Karlsruhe Institute of Technology

Towards a new low-overhead protection mechanism against non-control data attacks

Master Thesis

Katharina Männle

Department of Informatics
Karlsruhe Institute of Technology

Erstgutachter:	PD. Dr. Ingmar Baumgart
Zweitgutachter:	Prof. Dr. Ralf Reussner
Betreuende Mitarbeiter:	M. Sc. Marek Wehmer
	Dipl.-Inform. Jochen Rill

Time for Completion: 01. February 2018 – 31. August 2018

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 31. August 2018

Abstract

With the deployment of mitigations to protect against control flow attacks including NX, Stack Canaries, ASLR, and recently Control Flow Integrity, non-control data attacks are becoming an increasingly relevant technique for the exploitation of memory corruption vulnerabilities. Instead of modifying code pointers to gain direct control over the instruction pointer, non-control data attacks corrupt key data structures to manipulate the programs behavior in a way that benefits the attacker. The goal of this thesis is to investigate the impact of non-control data attacks and to introduce a mechanism that allows developers to protect critical data structures such that they cannot be corrupted by an attacker with the ability to read and write to arbitrary memory addresses. This is accomplished through the use of page-level memory permissions to mark critical data as read-only. A main focus of the mitigation lies on low overhead for read access, as critical data structures tend to be read frequently but only written rarely. To allow creation of protected data structures at runtime, the mitigation is implemented in the form of a secure heap allocator that stores its data in the protected memory region. Furthermore, this work evaluates conditions in which the introduced mitigation might fail and presents instructions on how to avoid their occurrences and ensure a secure application of the allocator. As expected, the performance evaluation shows that the overhead for read access to protected data is negligible, indicating that the mitigation can be applied in real-world scenarios.

Deutsche Zusammenfassung

Programmierfehler, die zur Korruption von Speicher führen können, zählen zu den am meisten ausgenutzten Software Verwundbarkeiten. Um vor der Ausnutzung dieser Schwachstellen zu schützen, wurden im Laufe der Jahre mehrere Gegenmaßnahmen eingeführt. Zu diesen Maßnahmen zählen *Stack Canaries*, *Data Execution Prevention (DEP)*, und *Address Space Layout Randomization (ASLR)*. Ein aktueller Forschungsschwerpunkt sind sogenannte *Control Flow Integrity (CFI)* Mechanismen, die verhindern sollen, dass ein Angreifer überhaupt den Kontrollfluss eines Programmes korrumpieren kann. Microsoft führte 2014 das erste weit verbreite CFI Schema *Control Flow Guard (CFG)* ein, gefolgt von LLVM für den *clang* Compiler.

Da Sicherheitsschwachstellen oft nicht prinzipiell verhindert werden können, sollen diese sogenannten Mitigations es Angreifern erschweren, Verwundbarkeiten böswillig auszunutzen. All diese Mitigations konzentrieren sich jedoch hauptsächlich darauf, Kontrolldaten, wie Zeiger auf Funktionen oder Rücksprungadressen zu schützen. Wenig Fokus liegt auf dem Schutz von Nichtkontrolldaten, die durch die immer zunehmende Schwierigkeit von Kontrolldatenmanipulation an Relevanz gewinnen. Sicherheitskritische Nichtkontrolldaten, wie zum Beispiel Befehlspuffer, können vom Angreifer einfach manipuliert werden, ohne dass bisherige Mitigations davor schützen. So kann ein Angreifer auf einfache Weise beispielsweise seine Privilegien erweitern oder sogar beliebigen Code ausführen.

Diese Arbeit analysiert die Relevanz von Nichtkontrolldatenangriffen indem Beispiele für kritische Daten aufgeführt werden und Wege diese böswillig zu missbrauchen betrachtet werden. Weiterhin wird ein Mechanismus entwickelt um Nichtkontrolldaten vor Manipulation zu schützen. Die Grundidee dieser Mitigation ist es, sich Zugriffsberechtigungen von Speicherseiten zu Nutze zu machen. Dabei werden Speicherseiten mit kritischen Daten nur lesbar, aber nicht schreibbar in den Speicher eingebunden. Da kritische Daten potentiell zur Laufzeit angelegt werden und deren Größe nicht statisch bekannt ist, wird ein Heap Allocator implementiert, der die typische *malloc()* und *free()* Schnittstelle bereitstellt, Daten aber auf einem separaten, gesicherten Heap allokiert. Gleichzeitig muss es jedoch auch möglich sein, legitime Änderungen an den Daten von Seiten des Programms zuzulassen. Dafür wird ein Mechanismus eingeführt, der es erlaubt die Daten kurzzeitig schreibbar einzubinden bis die Änderungen vollzogen sind.

Der Hauptfokus der Mitigation liegt darauf, den Lesezugriff auf geschützte Daten mit geringen Zusatzkosten zu ermöglichen, da diese oftmals selten geschrieben, aber oft gelesen werden. Um dies zu evaluieren wurde eine Prototypimplementierung der Mitigation umgesetzt. Die Evaluierung der Performance hat ergeben, dass die Zusatzkosten für Lesezugriff vernachlässigbar sind. Weiterhin wurde eine Sicher-

heitsevaluierung durchgeführt. In dieser wurden verschieden Szenarien analysiert, die dazu führen können, dass die Mitigation keinen Schutz für kritische Daten garantieren kann. Daraufhin wurde diskutiert, wie diese Situationen verhindert werden können, indem korrekte Anwendungsmuster dargelegt wurden. Alles in allem ermöglicht es die Mitigation Daten gezielt vor Korruption zu schützen. Bei Multi-Threaded Anwendungen können jedoch Wettlaufsituationen auftreten, die ein Angreifer ausnutzen kann. Die Wahrscheinlichkeit dies erfolgreich umzusetzen wird jedoch minimiert indem die Einbindung von schreibbaren Daten in den Speicher randomisiert wird, sodass der Angreifer nicht weiß wo diese liegen. Weiterhin hat der Angreifer meist nur eine Chance, da ein misslungener Angriff in einem Speicherzugriffsfehler resultiert und damit zum Abstürzen des Programmes führt.

Contents

1. Introduction

A necessary part of securing modern IT infrastructure is the protection against software vulnerabilities, as these may allow an attacker to gain initial access to the network or move laterally in it, obtaining access to critical data or interrupting business processes. To protect against such vulnerabilities, it is first necessary to study the source of their occurrence as well as the ways in which they can be exploited by an attacker. Out of the many different types of programming errors, memory corruption vulnerabilities are one of the most exploited ones.

