address on its own.

Using `mmap` directly is not useful if one needs only a small amount of memory because `mmap` is only able to manage memory areas of which the size is a multiple of the page size. Instead we usually call `malloc` and `free` to manage small allocations of memory in C. These methods are implemented in the C library which itself uses `mmap` to allocate memory from the kernel. It then divides the memory into smaller pieces, which are returned by `malloc`. To make memory allocations fast for multi-threaded applications, the code avoids locks by assigning a separate memory area (for memory allocations) to each thread. If more memory is needed by `malloc`, `mmap` is called or – for the main thread – `sbrk`, which

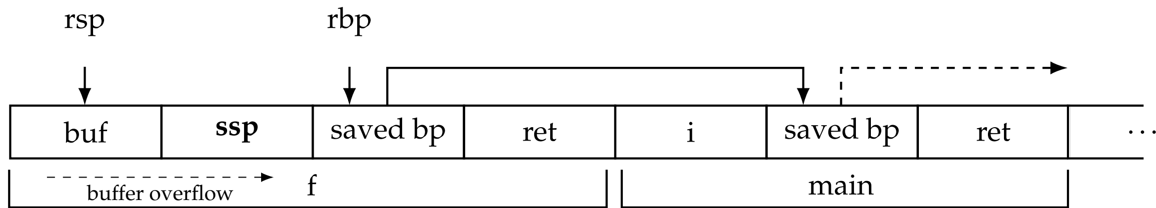


Figure 2: *Stack Smashing Protector*

increases or decreases the data segment size. For large memory requests ($> 128\,\text{KiB}$ by default), `malloc` redirects to `mmap`.

Another function of `mmap` is to map the content of a file into a process's memory or to share memory between processes.

2.3 Shellcode

Starting with this section, we explain certain attack and defense techniques that are available and used today.

The reason why C and C++ are dangerous languages, is that they have no bounds checks on array indexing and pointer dereferencing by default. That means if the code author forgets to check the bounds, an attacker can supply a too long input (in our example from Listing 1, longer than the 50 characters, that the buffer can hold) and overwrite the data behind the buffer. This can easily happen in line 5 in Listing 1, as no bounds are checked when reading the input. This so called *buffer overflow* enables the attacker to overwrite the saved frame pointer (which will be stored in the `rbp` register in the `main` function) and the return address. If the stack growed from low to high addresses, tampering with control data would not be possible in this way.

By overwriting the return address, the attacker can specify, which code will be executed when `f` returns. To abuse this ability, one can write a short snippet of assembler code which starts a shell (this is called *shellcode*) into the buffer and overwrite the return address with the address of the buffer. After `f` returns, the shellcode will be executed and the attacker has succeeded to start a shell, which he can now use to execute whatever he wants.

Due to the fact that this is easy to accomplish, techniques were developed that should make it harder, if not impossible, for an attacker to execute his code.

2.4 Stack Smashing Protector

The Stack Smashing Protector (SSP) is one technique which tries to prevent attackers from successfully abusing buffer overflows. The SSP that is used in todays compilers is based on StackGuard [7] from 1998 and used to protect return address and frame pointer from buffer overflows like the one that is described in subsection 2.3. At the beginning of a function, some value is pushed onto the stack, after the return address and the saved base pointer, but before any local variable. This can be seen in Figure 2. Before returning from the function, the SSP will be compared to its original value. If they are different, it indicates a buffer overflow and the process aborts. When compiling executables with `-fstack-protector-strong`, a prolog and an epilog that set and check the SSP will be inserted for each function that has a local buffer. In the example this will be done for `f`.

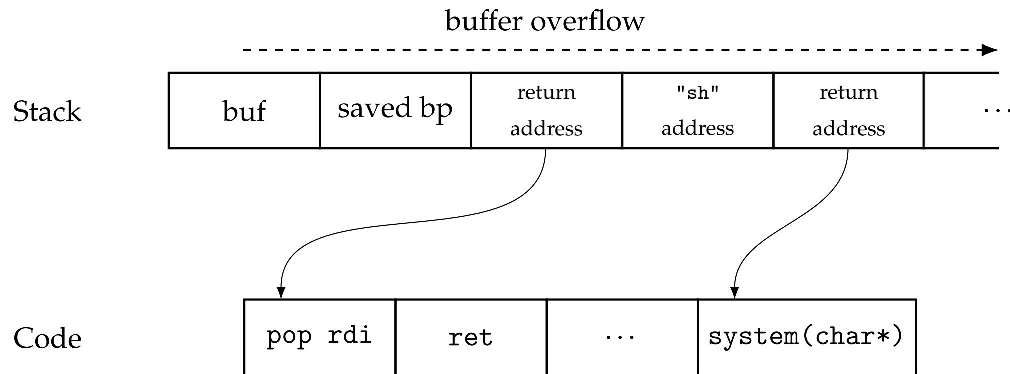


Figure 3: *Return Oriented Programming*

Methods with no buffer among the local variables are not vulnerable to a buffer overflow and will not be modified. The value that is written on the stack is different per process start. It is initialized from the *auxilary vector* that is filled by the kernel and then stored in the thread local storage by the C library.

These protectors, also called *canaries*, provide a strong protection against buffer overflows. Still, there are ways to avoid and circumvent these measures. For example, they do not at all protect from buffer underruns [14].

2.5 No eXecute bit

Another countermeasure against the use of shellcode is to mark all memory non-executable except the area where the program code is stored, which is instead read- and execute-only, hence not writable. This is also known as write xor execute (w$\oplus$x). This means an attacker cannot store code and execute it without changing the memory permissions.

No eXecute (NX) in linux with grsecurity [3] and Data Execution Prevention (DEP) on Windows (available since 2004 [2]) implement this mechanism. This effectively thwarts the usage of shellcode, but it can be circumvented by using *return oriented programmining* which is described in subsection 2.6.

2.6 Return Oriented Programming

NX prevents an attacker from executing own code. Therefore he can only reuse already existing code, which can be accomplished by overwriting the return address with the address of another code part. He can abuse this by jumping to short sections of code, that are followed by a return instruction. The attacker can supply many of these locations on the stack and they will be executed one after the other. These short sections of code are called Return Oriented Programming (ROP) gadgets and can for example be used to setup arguments for a final call to `system` (return-to-libc) as shown in Figure 3. First, the address of an `"sh"` string (the `sh` command starts a new shell) is stored in the `rdi` register, which contains the first argument in our calling convention. Then `system` is called by supplying another return address, taking the `"sh"` string as an argument.

2.7 Stack Pivoting

Sometimes, a difficulty with using ROP is that a buffer overflow is not large enough to create a working exploit by a simple buffer overflow. However, it can still be possible to abuse the overflow by writing the ROP chain (the return addresses that the executable will jump to) into another memory region and – using the limited overflow – point the stack pointer to the prepared memory, therefore the previously stored gadgets will be executed. This technique is called *stack pivoting* because the stack pointer is moved away from the original stack position.