To counter exploitation techniques, many mitigation mechanisms such as Stack Canaries, Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) have been introduced and implemented [VDSCB12]. Most research so far was aimed at preventing execution of arbitrary code and recent research focuses on mechanisms to provide Control Flow Integrity (CFI) in order to prevent attackers from manipulating a program's control flow [ABEL09][CBPW+15][KSPC+14]. Microsoft successfully implemented and deployed a CFI mechanism called Control Flow Guard (CFG), which is available on current Windows distributions. As arbitrary code execution becomes more difficult to achieve, non-control data attacks move into focus. With these, an attacker only modifies key data structures in a program and is able to gain more privileges or even code execution without ever changing control flow data like code pointers. A study analyzing the significance of non-control data attacks concluded that they have become a realistic threat [CXSG+05].

The goal of this thesis is to analyze the impact of non-control data attacks and to introduce a mechanism for protecting security-critical non-control data from manipulation, while inducing little overhead for read accesses to protected data. A prototype of the proposed mitigation will be implemented and evaluated to prove its feasibility.

1.1 Outline

This thesis begins by explaining background information needed to understand the requirements and components of the new mitigations in chapter 2. This includes a brief introduction to memory corruption vulnerabilities and an explanation of the

typical process memory layout and its different regions in order to understand the exploitation techniques presented in the next sections. Furthermore, a short history of the mitigations implemented to hinder the exploitation of memory corruption vulnerabilities will be given.

It then goes on to examine the impact of non control data attacks by outlining example targets and the importance of protecting them from corruption in chapter 3. This chapter will also give a summary about previous research on this topic, covering protection mechanisms for kernel and user space.

Next, the design of the new mitigation will be illustrated in chapter 4. The basic idea of the protection mechanism will be explained followed by a detailed overview of the components of the implementation. Finally, this chapter sums up the interface that will be provided for protecting critical data.

Finally, the implementation is evaluated regarding its performance and security properties in chapter 5. The first part of this chapter will discuss insecure usage patterns as well as problems that might still occur when applying the mitigation. The second part evaluates the performance of the mitigation. Ultimately, this work concludes with a discussion of future work.

2. Background

This chapter starts with a brief introduction to memory corruption vulnerabilities by first explaining the typical memory layout of a process and subsequently showing how to exploit such a vulnerability to gain code execution at the example of a stack-based buffer overflow in section 2.1. It will conclude with a short history of memory corruption, outlining different exploitation techniques and the mitigations developed to counter them in section 2.2.

2.1 Memory Corruption

Memory corruption is a class of software vulnerabilities in which program memory is modified in an unintended way. Although usually harder to exploit than other vulnerability classes such as injections, attackers are often able to abuse these issues for their benefit. To counter the occurrence of memory errors, memory-safe languages such as Java have been developed, which, however, come with a non-negligible performance overhead. Another approach taken is to build mitigations that do not stop memory errors from occurring, but instead make it more difficult for an attacker to exploit them in a harmful way.

Memory errors occur as result of programming mistakes in memory unsafe languages such as C and C++. Typical vulnerabilities found there are uninitialized memory, out-of-bounds memory access, type confusions and race conditions. Exploited by an attacker, these vulnerabilities can lead to information leaks, privilege escalation or arbitrary code execution. This section shortly explains how memory errors can be exploited by firstly outlining the typical process memory layout in section 2.1.1 and afterwards illustrating the mechanism of stack frames for function calls in section 2.1.2, which will be essential to understand the exemplary exploitation technique explained in section 2.2

2.1.1 Process Memory Layout

Each process is assigned a distinct virtual address space by the underlying operating system, which is divided into multiple regions that can hold different access permissions: readable, writable and executable. The following section depicts the

typical layout of a process' address space at the example of a Linux executable. The arrangement of the regions can slightly vary on different operating systems, the basic division into the various regions, however, is similar.

Generally, a process' virtual memory is divided into two areas with different privilege levels. While the *User Address Space* contains data used and managed by the process itself, the *Kernel Address Space* may only be accessed by the operating system kernel. If a process needs to perform a privileged operation such as opening network connections, it performs a system call and execution is transferred to the kernel which operates in its own address space. The process can neither read nor write to memory owned by the kernel or another process. Figure 2.1 illustrates the memory layout alongside explanations of the different regions.

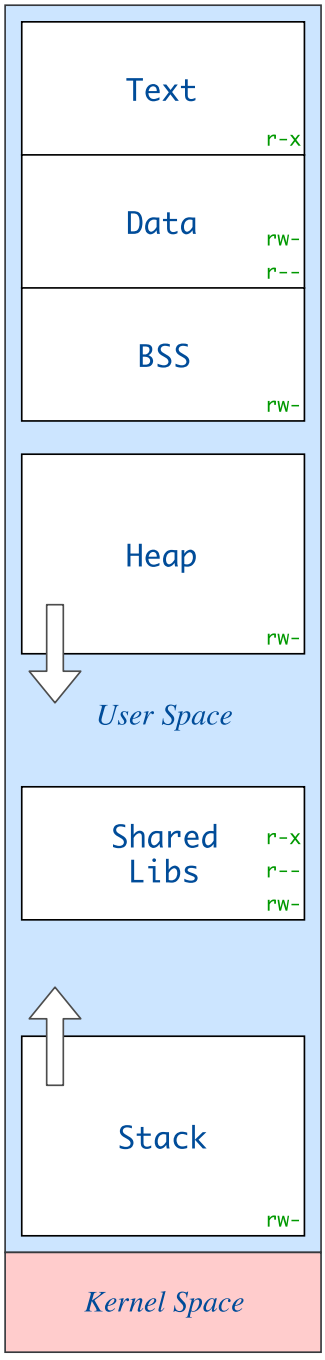

Figure 2.1: Virtual Memory Layout

Text The text or code segment contains the executable instructions of the binary. It is mapped executable and readable but not writable to prevent illegal modifications of instructions during runtime.

Data The data segment contains data initialized at compile time. These include both constant and modifiable global and static variables, which are respectively allocated in a read-only or read- and writable section. The total size of the data structures stored here must be known at compile time to reserve the required space.

BSS Contrary to the data segment, the BSS segment stores uninitialized data. The variables will only be initialized with their intended value at runtime and must therefore be both read- and writable. Just like in the data segment the size of the data structures must be known at compile time for statically allocating the memory.

Shared Libraries Shared libraries provide functionality in a separate file that can be imported by a program by mapping it into its address space and processing its exported functions and other symbols.

Stack The call stack provides a mechanism to store local variables that only exist during the execution of a function. At the beginning of each function call, a frame on the stack is reserved for local variables. This works on a Last In First Out basis: the current function's stack frame is pushed onto the top of the stack and removed when returning to the caller. To determine the required size of the stack frame, the size of all stack variables has to be known at compile time. On Linux, the stack is situated at the end of the user address space and grows towards lower addresses. Further details about the functionality of the stack are explained in section 2.1.2.

Heap The heap contains variables whose size is not known at runtime and can therefore not be statically allocated in the stack, BSS, or data segment. The memory required for these variables must thus be dynamically allocated at runtime. For that purpose, a heap allocator is used, which provides an interface for allocating memory with a specified size and freeing the requested memory if the data structure is no longer needed. The same heap segment is shared with all shared libraries mapped into the process's address space. Since the size of the heap is dynamic, the size of the heap segment cannot be determined at compile time. If the reserved heap size is not sufficient, the heap allocator can request further memory from the kernel and release it if it is no longer requires. The heap commonly grows towards higher memory addresses.