2.8 Address Space Layout Randomization

A widely used protection mechanism that is implemented in todays operating systems puts memory segments at random offsets. The idea is that often an attacker needs to know some memory addresses in order to exploit a program. We can make attacks harder, by randomizing the addresses, that are needed for an exploit. Now the attacker either has to leak them or find some other reliable way for an exploit. Address Space Layout Randomization (ASLR) implements this idea by adding random offsets to the position of all memory regions. This technique is used since 2006 in Windows [10] and is also implemented in different flavors for linux and BSD distributions. Due to ASLR, libraries, heap and stack are allocated at non-deterministic addresses on linux.

However, even if ASLR is on, the executable itself can still be at a static address. To randomize also the position of the executable, the Position Independent Executable (PIE) option has to be enabled during compilation. Otherwise an attacker can still call ROP gadgets from the main program code, which is often sufficient.

On an up to date arch linux system in June 2017, only eight out of 478 binaries in the `base` and `base-devel` package groups are position independant. On a fresh *Debian 9 Stretch* installation (including an ssh server and gnome), 1035 out of 1045 binaries in the `/bin` and `/usr/bin` directories are position independant. Ubuntu 17.10 [13] and openSUSE Tumbleweed [17], will enable PIE by default for packages to fix this flaw.

One more weakness of ASLR is the amount of randomness on 32-bit systems. Due to the limited address space, the offsets only contain 16 random bits which does not provide sufficient security [24].

2.9 RELocation Read Only

To use dynamically linked libraries, a *loader* is needed. The *loader* loads needed libraries at runtime into the address space of a process. To enable the process to call functions from these libraries, it writes pointers to the needed functions into the main program's address space. The memory area that stores all these pointers is called Global Object Table (GOT).

A widely used attack technique to gain code execution is to overwrite entries in this table. Every time the application tries to execute the original function, the code at the newly written address is called.

To mitigate this, all addresses can be resolved at the start of a process. Afterwards, the GOT memory is set read-only thus an attacker cannot tamper with these values anymore. This mechanism was invented in 2011 and is called RELocation Read-Only (RELRO) [6].

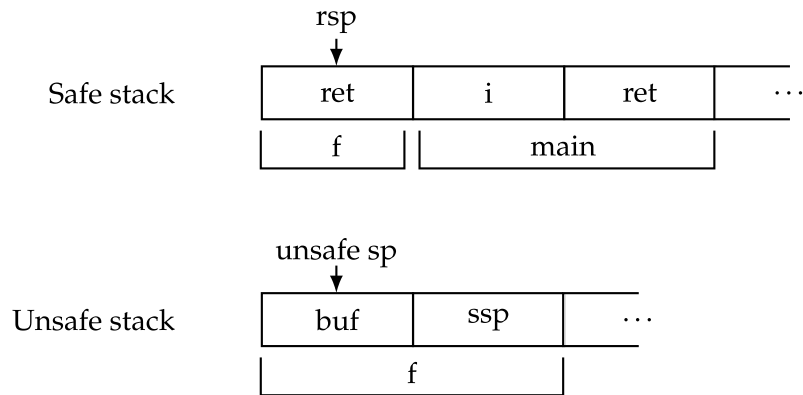


Figure 4: *SafeStack*

2.10 SafeStack

A lately developed mechanism, which aims to prevent attackers from overwriting control data, is called SafeStack [5]. It can be found in the llvm *clang* compiler. The idea is to mitigate buffer overflows by dividing the stack into two parts. The *unsafe* stack holds all buffers and the *safe* stack stores important, overflow-resistant data like return addresses, registers and local, non-buffer variables. Therefore return addresses cannot be overwritten by a buffer overflow because they are not stored behind a buffer. The cost is to hold a second stack pointer for the unsafe stack which can be seen in Figure 4.

Using SSPs in conjunction with this method does not protect return addresses, because they are stored in another memory region. But canaries can still be useful to protect the application. A class of attacks that do not overwrite control data are so called data-only attacks. They do not touch the return addresses but they overwrite regular data structures, which can be critical for e. g. a boolean that gives the current user administrator rights. Storing a SSP between function frames on the unsafe stack prevents overflows from one function frame into another. This is important to keep the integrity of data. It was even proposed to put a canary in front of every variable to prevent data-only attacks by buffer overflows [1].

3 Design

For each developed mechanism we will describe the existing problem, what we propose to fix the problem and how our solution performs.

3.1 Stack Pinning

With this patch we want to prevent attackers from successfully using stack pivoting. As described, stack pivoting is a commonly used technique to work around limited buffer overflows on the stack. An instruction that resets the stack pointer (e. g. when restoring the frame pointer) is used to point the stack pointer to an attacker controled memory area (for example some buffer that contains previously stored input data) during the execution of a ROP chain. For a valid process there are very few reasons to fiddle with the stack pointer, hence if it is not pointing to the stack area, it is unwanted behaviour and we conclude that the process is exploited.

The point where most exploits finally arrive is communication with the operating system. Often a shell is started using the `execve` system call, or `read` and `write` are used to leak information. As system calls are a critical point here, a check can be implemented in the kernel, right before the system calls are executed. A coarse but fast method is to check whether the stack pointer points to the memory area that was originally reserved for the stack.

Starting with Windows 8, which was released in 2012, Windows checks the stack pointer before executing system calls that are related to managing memory (which are often used in exploits on Windows) [22]. If the stack pointer does not point to the stack of the application, Windows kills the process.

Until now, there was no such implementation for linux. Our patch adds this check at the beginning of every system call. Looking at the stack pointer only in specific system calls would be faster but does also imply the risk, to overlook an important system call that makes it possible to continue exploitation. For example if `exec`, `open` and `write` do the check, an attacker could still use `openat` and `writev` to read secret data.

3.2 Randomize `mmap`

ASLR randomizes the base offset for addresses returned by `mmap`, when it is called with zero as the address argument. New memory blocks are assigned sequently, just in front

of the already assigned memory. The problem of this behaviour is that leaking a single heap address (which was allocated using `mmap`) makes ASLR useless for the heap area. Allocations are done deterministically thus an attacker is able to recompute all other addresses, even when they are on pages that were allocated later. This knowledge is useful e. g. for stack pivoting (see subsection 2.7).

In this section we discuss a remedy that makes the address of further areas, which are allocated using `mmap`, nondeterministic. This means an attacker cannot recompute all addresses and use them for exploitation.

William Roberts proposed a patch for the linux kernel that randomizes the addresses returned by `mmap` [21]. To accomplish this, the kernel adds a randomly chosen offset to nearby allocations, creating a gap between them and decoupling the addresses of subsequent allocations.

Using this method, if an attacker leaks one address, he can recompute addresses from the same `mmap` allocation, but he can only guess the position of other heap memory areas.

3.3 Improving the Stack Smashing Protector

When using the `-fstack-protector-strong` compiler option, the SSP is written and checked for each function call where a buffer is stored on the stack. The fact that the canary value is only set when a process is started and keeps its value throughout the lifetime of a process, induces some vulnerabilities. If it is leaked by an out-of-bounds read or by leaking uninitialized memory, it gets useless, because an attacker can just overwrite the SSP with the correct value, which he now knows, remaining undetected. Another way to find out its value is possible with servers that fork themself for each connection. The `fork` system call will clone the process memory, including the SSP. If an attacker can overflow a buffer, he can overwrite only the first byte of the protector ($\frac{1}{256}$ chance to succeed) and observe if the process aborted. If not, he guessed the first byte successfully. By guessing a static value byte by byte, the attack gets feasible. This specific problem was addressed by generating a new SSP on forks [16, 19].

The SSP values are written onto the stack, but they are not cleaned up therefore the value lies on the stack until it is overwritten by a subsequent function call. The presence of these values increases the probability that it is possible for an attacker to leak the value of the SSP. It can for example happen through reading uninitialized memory, that means the value of an unset variable is leaked to the attacker. If the previously stored value at this position is the canary, the attacker gets to know its value and the stack protector becomes useless, as described before. We want to improve the protection of SSPs by removing the values that are not needed anymore.

A simple way to prevent this possible leakage of the SSP is to set the canary memory to zero after checking its value. This leads to no more unnecessary copies of the SSP on the stack and completely eliminates the risk to leak the SSP value by uninitialized variables.

In addition, we developed a technique that makes the stored canary values random. This mitigates all attacks that are based on a static SSP value. Leaking a canary or parts of it has little gain with this change because the next canary that is used has a completely different value. Therefore the attacker cannot compute the value of other canaries, based on leaked SSPs.

To achieve this, we xor the static SSP with a randomly generated value each time a function is called. One problem with this idea is that the used random value has to

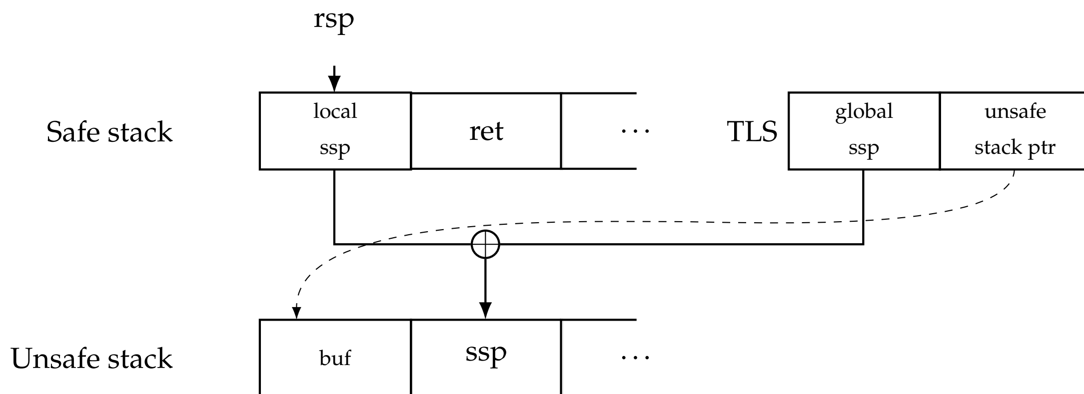


Figure 5: *Randomize canaries*

be stored at a secure location, that cannot be accessed as easily by an attacker as the location of the SSP. For this goal we leverage `SafeStack`. The buffers and checked canaries lie on the separate, unsafe stack while the random value is stored at the same area as the return address, on the safe stack. When checking the canary value before a function returns, we xor the global SSP and the random value again. The dataflow is visualized in Figure 5. As the value on the unsafe stack is completely random, leaking this value leads to no conclusion for other SSP values, thus only the leaked canary can be successfully overwritten by an attacker, all other changes will be detected.

4 Implementation

Some corner cases and restrictions of the presented techniques are discussed in the following sections.

4.1 Stack Pinning

To understand the implementation of the patch, one has to know that in linux, different processes and threads are handled in an equivalent way. Each thread and process is a task and threads of the same process just share the same memory, file descriptors, etc.

The stack pinning implementation for the linux kernel adds two new attributes to the task struct, which holds all information for a task, that the kernel needs: The start and the end of the current stack region (shown in Listing 3). If the process forks, we either copy the bounds or – if a new thread is created using the `clone` system call – we store the new stack bounds of the thread. The allowed range is determined by the virtual memory area where the stack pointer points to. In the case of a new thread, this area is allocated by the application and the start address for the stack is supplied to the kernel.

We have to respect that the main stack, which is reserved at the start of an application, is able to grow up- or downwards, depending on the architecture. If the current stack is exhausted, the kernel expands the memory area and updates the bounds.

Another case that has to be considered is the alternate signal stack. If a signal is sent to a process (like when a user presses `Ctrl+C` in a terminal) and a signal handler was registered before, the current task is interrupted and the handler function is executed on top of the current stack. In the rare case, where the stack is completely exhausted, a signal will be sent, but it cannot be handled on the regular stack, because it is already full. For this case the linux kernel gives applications the possibility to register an alternate stack for signal handling. If such a stack is registered, it will be used to handle every incoming signal. This memory area is also a valid range for the stack pointer, which has to be kept in mind when checking for stack pivoting.

4.2 Randomize `mmap`

The original patch consists of 24 added lines. The key function in the patch is the `randomize_mmap` function, which takes a free memory area that was found by the kernel

```
struct task_struct {
    ...

    /* The lower bound of the process stack, set to zero if not used. */
    unsigned long stack_start;
    /* The upper bound of the process stack: */
    unsigned long stack_end;

    ...
}
```

Listing 3: *Extended task struct*

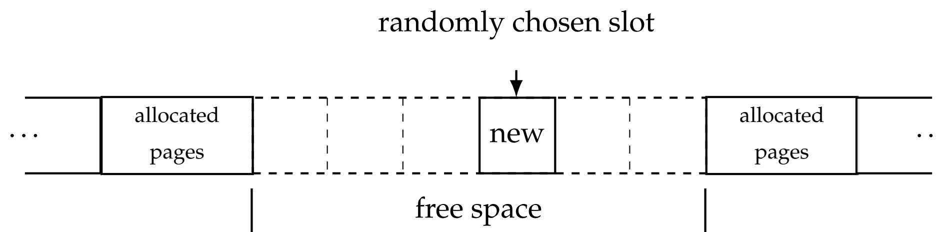


Figure 6: *Add a gap between* `mmap` *allocations*

and divides it into slots of the requested size. Then it returns a random slot as illustrated in Figure 6. An excerpt from this function be seen in Listing 4.

4.3 Improving the Stack Smashing Protector

Zeroing the canary after checking it can be done with seven added lines of code, including comments. Our patch covers the SSP check on x86 architectures, including the case when SafeStack is used. There is one more implementation of a stack protector in *llvm* in the SelectionDAG stage, that is not modified in our work.