2.1.2 Function Calls

This section gives a more detailed explanation of the call stack mechanism introduced above with the help of an illustrative code sample.

The code snippet in listing 2.1 shows the *main()* function of a program calling another function *f(int a, int b)* that adds two integer arguments and returns the result. As explained in section 2.1.1, the program allocates a frame on the stack for each function call. This example will accordingly first allocate a frame for the *main()* function and then another frame for the call of *f(1, 2)*.

```
1  int f(int a, int b) {
2      int c;
3      c = a + b;
4      return c;
5  }
6
7  int main(int argc, char* argv[]) {
8      int x;
9      x = f(1, 2);
10     return x;
11 }
```

Listing 2.1: Function calls

Figure 2.2 illustrates this process. On the left-hand side, the stack frame is depicted at the beginning of the *main()* function. The stack frame includes the local variable x created in line 8 of listing 2.1. The start of the topmost stack frame, the so-called *stack pointer (SP)*, is stored in a special register which is updated at each function call and return. The *base pointer (BP)* indicates the end of the topmost stack frame. Another important value on the stack is the function's return address which stores at which address the execution must be continued when the function returns.

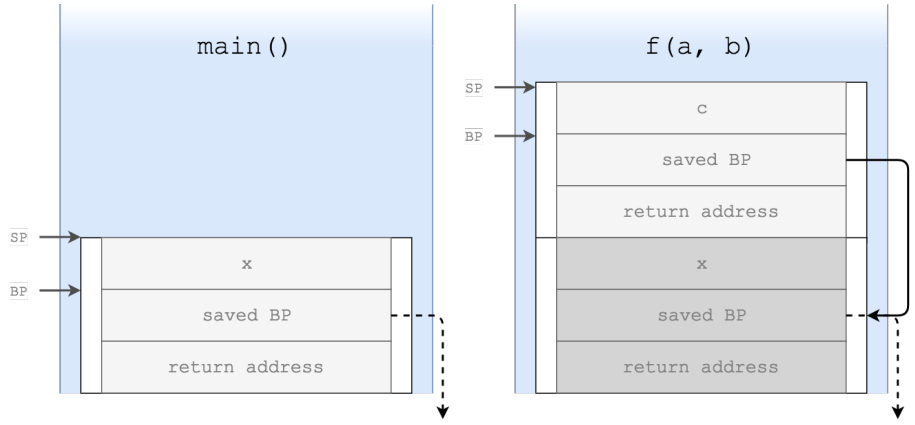

Figure 2.2: Function Stack Frames

This mechanism can best be explained by looking at the stack frame of *f()* on the right-hand side. When the call to *f(1, 2)* in line 9 is executed, the current instruction pointer is stored on the stack as the return address. Furthermore, the so called function prologue is executed, reserving space for the local variable c declared in line 2. This is achieved by subtracting the required frame size from the stack pointer. Additionally, the previous base pointer is stored on the stack, so it can be restored later, before the base pointer is adapted to point to the end of the new stack frame. Upon finishing execution, the previous state of the stack must be restored. First, the base pointer is moved into the stack pointer, essentially cleaning up all local variables in the current stack frame. Next the saved base pointer is restored from the stack. Finally the function returns to its caller by popping the saved return address from the stack and jumping to that address. The stack pointer now points to the start of the *main()* function's frame again. These 3 steps are commonly called the function epilogue.

2.2 A Short History of Memory Corruption

This chapter gives a brief overview of various mitigations introduced to make the exploitation of memory error vulnerabilities more difficult. In order to illustrate the incentive behind each mitigation a classic vulnerability, the stack buffer overflow, and its exploitation without the application of mitigations will be explained. Next, the most important mitigations and how they affect the presented exploitation technique will be listed. Apart from the presented stack buffer overflow, further vulnerabilities

exist and are frequently exploited. Amongst these are heap-based buffer overflows to corrupt both application data and heap meta data, format string bugs, which are nowadays mostly detected at compile time, integer overflows, and many more.

Example: Stack Buffer Overflow

Following up on section 2.1.2 this paragraph illustrates a basic example of the exploitation of a stack buffer overflow. This example shows how such a vulnerability can lead to arbitrary code execution. A buffer overflow commonly occurs when a programmer omits or fails in doing proper bounds checking on accessing a fixed size buffer located on the stack. As a result, an attacker is able to write memory beyond the intended buffer and overwrite other variables stored next to it. As illustrated in section 2.1.2 a function's return address is stored next to local variables in the stack frame. By overwriting this address, an attacker can manipulate the control flow.

```
1  void g(char* input) {
2      char buf[10];
3      int someVariable;
4      int anotherVariable;
5      ...
6      strcpy(buf, input);
7      ...
8  }
```

Listing 2.2: Function vulnerable to a buffer overflow

Listing 2.2 shows a function *g()* which is vulnerable to a stack buffer overflow if the argument is controlled like for example when user input is processed. The input is copied with *strcpy()* into a buffer of fixed size ten without checking the actual size of the input string. If said string is larger than ten bytes, variables that are adjacent to *buf* in memory can be overwritten. Furthermore, the return address can be manipulated. The stack frame layout of *g()* is shown in figure 2.3 on the left-hand side.

The right-hand side shows the exploitation of the buffer overflow. By submitting an input string that is to large, the attacker overwrites the adjacent variables including the return address. A common goal for an attacker is to spawn a Unix shell on the vulnerable machine. For this purpose, the attacker has to place shellcode in memory and make the return address point to it. The term shellcode describes a piece of machine code that starts a command shell when it is executed. In this scenario, the shellcode can be easily written onto the stack before or behind the return address by appropriately adapting the input to the function *g()*. By overwriting the return address to point to the start of the shellcode, the shellcode will be executed as soon as the function returns and a shell will be spawned instead of the intended continuation of the program. Placing the shellcode directly on the stack allows for an easy exploitation of buffer overflows.