Implementing randomized SSPs is more complex. The question arising with this idea is how random values can be generated. On newer `x86` architectures (since Ivy Bridge and Bulldozer v4 [27]), the `rdrand` instruction exists. This instruction overwrites a register with a random value. Unfortunately there is no equivalent on other architectures. A Pseudo-Random Number Generator (PRNG) like a Linear Congruential Generator (LCG) can generate fast random numbers. However, even if we truncate the generated values, an attacker needs only a few leaks to recompute the used parameters [8], which would again enable the attacker to overwrite the canary without being detected. For this reason, the only reasonable protection can be achieved by using a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG).

```
unsigned long randomize_mmap(unsigned long start, unsigned long end, unsigned long len)
{
    unsigned long slots = (end - start) / len;
    return PAGE_ALIGN(start + ((get_random_long() % slots) * len));
}
```

Listing 4: *Add gaps between* `mmap` *allocations*

The random number generator which is used in our patch is *ISAAC* [12]. ISAAC comes in different versions for different purposes. The used version of ISAAC is the optimized 64-bit version for 64-bit systems and the portable version for 32-bit systems. The internal state and the output array of the generator are 256 words large where the size of a word is 64 or 32 bit, depending on the architecture. For our purpose we use a slightly modified version of ISAAC. It produces the same results as the original versions but it does not work in blocks that refill the whole output and state. Instead, the output consists of only four words and is regenerated every fourth function call. That is done to assure that function calls do not stall for a longer time, which would be the case if the output buffer was completely recomputed every 256 calls.

The entropy to initialize the random number generator is taken from the original SSP. Leaking that value would allow an attacker to recreate the random numbers, so the value is overwritten with a generated random word, after the random number generator was initialized. Creating a new thread initializes the thread's local ISAAC state and the canary with a random number, generated by the starting thread.

5 Evaluation

After describing the concrete implementation, we discuss the performance impact and the security implications of the proposed changes.

The tables that display the performance results contain the following columns:

1. The configuration, which was used to execute the test.
2. The average of the measurements and the corresponding standard error of the mean (SEM), which is computed by the student-t test using a confidence level of 68.3 %.
3. The standard deviation (SD) of the results.
4. The amount of taken samples.
5. And a performance comparison, relative to the first row. A positive value means, that the result in the respective row was slower than the basis point, a negative value means that it was faster.

For the benchmarks in the *Stack pinning* and *Randomizing `mmap`* section, a *QEMU* virtual machine was used. The VM had eight out of twelve threads of the host CPU (Intel i7-5820K @ 3.3 GHz). The memory consists of 4 GiB of RAM and a 20 GiB disk image. The disk is a raw image and the access is not cached by the host, which is important to not distort the results. The used kernel is based on linux 4.11.0-rc5.

For the SSP changes, the benchmarks were executed on a six core/twelve thread Intel i7-5820K @ 3.3 GHz with 20 GB RAM.

ApacheBench was used from the apache tools in version 2.4.25. The server that was stressed is nginx 1.12. Apart from the microbenchmarks and the modified ApacheBench (the public version from `openbenchmarking.org` does not compile on arch linux), all benchmarks are publicly available at `openbenchmarking.org`.

5.1 Stack Pinning

Stack pinning is done by default on Windows. For linux this approach works almost always good too. Unfortunately some rare applications change the used stack area by design and it is undecidable for the kernel if such an application is currently exploited or working correctly. One example is *Wine*, which can execute Windows applications on

linux and for that purpose it prepares another stack and switches to the new memory area when starting the Windows program. The same behaviour can be observed with the *Go* programming language, when using goroutines. The stack frames for these functions are also stored on the heap. Because a kernel with activated stack pinning checks the position of the stack pointer in every system call, these applications would get killed as soon as they try to make a system call.

Due to that behaviour, our proposed patch cannot activate stack pinning by default for every application. We decided to change this feature to be opt-in per process, so existing applications continue to work, no matter where the stack pointer points to. Programs can make a `prctl(PR_PIN_STACK, 0, 0, 0, 0)` system call, which will activate the protection and save the stack bounds for the current stack pointer. After this system call succeeded, all subsequent system calls will first check the position of the stack pointer before proceeding. If the stack pointer does not point to the same memory region that was used while activating the feature, the process will get killed (an exception is the alternate signal stack as explained in subsection 4.1).

Security With regard to security, the same counter attacks as for Windows are possible [22]. That means, if an attacker knows the address of the stack, attacking is still possible. If the size of the buffer overflow, which is used to start the exploit, is too small, an attacker can use stack pivoting to execute a bigger ROP chain. Using this chain, he can write some ROP gadgets onto the real stack and execute system calls from this stack position as displayed in Figure 7. This will not trigger the check in the kernel because the actual system call is done from the stack memory area and the kernel cannot observe if the stack pointer was pointing somewhere else in between system calls.

If an attacker is not able to execute system calls from the real stack, successful exploitation using system calls is impossible, because the application will get killed, before any system call is executed.

Performance To measure the worst case performance impact, we executed the `getpid` system call for 2×10^8 times in a loop. The benchmark was executed in three different configurations: Using an unpatched kernel, using a patched kernel without activating stack pinning for the benchmarked process and with activated stack pinning on the patched

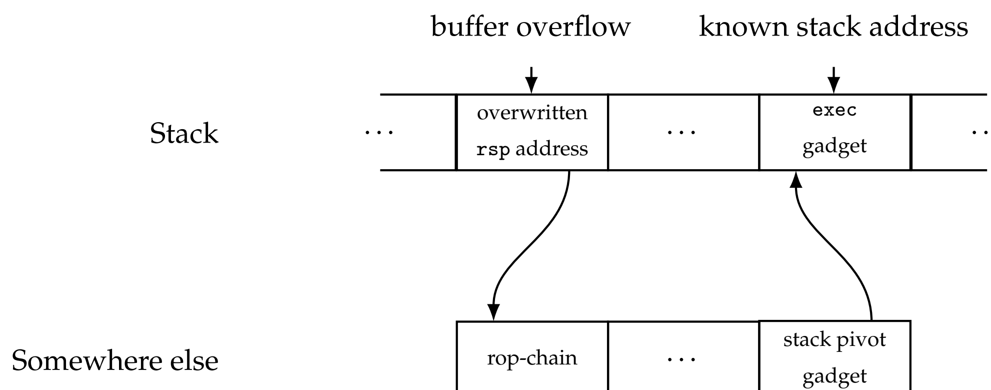


Figure 7: *Stack pinning counter attack*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	(18.0 ± 0.5) s	0.97 s	6	0 %
Patched, inactive	(18.4 ± 0.4) s	0.80 s	6	(3 ± 4) %
Patched, active	(18.4 ± 0.8) s	1.6 s	6	(2 ± 5) %

Table 2: *Stack pinning:* 2×10^8 `getpid` *system calls*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	$(2609 \pm 29) \times 10^1$ R s^{-1}	675 R s^{-1}	7	0 %
Patched, inactive	$(2750 \pm 22) \times 10^1$ R s^{-1}	474 R s^{-1}	6	(-5.4 ± 1.5) %
Patched, active	$(2720 \pm 14) \times 10^1$ R s^{-1}	295 R s^{-1}	6	(-4.3 ± 1.3) %

Table 3: *Stack pinning: ApacheBench with nginx, the unit is* requests per second, *hence the patched version was slightly faster*

kernel. The results, which are displayed in Table 2, show a performance loss of (3 ± 4) % compared to an unmodified kernel.

For a more realistic measurement, an application that does many system calls is necessary. We picked a patched *ApacheBench* that can enable stack pinning using the `prctl` syscall and a similarly patched nginx as a server. There are no regressions visible in this benchmark (see Table 3), the general uncertainty seems to be bigger than the impact of the stack pinning change, because the patched versions were faster than the unpatched version.