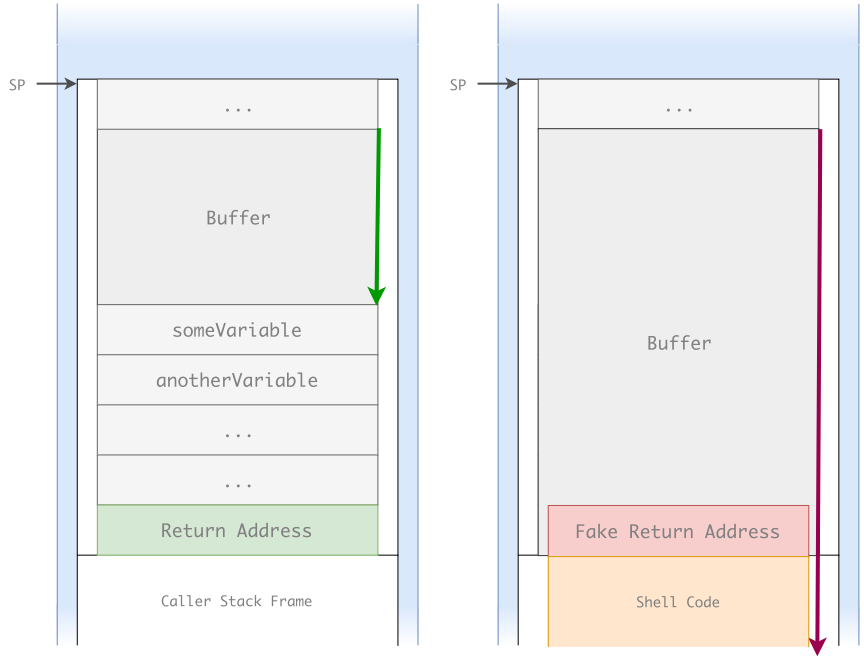

Figure 2.3: Overflow of a fixed-size buffer on the stack

2.2.1 NX Stack

Historically the stack segment was mapped into memory as read-, write- and executable. While the read and write permissions are necessary for the stack's functionality, the executable permission has no enhancing purpose. On the contrary, it can lead to an easy and severe exploit primitive for buffer overflow programming errors. The first countermeasure against the described attack was to remove the executable permission from the stack segment as well as from other memory regions where it was not required. This mitigation is commonly called NX (No-eXecute) or DEP (Data-Execution Prevention) and was first introduced in 1997. As the stack segment does not hold any code that would legitimately be executed, this does not affect the intended program behavior, but prevents an attacker from simply injecting shell code. Even though shellcode can no longer be executed on the stack, bypasses to *NX* have quickly been discovered and will be described next.

Return-To-Libc and Return Oriented Programming (ROP)

The basic idea behind bypassing *NX* is not to inject own shellcode but instead to reuse existing code that is already mapped executable. Almost every program has linked a C standard library, which provides useful type and function definitions. Amongst these functions is the *system()* function which can be called to execute shell commands. In order to spawn a shell on Linux the attacker has to call the function with the following argument: *system("/bin/sh")*. Therefore, the return address must be corrupted to point to the address of *system()* in the C standard library, that is mapped into the address space of the process. Furthermore, the argument for the function must be provided. On 32-bit x86 systems the arguments to functions are usually passed on the stack. The attacker can simply write a pointer to the string *"/bin/sh"* to the location above the return address which will be treated as the first argument when *system* is executed. On 64-bit systems the first arguments of

a function are commonly passed in registers instead, which requires further steps to load the string to the right register. This can be achieved with the technique described next.

A more general approach to the Return-To-Libc technique is called *Return Oriented Programming (ROP)*. This technique not only calls preexisting functions, but also uses arbitrary code snippets, called gadgets, that are located in existing executable memory regions. More precisely, the gadgets consist of mostly short instruction sequences that end with a return instruction. Listing 2.3 shows two exemplary ROP gadgets that can be used to control the content of registers. Since every gadget ends in a return instruction, execution will continue at the address at the top of the stack which is controlled by the attacker. As such multiple gadgets can be executed successively by writing their addresses onto the stack. By chaining different gadgets an attacker can potentially execute arbitrary code and spawn a shell. More information can be found in the paper by Hovav Shacham [Shac07], who first published the technique in 2007. ROP has been shown to be Turing-complete [TEBJ+11].

```
1 0x000000000045c4c1 pop eax; ret;
2 0x000000000049b40e mov edi, ebx; pop rbx; ret;
```

Listing 2.3: Two ROP gadget examples

2.2.2 Stack Canaries

The next mitigation introduced in 1998 aims at detecting a stack buffer overflow and the corruption of the return address, thus, preventing and attacker from using the techniques explained above. The so-called canary, a value which is randomly chosen at program start, is stored next to the return address as shown in figure 2.4. An attacker overwriting the stack buffer and corrupting the return address, would now overwrite the canary value as well. On function return, however, the canary is compared to its initial value and if the comparison fails an error is raised and the program is terminated in a controlled fashion. This mitigation can successfully stop exploitation of stack-based buffer overflows, however in certain circumstances it can still be bypassed, for example if the attacker can leak the canary value or if the overflow is non-linear and can thus corrupt the return address without modifying the canary.

2.2.3 Address Space Layout Randomization (ASLR)

The exploitation techniques demonstrated above rely heavily on the attacker knowing the address of the stack, libraries or text segments. The next mitigation, Address Space Layout Randomization (ASLR) introduced in 2001, randomizes the base addresses of each memory segment and maps them to a different position during each execution of a program. In order to exploit vulnerabilities as before, the attacker now needs an additional information leak to find out where the segments are stored in memory, which is a further obstacle. This might make exploitation less reliable or even impossible if the exact location is not known.

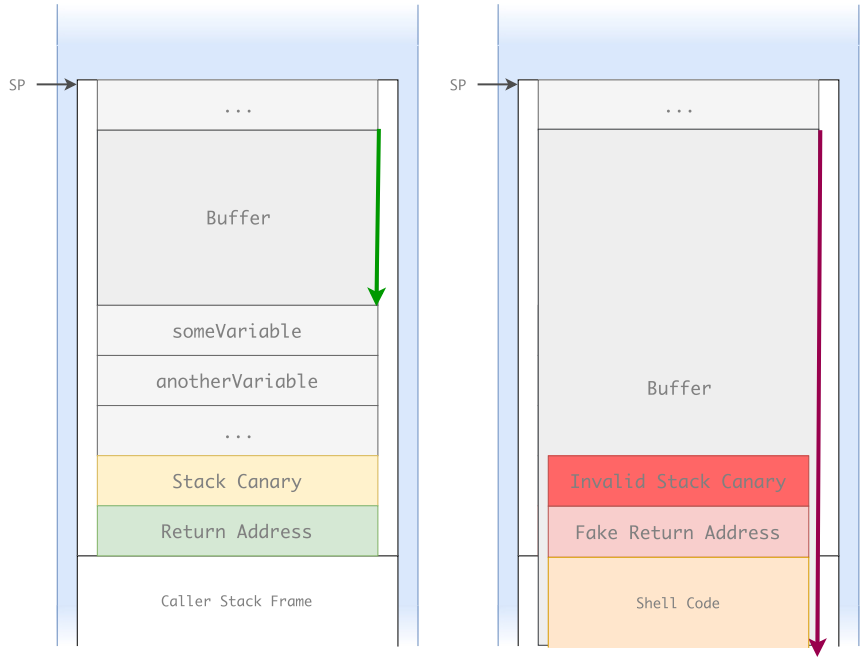

Figure 2.4: Stack Layout with Canary

2.2.4 Control Flow Integrity (CFI)

The most recent mitigation from 2014 presented in this thesis is Control Flow Integrity (CFI). The goal of CFI is to protect the program's control flow from being modified. CFI is currently under active research [ABEL09] [TRCC+14] [CBPW+15] [Bial18] and major compilers added support for it in the recent years. The most relevant mechanisms at this point are Microsoft's *Control Flow Guard (CFG)* and *LLVM Control Flow Integrity* in the *clang* compiler. The basic idea is to add runtime checks to the program which verify that the target of a function call actually is a valid function. In its most basic form, a CFI schema would allow all function entry points in the entire program as targets for indirect calls.

2.2.4.1 Microsoft Control Flow Guard

Microsoft has published research about CFI as early as 2005 [ABEL09] but only released a first version in 2014 with Windows 8.1 Update 3. *CFG* uses a bitmap to keep track of valid function call targets. Each indirect *call* instruction of a binary protected by *CFG* is prepended by a lightweight bitmap lookup (*_guard_check_icall*) that checks whether the target is valid; if not the program terminates. This partly stops an attacker from using the Return Oriented Programming technique described above, since he cannot call arbitrary code gadgets. He can, however, still exchange a valid function with another valid function and can thus manipulate the control flow to some degree. He is, nevertheless, limited in his possibilities as he can only execute entire functions instead of ROP gadgets.

It should be noted that *CFG* does currently not protect return addresses on the stack. These can still be used to jump to arbitrary addresses if all other mitigations are bypasses. Microsoft is expected to extend CFG to protect return addresses once hardware support is available such as Intel CET described later in section 2.2.4.3.

2.2.4.2 LLVM Control Flow Integrity

Control Flow Integrity is supported in the *LLVM clang compiler* since version 3.7. Apart from checking if a target address is a valid function as *CFG* does, *LLVM* performs additional integrity checks. For example, it further restricts the amount of possibly valid function calls by not only checking whether the target is valid, but also whether the number and the type of the arguments match. Furthermore, options are available to protect virtual calls and type casts in C++. Again, the CFI scheme does not protect from control flow manipulation through the corruption of return addresses.

2.2.4.3 Hardware Protections

Apart from software solutions like *CFG* and *LLVM CFI*, CPU manufacturers work on hardware-based mechanisms to protect a program's control flow. Qualcomm Technologies, Inc. released a paper in 2017 in which they describe pointer authentication on *ARMv8.3* [Secu17], that can be used for Control Flow Integrity. Intel Corporation has released a technology preview announcing measures to provide control flow enforcement, including the protection of return addresses by so-called *shadow stacks* [Corp16]. This shows that the existing software solutions might be supported by hardware features in the near future.

2.3 Summary

This chapter explained how memory corruption vulnerabilities form a large vulnerability class and are often exploited by attackers. Afterwards, basic information about the process virtual memory layout is given, illustrating the various memory segments text (code), data, BSS, Heap, and stack. Then, the stack segment is explained in more detail resulting in an exemplary exploitation technique of stack buffer overflows where the return address is overwritten. With the help of this example the most important mitigations of the last years were introduced and explained. These are the non-executable stack (*NX*), *Stack Canaries*, *Address Space Layout Randomization (ASLR)*, and finally *Control Flow Integrity*. In the end an overview of current research about hardware protection mechanisms was given.

3. Analysis

This chapter first shows the increased difficulty of exploiting memory corruption vulnerabilities today versus 20 years ago in section 3.1. It then proceeds by providing examples of non-control data structures whose corruption can lead to fatal consequences, thus emphasizing the impact of non-control data manipulation in section 3.2 and showing the importance of protecting against it. Finally, an overview of previous research on mechanisms against non-control data attack is given in section 3.3 followed by a description of the contribution of this thesis.

3.1 Status Quo

The mitigations presented in the previous chapter led to a change in the exploitation process. Whereas arbitrary code execution could be achieved easily 20 years ago by a simple buffer overflow, modern exploits now require bypasses for various exploit mitigations, thus increasing their complexity. In order to facilitate this process a common approach is to extend the abilities gained by the initial memory safety violation, be it a buffer overflow, integer overflow, use-after-free or any other vulnerability. Therefore, the attacker commonly tries to obtain a primitive that can arbitrarily read and/or write memory. Sometimes multiple vulnerabilities have to be combined to achieve this. Given the ability to read and write arbitrary memory, the attacker has a good foundation to defeat the remaining mitigations. A schematic sequence of the steps used to exploit a program is shown in figure 3.1, outlining the different difficulty level of today on the left-hand side versus the difficulty level 20 years ago on the right-hand side. In summary, this shows that exploitation of control data has become more difficult but is still possible with additional effort.

3.2 Impact of Non-control Data Manipulation

So far mitigations mainly focused on preventing the manipulation of a program's control flow by corrupting code pointers. However, little attention has been given to the impact of non-control data manipulation. This section will give examples of different types of data structures that can be dangerous if targeted by an attacker.

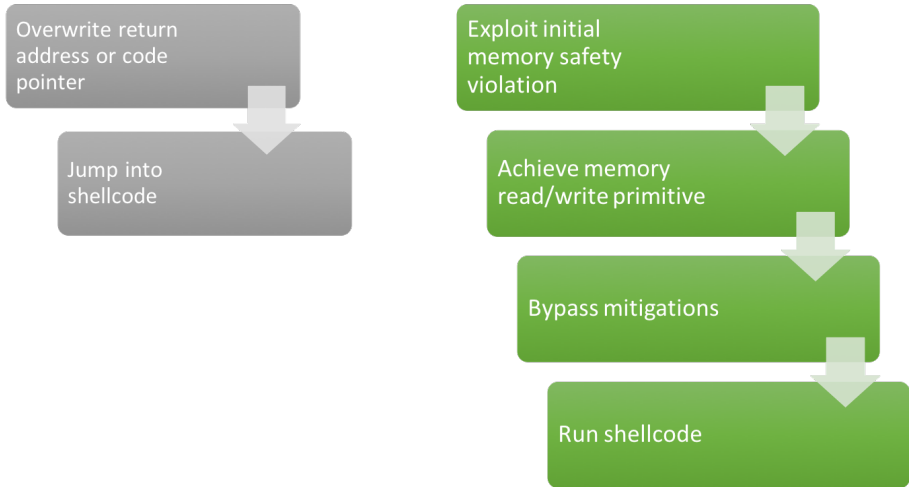

Figure 3.1: Typical exploitation process 20 years ago versus today

User Identity Data

Every program handling different users and privilege levels must keep the corresponding information somewhere in memory, making them vulnerable to corruption. An attackers can for example overwrite his own user credentials, which bind him to a certain privilege levels, with that of a more privileged user, and can consequently elevate his privileges. This potentially enables him to perform operations that might leak, corrupt, or destroy further critical data to which he should not have access or perform actions that are only allowed to be executed by an administrator.