5.2 Randomize `mmap`

Security In a simple test, that does three allocations using `mmap`, the largest introduced gap in 1000 runs was `0x4 1762 14e4` pages big (= `0x4176 214e 4000` bytes). That means for an allocation of a single page (with only one other already allocated page), there is an entropy of 34 bit. It should be noted that the randomness shrinks after each allocation because the available gaps get smaller. For example the average size of the second gap is only half as big as the size of the first gap (`0x135c 70f8` compared to `0x2332 2b76` pages).

The higher bits of the random hole size are not evenly distributed as we can observe from Figure 8 (note the logarithmic scaling of the x axis). A linear scaling would compress the majority of the data points on the left side which makes it hard to spot something useful.

Figure 9 partitions the total range, in which gap sizes occur, into slots of 8 MiB. It displays the distribution of gap sizes, e. g. of 0 to 8 MiB occur with a probability of 16.6 % between the first and second page allocation. It is visible that smaller gaps occur much

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	(8.59 ± 0.04) s	0.043 s	3	0 %
Patched	(8.836 ± 0.026) s	0.033 s	3	(2.9 ± 0.5) %

Table 4: *Randomize* `mmap`*:* 2×10^7 *pages allocated with* `mmap`

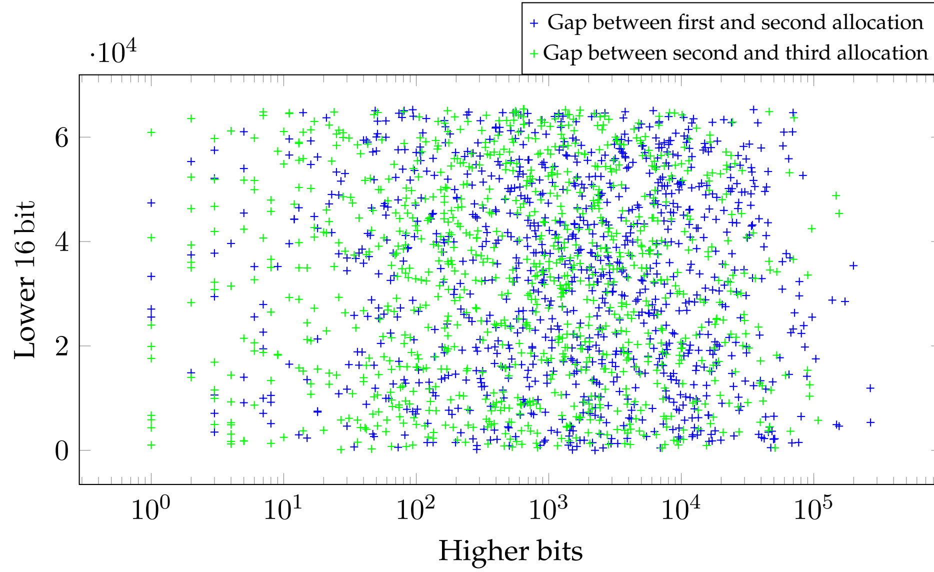


Figure 8: *Distribution of `mmap` gaps*

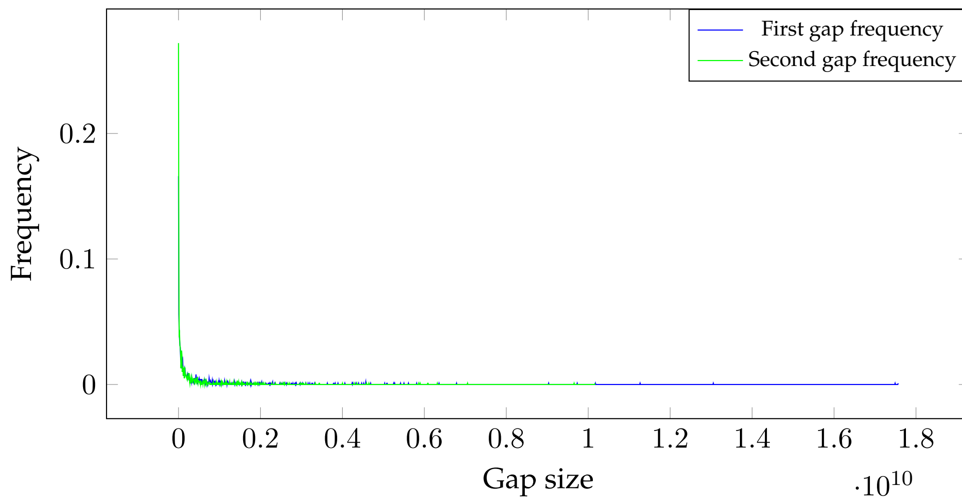


Figure 9: *Distribution of `mmap` gaps*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	(10.654 ± 0.028) s	0.036 s	3	0 %
Patched	(10.79 ± 0.11) s	0.14 s	3	(1.3 ± 1.1) %

Table 5: *Randomize `mmap`: 2×10^8 times allocating a 1024 byte block with `malloc`*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	(125.5 ± 1.6) s	2.0 s	3	0 %
Patched	(124 ± 4) s	4.1 s	3	(-1.0 ± 2.8) %

Table 6: *Randomize `mmap`: Compiling the linux 4.9 kernel with a normal and a patched kernel*

more often than large gaps as it can be observed in Figure 9. The median gap size is `0x071b 1c20` pages for the first gap and `0x02ce 7d01` pages for the second.

The randomness that is introduced by adding gaps between allocations makes it impossible for an attacker to deterministically compute locations of further mapped memory.

Performance The `mmap` microbenchmark (see Table 4), which executes just a loop of `mmaps` (or `mallocs`) and sets the first ten bytes of each allocation, shows a slight regression of $(2.9 \pm 0.5)\,\%$. For `malloc` (Table 5), the microbenchmark regressed by $(1.3 \pm 1.1)\,\%$. For normal, high workload such as compiling the linux kernel, no performance reduction could be measured as we can observe from Table 6.

5.3 Improving the Stack Smashing Protector

Security Clearing the SSP reduces the amount of canaries that are stored on the stack to the necessary minimum. Therefore it provides the maximal possible security against leaking canaries by uninitialized memory.

Performance For the microbenchmark, the tested program calls a function, which stores two bytes into a buffer, checks the SSP and returns, for 5×10^9 times. One approach that was tested, is clearing the SSP by a simple `mov` operation. Another way uses `xor` to zero the canary, but the approach using `xor` is slower by $(18.2 \pm 1.5)\,\%$ compared to not zeroing the SSP. Table 7 shows that the `mov` variant was slightly, but not significantly, faster than the non-zeroing method in three test runs ($(0.8 \pm 1.1)\,\%$).

We conclude that clearing the SSP is a useful prophylactic measure because it does not involve a noticable overhead, even in the microbenchmark that only tests the canary in a loop.

Security Securitywise, leaking a randomized SSP is not as critical as without randomizing canaries. When leaking one ore more SSPs e. g. by reading out of bounds of a buffer, the leaked canaries get useless because an attacker can just overwrite them with the now known value. All other canaries are still secure because the attacker cannot compute their value. This includes newly created protectors thus an attacker is not able to return from any other function that is protected by a SSP.