User Input

Generally, programs have to validate user input before processing it in order to prevent malicious input from interfering with the program's intended behavior. This can be observed in web applications where user input is used in SQL queries and must be sanitized in advance to avoid SQL injection attacks. The sanitized user input is thus stored in memory where an attacker with the ability to write to arbitrary memory locations can undo the sanitizing and rewrite the malicious input to perform the aforementioned attack.

Decision-making Data

Programs are sometimes restricted in their behavior depending on external or internal factors. For example, a program might only execute the desired operations during a certain time of the day, or only a certain number of times. Corrupting this sort of restricting data in memory can force the program to behave in a non-intended way.

File Descriptors

File descriptors are opaque handles to different types of resources. They can amongst others be regular files, network sockets, or pipes. Given an example program that writes events into a log file, an attacker can overwrite the log file's file descriptor with that of a critical system file, thus corrupting the file and potentially gaining more privileges or causing the system to malfunction in other ways.

File and Directory Paths

Programs that work on files, as for example the log files mentioned above, likely need to store the directory path to said files somewhere in memory. Furthermore, programs are often configured to load configuration files or scripts from certain directories. Manipulating these file system paths in memory can lead to interesting attack vectors. Chen et al. [CXSG+05] showed at the example of an http server executing scripts from a certain directory, that overwriting the path to said directory can lead to arbitrary code execution. Overwriting any file path furthermore allows the creation and modification of any file the program is permitted to access, thus for example opening the possibility to create startup scripts executed at boot time if the file content is also controlled by the attacker.

Command Buffer and Command Arguments

A straight-forward non-control data attack involves data buffers that are at some point used as executable code by the application. Examples include shell commands, SQL commands used in database queries. By inserting commands of his choosing, the attacker can execute arbitrary code, only restricted by the size of the command buffer.

Remote Procedure Call Routine Numbers

Remote procedure calls allow subroutines to be executed in a different process or even on a different machine in the network. By faking their identifiers an attacker can possibly execute unintended code on the respective endpoint.

Kernel Structures

The operating system kernel hosts a variety of security critical data structures for managing processes and memory. Examples include process credentials responsible for indicating a process' privileges. Corrupting one of these essential structures can lead to an elevation of privileges.

This list only shows a small sample of data that can lead to information leaks, privilege escalation, or code execution if corrupted. Depending on the program and context endless scenarios can be thought of. A domain-specific example exists in the same-origin policy used by web browsers. This security policy ensures that a web page may only access resources such as user profiles of a second web page if the two share the same origin. An attacker with an arbitrary memory write primitive can corrupt the data structure containing the origin information by inserting his own server's address and can therefore trick the browser into accessing resources from a different origin. This attack is only applicable in web browsers and shows that different contexts allow for different corruption targets and result in different consequences.

A real-world example of an exploit for a memory corruption vulnerability[1] which makes use of corrupting a non-control data structure to bypass several mitigations

[1] https://nvd.nist.gov/vuln/detail/CVE-2018-6789

like NX and ASLR has been published in February 2018. The vulnerability was found in the EXIM mail transfer agent, which accounts for 57 % of all publicly reachable mail servers according to a survey[2] performed by E-Soft in May 2018. The *base64d* function in the SMTP listener was affected with a one byte heap-based buffer overflow. The security researchers who discovered the vulnerability show that by modifying a key data structure of the application through a partial write primitive remote code execution can be achieved and existing mitigations are bypassed[3].

To conclude, all of these examples show that non-control data manipulation can result in code execution, bypassing security mechanisms and mitigations in a simple way. Figure 3.2 shows a schematic comparison of typical control-data exploitation on the left-hand side and non-control data exploitation on the right-hand side. It can be seen that targeting non-control data provides an easy way to bypass existing control data protections like CFI and can facilitate the exploitation process given a suitable data structure. This can facilitate the exploitation process significantly.

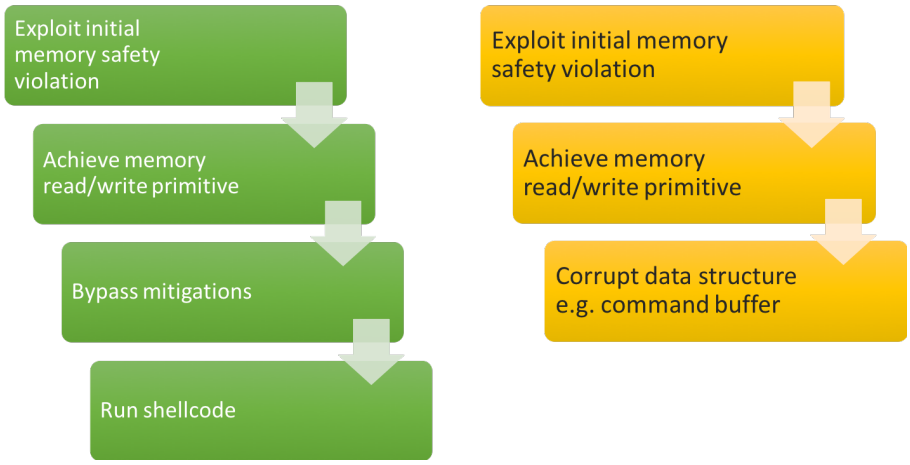

Figure 3.2: Comparison of control-data (left) and non-control data (right) exploitation

A study by Chen et al. [CXSG+05] concluded that non-control data attacks are realistic threats. Similarly Hu et al. [HSAC+16] illustrated the expressiveness of non-control data attacks. As a consequence non-control data should be protected against corruption in order to prevent existing mitigations to be bypassed easily.

[2] http://www.securityspace.com/s_survey/data/man.201804/mxsurvey.html
[3] https://devco.re/blog/2018/03/06/exim-off-by-one-RCE-exploiting-CVE-2018-6789-en

3.3 Related work

With non-control data manipulation becoming more relevant, researchers have developed multiple concepts to protect against their corruption.

Research like CCured [NeMW02] and Cyclone [GHJM05] attempt to add type-safety to the C programming language thus preventing memory errors from occurring and consequently preventing both control and non-control data attacks. This, however, results in non-negligible performance overhead due to the additionally required runtime checks. Therefore, in addition to the mitigations against control-data attacks, research towards mitigations against non-control data attacks has been carried out.

Amongst these approaches is SIDAN [DeTT09], which statically analyzes the source code to build a detection model for non-control data corruption. Based on the created detection model the source code is instrumented by inserting additional runtime checks to verify the consistency of the data. The induced security and performance overhead of this approach depend on the amount of reasonableness checks inserted. The higher the accuracy of the code instrumentation the bigger the incurred overhead will be. Similarly Castro et al. perform static code analysis to compute a data-flow graph for instrumenting the code to ensure at runtime that data-flow is in conformity with the computed flow graph [BeAA06].