Leaking the global SSP or one used SSP and the corresponding random value (which enables the attacker to compute the global SSP) allows an attacker to break arbitrary canaries on the condition that he is also able to supply crafted "random values" on the SafeStack or that he knows the values that are stored at the respective positions.

Apart from the described advantages, executables are still vulnerable to attacks that do not touch the canary as described in subsection 2.4.

Performance When testing the performance of a randomized canary, ISAAC performed a lot faster than the `rdrand` hardware instruction for a source of randomness.

The microbenchmark in Table 8 revealed a huge performance impact when using `rdrand`. It was (20.4 ± 0.4) times slower than just SafeStack. ISAAC on the other side was only (2.65 ± 0.04) times slower. For the tested workload, the greatest impact of

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	(7.35 ± 0.07) s	0.091 s	3	0 %
mov	(7.295 ± 0.030) s	0.038 s	3	(-0.8 ± 1.1) %
xor	(8.69 ± 0.07) s	0.088 s	3	(18.2 ± 1.5) %

Table 7: *Clear the SSP: Call a protected function* 5×10^9 *times*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	(1.608 ± 0.017) s	0.022 s	3	0 %
gcc	(3.078 ± 0.018) s	0.023 s	3	(91.5 ± 2.3) %
ssp-strong	(2.030 ± 0.016) s	0.021 s	3	(26.3 ± 1.7) %
SafeStack	(2.36 ± 0.04) s	0.040 s	3	(47.1 ± 2.5) %
rdrand	(48.3 ± 0.7) s	0.84 s	3	$(290 \pm 6) \times 10^1$ %
isaac	(6.27 ± 0.05) s	0.057 s	3	(290 ± 5) %

Table 8: *Randomize the SSP: Call a protected function* 5×10^8 *times*

compiler options is activating `stack-protector-strong` ((26.3 ± 1.7) % slowdown compared to clang without options) and using `rdrand`. Enabling SafeStack (additionally to `stack-protector-strong`) impairs the speed by (16.4 ± 1.8) %.

We also tested the change for some more realistic workloads. 7-Zip (Table 9) shows a regression of (1 ± 1) % (compared to SafeStack), LAME MP3 Encoding (Table 10) (1.3 ± 0.4) %. The overall performance penalty of randomizing the SSP is low enough hence it can be useful.

More benchmarks can be found in the appendix.

On platforms that support the `rdrand` instruction it could still be useful if it is used to initialize the state vector of ISAAC which does not need to be done often.

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(2943 \pm 11) \times 10^1$ MIPS	139 MIPS	3	0 %
ssp-strong	$(2980 \pm 17) \times 10^1$ MIPS	215 MIPS	3	(-1.3 ± 0.7) %
SafeStack	$(2966 \pm 8) \times 10^1$ MIPS	101 MIPS	3	(-0.8 ± 0.5) %
rdrand	$(2945 \pm 13) \times 10^1$ MIPS	157 MIPS	3	(-0.1 ± 0.6) %
isaac	$(2926 \pm 27) \times 10^1$ MIPS	353 MIPS	3	(0.6 ± 1.0) %

Table 9: *Randomize the SSP: 7-Zip compression, the unit is* Million Instruction Per Second

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	(11.31 ± 0.04) s	0.076 s	5	0 %
ssp-strong	(11.23 ± 0.05) s	0.097 s	5	(-0.7 ± 0.6) %
SafeStack	(11.173 ± 0.025) s	0.048 s	5	(-1.2 ± 0.5) %
rdrand	(12.11 ± 0.05) s	0.093 s	5	(7.1 ± 0.6) %
isaac	(11.318 ± 0.028) s	0.053 s	5	(0.1 ± 0.5) %

Table 10: *Randomize the SSP: LAME MP3 Encoding*

6 Discussion

In this section we describe similar and related techniques and possible additions that can provide even more security.

6.1 Related Work

An overview over vulnerability and mitigation techniques, including some performance measurements was done at Meiji University [23]. Another thorough analysis of memory corruption bugs was done in *SoK: Eternal war in memory* [26]. In contrast, our work focuses on selected techniques, contains benchmark results and analyzes security mechanisms. Additionally we developed new methods to prohibit certain types of exploits.

Windows implemented a bounds check in system calls that manipulate memory and are often used in exploits. Our approach follows this way and adds a check for the stack pointer in every system call on linux. A remedy to stack pivoting with the same goal was developed at Syracuse University [20]. Their approach modifies the compiler generated code to insert stack pivoting checks at every absolute modification of the stack pointer. This will insert a check e. g. after each `leave` instruction. Relative modifications are not checked. The method uses another starting point (the compiler instead of the kernel) compared to our implementation. It has the advantage, that stack pivoting gets detected earlier, right at the point where an attacker tries to modify the stack pointer. Because of this, the main counter attack that was presented for Windows [22] and which also works for the method used in this work, is successfully thwarted.

SafeStack needs two stack pointers, one for the safe and one for the unsafe stack. This comes with runtime costs. Another proposed approach trades performance for memory consumption [28]. There, the different stacks have a fixed offset to each other. That means, every stack has the same size and each function frame is present on each stack. But the stored data is available on only one of the stacks. Stack addresses can be computed without additional runtime costs, just by adding a fixed offset to the address of each variable at compile time. This offset determines the stack, on which the respective variable is stored. The disadvantage of this approach is that the consumed memory area is larger since the different stacks have all the same size. An object that is allocated on one stack still reserves the same space on all other stacks (i.e. two stacks double the stack memory consumption). Compared to the approach of *SafeStack*, it has no performance overhead and is easily scalable to more than two stacks, but that comes at a high stack memory overhead.

The discussed `mmap` patch improves randomization of the address space to complicate successful address guesses by an attacker. *ASLR-Guard* also hardens the randomization of addresses by preventing leaks of code addresses [15]. To accomplish that, it takes a similar approach to *SafeStack* and creates multiple memory areas. Additionally, code pointers that are used as data get encoded, therefore their leak does not lead an attacker to code addresses. This is different from the `mmap` hardening as it works on a complete protection of code pointers while the `mmap` randomization improves hiding data addresses.

Clearing canaries after checking them is used to protect from leaking them, especially from uninitialized buffer reads. An efficient way to prevent data leaks in all cases by initializing all used memory is provided by *SafeInit* [18]. This compiler modification initializes all allocated memory with zeroes. Stack variables are also initialized when they come in scope. To reduce the performance overhead, compiler optimizations, which e. g. remove double stores without a read in between, are used. Because of these optimizations, the danger of uninitialized data can be eliminated with a speed loss of $<5\,\%$.

As mentioned in subsection 2.4, renewing the SSP value on process forks fixes one of the problems related to static canaries [16]. The described technique has the advantage that it can be used by preloading a shared library. Additionally it has less performance overhead than randomizing all used SSPs. One flaw of modifying the SSP on forks in the proposed way is that correct code is not guaranteed to work. Changing the global canary value means, that all newly written SSPs are using the new value. For checking the canaries, the new value is used. The problem is that functions, which were called before the fork, still used the old canary. Hence if the function that called `fork` returns, it will check the stored SSP (which is the old value) against the global SSP, which holds the new value. Therefore the process will abort, even if nobody tampered with the canary. While there is no program in our knowledge where the child process returns from the forking function, it is still important, that correct programs do not show invalid behaviour. At the cost of performance, this problem can be addressed [19]. The proposed remedy stores addresses to all SSPs which are currently saved on the stack. On a fork, the global canary and additionally all already stored values are modified. Our proposed method, that generates a random SSP per function call, also fixes the problem of forks, but it also includes overhead when no fork is made.

6.2 Future Work

Randomizing the SSP and using stack protectors in every function reduces the amount of available ROP gadgets because an attacker would always have to provide a valid canary. It would be even better if the canary was checked after restoring the registers and `rsp`, thus not even register setting gadgets are available to an attacker. This is difficult to implement in *llvm* because the stack frame allocation code is introduced late in the compilation process, much later than the creation of SSPs, thus it was not in the scope of this work.

The patches focus only the linux operating system and the llvm compilers. The ideas should be applicable to other compilers and operating systems too and should have a similar effect on speed.

6.3 Conclusion

Our goal is to make it more difficult for attackers to exploit existing bugs in software. The new techniques, that we introduced, support this goal because each of them makes circumventing security measures harder for an attacker.

The maximal speed loss of our changes, that we tested by microbenchmarks, is only $(3 \pm 3)\,\%$ for stack pinning and $(2.8 \pm 0.5)\,\%$ for `mmap` randomization. The only case, where it was a lot higher, is for random SSPs. Using the ISAAC cipher was (2.65 ± 0.04) times slower than only using SafeStack with `-fstack-protector-strong`. The performance impact on existing applications was not measurable for most of the introduced techniques. Only randomized canaries show a regression of $<2\,\%$, which we see as still acceptable for relevant applications.

References

[1] Steven Van Acker et al. *ValueGuard: Protection of Native Applications against Data-only Buffer Overflows*. 2010.

[2] Starr Andersen and Vincent Abella. *Memory Protection Technologies*. `https://technet.microsoft.com/en-us/library/bb457155.aspx`, visited 2017-06-12. 2004.

[3] Daniele Argento, Patrizio Boschi, and Luca Del Basso. *NX bit*. 2007.

[4] Donnie Berkholz. *Bypassing StackGuard and StackShield*. `http://redmonk.com/dberkholz/2014/05/02/github-language-trends-and-the-fragmenting-landscape/`, visited 2017-06-13. 2014.

[5] Gang Chen et al. *SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities*. 2013.

[6] J. Cohen. *RELRO: RELocation Read-Only*. 2011.

[7] Crispan Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *USENIX Security Symposium*. Vol. 7. 1998.

[8] Alan M. Frieze et al. *Reconstructing Truncated Integer Variables Satisfying Linear Congruences*. 1998.

[9] Google. *AdressSanitizer*. `https://github.com/google/sanitizers/wiki/AddressSanitizer`, visited 2017-06-14. 2011.

[10] Michael Howard. *Address Space Layout Randomization in Windows Vista*. `https://blogs.msdn.microsoft.com/michael_howard/2006/05/26/address-space-layout-randomization-in-windows-vista/`, visited 2017-06-12. 2006.

[11] *John the Ripper FAQ*. `http://www.openwall.com/john/doc/FAQ.shtml`, visited 2017-07-12. 2015.

[12] Robert J. Jenkins Jr. *ISAAC*. `http://www.burtleburtle.net/bob/rand/isaac.html`, visited 2017-06-18. 1993.

[13] Steve Langasek. *Ubuntu Foundations Team - Weekly Newsletter, 21017-06-15*. `https://lists.ubuntu.com/archives/ubuntu-devel/2017-June/039816.html`, visited 2017-06-16. 2017.

[14] David Litchfield. *Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform*. 2005.

[15] Kangjie Lu et al. *ASLR-Gurd: Stopping Address Space Leakage for Code Reuse Attacks*. 2015.

[16] Hector Marco-Gisbert and Ismael Ripoll. *Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers*. 2013.

[17] Marcus Meissner. *[opensuse-factory] openSUSE Tumbleweed now full of PIE*. `https://lists.opensuse.org/opensuse-factory/2017-06/msg00403.html`, visited 2017-06-17. 2017.

[18] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. *SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities*. 2017.

[19] Theofilos Petsios, Vasileios P. Kemerlis, and Michalis Polychronakis. *DynaGuard: Armoring Canary-based Protections against Brute-froce Attacks*. 2015.

[20] Aravind Prakash and Heng Yin. *Defeating ROP Through Denial of Stack Pivot*. 2015.

[21] William Roberts. *Introduce mmap randomization*. `https://patchwork.kernel.org/patch/9248669/`, visited 2017-06-12. 2016.

[22] Dan Rosenberg. *Defeating Windows 8 ROP Mitigation*. `http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/`, visited 2017-06-12. 2011.

[23] Takamichi Saito et al. "A Survey of Prevention/Mitigation against Memory Corruption Attacks". In: *International Conference on Network-Based Information Systems*. Vol. 19. 2016.

[24] Hovav Shacham et al. *On the Effectiveness of Address-Space Randomization*.

[25] Ian Skerrett. *Profile of an IoT Developer: Results of the IoT Developer Survey*. `https://ianskerrett.wordpress.com/2016/04/14/profile-of-an-iot-developer-results-of-the-iot-developer-survey/`, visited 2017-06-13. 2016.

[26] László Szekeres et al. *SoK: Eternal War in Memory*. 2013.

[27] Crypto++ Wiki. *RDRAND*. `https://www.cryptopp.com/wiki/RDRAND`, visited 2017-07-02. 2017.

[28] Yves Younan et al. *Extended protection against stack smashing attacks without performance loss*. 2006.

Acronyms

ASan AdressSanitizer.

ASLR Address Space Layout Randomization.

CSPRNG Cryptographically Secure Pseudo-Random Number Generator.

DEP Data Execution Prevention.

GOT Global Object Table.

IoT Internet Of Things.

LCG Linear Congruential Generator.

NX No eXecute.

PIE Position Independent Executable.

PRNG Pseudo-Random Number Generator.

RELRO RELocation Read-Only.

ROP Return Oriented Programming.

SSP Stack Smashing Protector.