Further mitigations against non-control data attacks can be classified by their intended target. Protections for both kernel and user space, which require different challenges to be solved, have been proposed in the last years. The following sections give a brief overview over the different studies which collectively highlight the increasing relevance of protections against non-control data attacks.

3.3.1 User Space Protection

For protecting user space applications from non-control data attacks researchers on the one hand introduce libraries that can be integrated to secure critical data and on the other hand apply compiler modifications to add language support for said task. Each of these approaches will be discussed below.

3.3.1.1 Libraries

Packaging the functionality to protect security critical data in a library, which can then be included in various projects, facilitates the process of adopting the intended security measures. While including and using the library requires code to be adapted, no modifications to the build infrastructure have to be made. However, the solutions are bound by language restrictions, that is to say that only operations possible in the used language are available for building the security measures. This section illustrates two example libraries that aim at protecting against non-control data attacks. The first library is based on additional integrity checks on read and write access while the second library relies on encrypting pointers to protected data structures.

Samurai

In 2008 Pattabiraman et al. describe the Critical Memory (CM) model that extends the classic load/store memory model [PaGZ08]. The critical memory area overlays the normal memory and is only accessible through special load and store

instructions. Normal load and store instructions are not allowed to access Critical Memory. Critical load and store instructions can detect anomalies by comparing the value in normal memory to its shadow version in critical memory. The detected inconsistencies can be handled with different error recovering strategies, starting with correcting the value in normal memory and continuing execution, and ending with raising an exception and terminating the program. The authors implemented and evaluated Critical Memory using the first variant of error correction.

Furthermore, an approach, which requires compiler modifications in order to support a new *critical* keyword to mark critical data, has been proposed. According to the authors their model also includes protection against hardware faults, uninitialized reads and invalid frees of memory addresses.

Samurai, however, cannot detect corruption if both the normal value and its shadow are corrupted. To minimize the probability of that they use a randomized allocation policy, thus, making it harder for an attacker to locate both versions of the data.

Nevertheless, an attacker can leak the meta data of a critical object and consequently retrieve the location of the shadowed memory area. A solution proposed to counter that problem is to encode the pointer value which, however, results in a constant performance overhead for decoding at both load and store.

Heap Data Pointer Encoding

Kim et al. [KiPy12] presented a mechanism that aims to protect heap memory from non-control data attacks with special focus on heap overflow attacks corrupting the linked list meta data for managing free chunks on the heap.

Firstly, the authors suggest an approach that protects the linked list pointers by encryption: list pointers are encrypted during initialization and only decrypted when dereferenced. That way an attacker cannot manipulate said pointers in a controlled way, even with write access through for instance a heap buffer overflow, since the pointer decryption will result in unpredictable values. However, it has to be noted that an attacker, who is able to leak the encryption key can manipulate the pointers a controlled fashion.

In addition, the authors extend the existing linked list structure to use duplicate list for increased robustness against corruption. Still this research only focuses on protecting heap meta data and not the actual data itself which can lead to attacks targeting the data discussed in section 3.2.

3.3.1.2 Approaches Involving Compiler Modifications

Modifying compilers provides an easy way to integrate security measures into existing programming languages, often requiring only slight modifications of the original code. In contrast to library-only solutions fewer restrictions apply, for example enabling the compiler to analyze code during compile time and apply security checks based on these results. On the other hand, customized build infrastructure likely causes slower adoptions of solutions since it is time consuming and possibly complicated to apply it to complex projects. Two examples of studies modifying the compiler are illustrated in this section.

Yarra

In 2014 Schlesinger et al. presented a server and runtime system (Yarra) to protect applications from non-control data attacks [SPSW+14]. Their work focuses on programming theory but also provides a practical prototype implementation of the concept. Yarra operates in two different modes: the first mode aims at providing memory safety by instrumenting the source code with dynamic checks where they are unable to be prove memory safety at compile time, similar to CCured and Cyclone. This generally results in high performance overhead. The second mode is inspired by Samurai (3.3.1.1) and relies on page permissions for data protection. In particular they disable write permission of memory pages while executing untrusted functions.

DataShield

In 2017 Carr et al. introduce a low overhead protection mechanism to provide Data Confidentiality and Integrity (DCI) by extending the C and C++ programming language with annotations that can be added by the programmer to protect security critical data like passwords, cryptographic keys, and identification tokens from illegal read and write access [CaPa17]. By having the programmer select these critical data structures and annotate them, instead of protecting all data, the performance overhead is reduced.

A prototype of the compiler and runtime system was evaluated by selecting and annotating critical data in existing libraries, pointing out the difficulty to choose which data is critical to be protected. The results show that especially guaranteeing confidentiality of critical data leads to non-negligible performance overhead while integrity-only protection only generates a slighter decrease in performance.

3.3.2 Kernel Protection

Corrupting kernel data can lead to serious security breaches such as privilege escalations. The following studies each focus on one particular kernel data structure that must be protected against manipulation.

PrivWatcher

In 2017 Chen et al. introduce PrivWatcher [CAGN17], which focuses on monitoring and securing process credentials in the Linux kernel. Process credentials are an important part of the kernel's access control system and determine amongst other things the privilege of a process. In fact, the authors state that most existing Linux kernel memory corruption exploits achieve privilege escalation by overwriting process credentials.

PrivWatcher firstly isolates process credentials and secondly validates runtime updates to them by verifying the new value and checking the authenticity of the verification context. Thirdly it enforces time of check time of use (TOCTOU) consistency which means that credentials must not be modified after validation and that the verification and usage of values must be in same context, so that verified credentials cannot be used for another process. To achieve this an execution domain isolated from the kernel and with control over the Memory Management

Unit (MMU) relocates the process credentials to a safe memory region that is not writable by the kernel. Furthermore, they implement kernel extensions to guarantee TOCTOU consistency and heuristics to detect potentially malicious modification of process credentials. They close their work by announcing future work on extending the system to cover further kernel data structures.

PT-Rand

The second study on kernel data-only exploitation by Davi et al. in 2017 [DGLS17] highlights the importance of protecting page tables from corruption. The authors claim to introduce the first practical solution not requiring hardware trust anchors, hypervisors, or expensive integrity checks. PT-Rand randomizes the location of page tables and prevents this information from leaking by obfuscating references to the page table, thus protecting a page's access permissions from manipulation. The authors evaluate a prototype implementation on a Linux kernel in combination with a CFI scheme to protect against control data attacks on kernel page tables and observe low overhead on multiple benchmarks.

3.4 Contribution

The goal of this work is to introduce a new low-overhead protection mechanism against non-control data attacks. The introduced mitigation is designed to protect user space data structures. Section 3.3.1 has already presented previous research on protecting both the integrity and confidentiality of user space data structures. Moreover, the advantages of implementing a library as opposed to modifying the compiler have been discussed there as well. As such, a library will be implemented to avoid the need for changes to existing build infrastructure and facilitate an easy adoption of the security measures. In particular, this mechanism will ensure the integrity of security critical data to mitigate easy bypasses of mitigations against control-data exploitation like CFI presented in section 3.2. A main focus of the mitigation lies on low overhead for read access, as critical data structures tend to be read frequently but only written rarely. This library, however, will only protect data from corruption and does not guarantee confidentiality of the data. This limitation makes it possible to design a mitigation with low overhead for secured read access, since no decryption of pointers has to be performed as in the system designed by Kim et al. presented in section 3.3.1.1.