Appendix

Randomize `mmap`

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	$(26.8 \pm 0.6)\,\mathrm{s}$	$1.1\,\mathrm{s}$	6	$0\,\%$
Patched	$(25.7 \pm 1.1)\,\mathrm{s}$	$2.4\,\mathrm{s}$	6	$(-4 \pm 5)\,\%$

Table 11: *t-test1, 1 thread*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
Unpatched	$(7.03 \pm 0.05)\,\mathrm{s}$	$0.058\,\mathrm{s}$	3	$0\,\%$
Patched	$(7.019 \pm 0.028)\,\mathrm{s}$	$0.036\,\mathrm{s}$	3	$(-0.2 \pm 0.8)\,\%$

Table 12: *t-test1, 2 threads*

Randomizing the Stack Smashing Protector

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$122\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$0\,\%$
ssp-strong	$116\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$5\,\%$
SafeStack	$(115.3 \pm 0.9)\,\mathrm{I\,min^{-1}}$	$1.2\,\mathrm{I\,min^{-1}}$	3	$(5.5 \pm 0.8)\,\%$
rdrand	$(114.3 \pm 0.9)\,\mathrm{I\,min^{-1}}$	$1.2\,\mathrm{I\,min^{-1}}$	3	$(6.3 \pm 0.8)\,\%$
isaac	$(114.7 \pm 0.9)\,\mathrm{I\,min^{-1}}$	$1.2\,\mathrm{I\,min^{-1}}$	3	$(6.0 \pm 0.8)\,\%$

Table 13: *GraphicsMagick: Blur, the unit is* Iterations per minute

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(115.7 \pm 0.5)\,\mathrm{I\,min^{-1}}$	$0.58\,\mathrm{I\,min^{-1}}$	3	$0\,\%$
ssp-strong	$113\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$(2.3 \pm 0.4)\,\%$
SafeStack	$114\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$(1.4 \pm 0.4)\,\%$
rdrand	$(113.7 \pm 0.5)\,\mathrm{I\,min^{-1}}$	$0.58\,\mathrm{I\,min^{-1}}$	3	$(1.7 \pm 0.6)\,\%$
isaac	$114\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$(1.4 \pm 0.4)\,\%$

Table 14: *GraphicsMagick: Sharpen*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(156.7 \pm 1.2)\,\mathrm{I\,min^{-1}}$	$1.5\,\mathrm{I\,min^{-1}}$	3	$0\,\%$
ssp-strong	$154\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$(1.7 \pm 0.8)\,\%$
SafeStack	$(154.7 \pm 0.9)\,\mathrm{I\,min^{-1}}$	$1.2\,\mathrm{I\,min^{-1}}$	3	$(1.3 \pm 1.0)\,\%$
rdrand	$(154.0 \pm 0.8)\,\mathrm{I\,min^{-1}}$	$1.0\,\mathrm{I\,min^{-1}}$	3	$(1.7 \pm 0.9)\,\%$
isaac	$(154.3 \pm 1.2)\,\mathrm{I\,min^{-1}}$	$1.5\,\mathrm{I\,min^{-1}}$	3	$(1.5 \pm 1.1)\,\%$

Table 15: *GraphicsMagick: HWB Color Space*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(89.7 \pm 0.5)\,\mathrm{I\,min^{-1}}$	$0.58\,\mathrm{I\,min^{-1}}$	3	$0\,\%$
ssp-strong	$87\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$(3.0 \pm 0.5)\,\%$
SafeStack	$(87.3 \pm 0.5)\,\mathrm{I\,min^{-1}}$	$0.58\,\mathrm{I\,min^{-1}}$	3	$(2.6 \pm 0.7)\,\%$
rdrand	$82\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$(8.6 \pm 0.5)\,\%$
isaac	$85\,\mathrm{I\,min^{-1}}$	$0\,\mathrm{I\,min^{-1}}$	3	$(5.2 \pm 0.5)\,\%$

Table 16: *GraphicsMagick: Local Adaptive Thresholding*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(11.61 \pm 0.22)\,\mathrm{s}$	$0.36\,\mathrm{s}$	4	$0\,\%$
ssp-strong	$(11.12 \pm 0.25)\,\mathrm{s}$	$0.54\,\mathrm{s}$	6	$(-4.2 \pm 2.8)\,\%$
SafeStack	$(10.86 \pm 0.11)\,\mathrm{s}$	$0.14\,\mathrm{s}$	3	$(-6.5 \pm 2.0)\,\%$
rdrand	$(12.69 \pm 0.15)\,\mathrm{s}$	$0.19\,\mathrm{s}$	3	$(9.3 \pm 2.4)\,\%$
isaac	$(12.27 \pm 0.21)\,\mathrm{s}$	$0.27\,\mathrm{s}$	3	$(5.7 \pm 2.7)\,\%$

Table 17: *FFmpeg: H.264 HD To NTSC DV*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(843 \pm 19) \times 10^{1}\,\mathrm{C\,s^{-1}}$	$411\,\mathrm{C\,s^{-1}}$	6	$0\,\%$
ssp-strong	$(81 \pm 4) \times 10^{2}\,\mathrm{C\,s^{-1}}$	$686\,\mathrm{C\,s^{-1}}$	6	$(4 \pm 5)\,\%$
SafeStack	$(8791 \pm 16)\,\mathrm{C\,s^{-1}}$	$20\,\mathrm{C\,s^{-1}}$	3	$(-4.2 \pm 2.4)\,\%$
rdrand	$(872 \pm 6) \times 10^{1}\,\mathrm{C\,s^{-1}}$	$75\,\mathrm{C\,s^{-1}}$	3	$(-3.4 \pm 2.4)\,\%$
isaac	$(857 \pm 21) \times 10^{1}\,\mathrm{C\,s^{-1}}$	$266\,\mathrm{C\,s^{-1}}$	3	$(-2 \pm 4)\,\%$

Table 18: *John The Ripper: Blowfish, the unit is means* combinations of candidate password and target hash per second *[11]*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(259 \pm 8) \times 10^5\,\mathrm{C\,s^{-1}}$	$1\,655\,184\,\mathrm{C\,s^{-1}}$	6	$0\,\%$
ssp-strong	$(259 \pm 13) \times 10^5\,\mathrm{C\,s^{-1}}$	$2\,831\,141\,\mathrm{C\,s^{-1}}$	6	$(0 \pm 6)\,\%$
SafeStack	$(272\,016 \pm 22) \times 10^2\,\mathrm{C\,s^{-1}}$	$2886\,\mathrm{C\,s^{-1}}$	3	$(-5 \pm 4)\,\%$
rdrand	$(271 \pm 4) \times 10^5\,\mathrm{C\,s^{-1}}$	$456\,711\,\mathrm{C\,s^{-1}}$	3	$(-5 \pm 4)\,\%$
isaac	$(256 \pm 7) \times 10^5\,\mathrm{C\,s^{-1}}$	$1\,494\,503\,\mathrm{C\,s^{-1}}$	6	$(1 \pm 4)\,\%$

Table 19: *John The Ripper: Traditional DES*

Configuration	Avg ± SEM	SD	Samples	Rel. perf. loss
clang	$(105 \pm 4) \times 10^3\,\mathrm{C\,s^{-1}}$	$8798\,\mathrm{C\,s^{-1}}$	6	$0\,\%$
ssp-strong	$(1133 \pm 12) \times 10^2\,\mathrm{C\,s^{-1}}$	$1527\,\mathrm{C\,s^{-1}}$	3	$(-7 \pm 5)\,\%$
SafeStack	$(107 \pm 5) \times 10^3\,\mathrm{C\,s^{-1}}$	$9900\,\mathrm{C\,s^{-1}}$	6	$(-2 \pm 6)\,\%$
rdrand	$(1151 \pm 8) \times 10^2\,\mathrm{C\,s^{-1}}$	$1000\,\mathrm{C\,s^{-1}}$	3	$(-9 \pm 5)\,\%$
isaac	$(107 \pm 5) \times 10^3\,\mathrm{C\,s^{-1}}$	$10\,123\,\mathrm{C\,s^{-1}}$	6	$(-2 \pm 6)\,\%$

Table 20: *John The Ripper: MD5*