3.5 Summary

This chapter first described how control-data attacks have become increasingly more difficult throughout the last years as more and more mitigations were introduced. It then proceeded with analyzing the impact of non-control data attacks on the basis of various examples of critical data structures such as file descriptors and command buffers. Next, this chapter discussed the relevance of this type of attack compared to classic exploitation techniques. Furthermore, related work about the protection against non-control data attacks has been presented. These can roughly be grouped into two categories: first, mechanisms that protect critical kernel data structures such as page tables and process credentials. Second, protection mechanisms that prevent attackers from manipulating user space data structures. These can further

be divided into libraries that can be included by the programmer like Samurai, and solutions that require modifications of the compiler like Yarra and DataShield. Finally, these solutions were compared to the goal of this thesis, stating that the new mitigation focuses on the integrity of critical data structures in user space, and aims for low overhead for read access to protected data.

4. Design and Implementation

Section 3.3 has shown different approaches to protect from non-control data manipulation, ranging from user space to kernel space protections. The mitigation introduced in this work will provide a protection mechanism for mitigating non-control data manipulation in user space. All in all the mitigation must meet the following requirements:

- Protect critical data from illegal modification
- Prevent an attacker from faking critical data structures and replacing the original
- Introduce only low overhead for read access of critical data.

It must, however, be emphasized that the mitigation does not protect critical data from illegal reads, as such it does not guarantee the confidentiality of the data.

Section 3.3 already illustrated the advantages of packaging the required functionality in a library. Therefore, this mitigation will be implemented as a shared library that can easily be included in projects. The basic idea is that the programmer specifies which data structures are critical and applies the protection mechanism provided by the library to protect that data. Due to this approach the overhead introduced by the mitigation only applies to protected data while non-critical data access can be performed without overhead. Nevertheless, the programmer must be aware which data structures are security critical and should thus be protected by this mitigation.

Section 3.3 already showed mechanisms that encrypt pointers to critical data structures thus preventing an attacker from dereferencing them without knowing the key. While this approach aims at guaranteeing both integrity and confidentiality it does not meet the requirements for this mitigation. Encrypting the pointers means that on each read or write access the pointer must be decrypted. This results in noticeable overhead. Moreover the encryption key likely has to be kept in memory, making it accessible to an attacker who can read the process' memory. Since a design goal of this mitigation is to induce small overhead on read access of critical data, this

approach will not be pursued further. Instead, the mitigation will rely on memory access permissions. In order to protect critical data from being corrupted, the write permission is removed from pages containing it. This, however, poses several problems to be addressed: on the one hand read-only memory pages do not only prevent an attacker from corrupting critical data, but also prevent the program itself to apply valid modifications to protected data structures. Therefore, write access must be granted by remapping the page writable whenever the program needs to update the relevant data. For this purpose a *unlock* and *lock* functionality will be provided that grant write permissions to the protected data if needed and revoke it afterwards. On the other hand a mechanism for dynamic memory allocation must be provided since security critical data is often created at runtime. Finally a mechanism to validate pointers to protected objects is required. Figure 4.1 sums up the different components needed to implement the mitigation. The read-only mapping serves as a basic property. On top, the *unlock* and *lock*, the dynamic memory allocation, and the read validation components will be built.

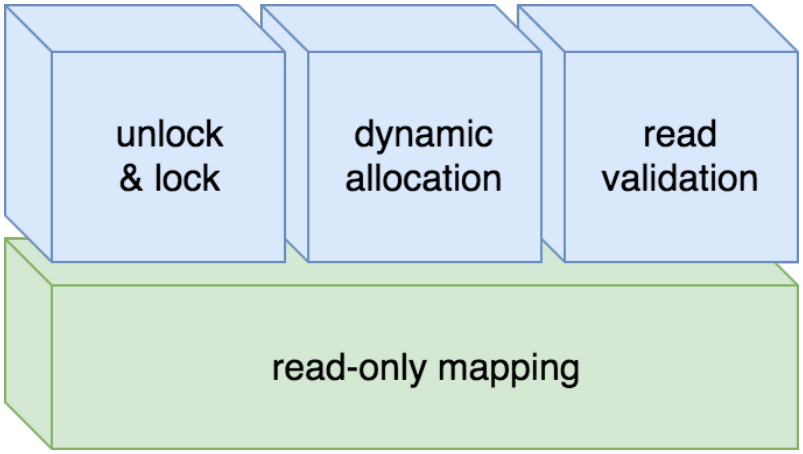

Figure 4.1: Components of the mitigation

The following sections illustrate in detail the main components named above and describe their implementation. It should be noted that the implementations of each component can potentially be improved performance- or security-wise, depending on the underlying system architecture. The implementation presented in this work represents a generic and architecture-independent solution, however, the individual implementations can be exchanged easily without the need to change the external interface.

4.1 Unlock and Lock

A core requirement is the ability to allow legitimate modifications of protected data. Page permissions can be changed by using the *mprotect* syscall depicted in listing 4.1. This way the write permission can be added when the data must be written and revoked afterwards.

```
1 int mprotect(void *addr, size_t len, int prot);
```

Listing 4.1: The mprotect system call

It is, however, advisable to prevent a potential attacker from knowing the location of writable memory pages containing critical data to stop him from writing there while the write permissions are turned on. A common way to achieve this is to randomize the location memory. This approach has already been seen in section 2.2 when discussing the ASLR mitigation. Nevertheless, in this specific case, randomization is not necessary for read access, where it would induce unnecessary overhead, but is only relevant for writable mappings. A possible solution, taking advantage of this observation, is to keep multiple read-only mappings of the same data and chose a random mapping to add write permissions when writing to it. This way read access on every mapping is valid and no further checks are necessary, but only one mapping is writable. Another solution, which is implemented in this work, is to use shared memory. Shared memory is normally used for inter-process communication allowing different processes to map the same memory into their address space. Modifications are thus synchronized between the processes. In this case, the shared memory is not needed to communicate with another process, but to enable the heap to manage different views onto the memory, namely a read-only view and a writable view. Figure 4.2 illustrates the different mappings of the shared memory in a process's address space. On the one hand a read-only mapping exists permanently that will be used for any read access to protected data. On the other hand a writable view will be mapped into memory at a random location when the protected data is unlocked and directly unmapped afterwards. Write access will only be possible on this mapping.

4.1.1 Randomization

An important requirement for mapping memory at random locations is to obtain randomness that cannot be predicted by an attacker. The base address of the *mmap* region is randomized due to ASLR. However, subsequent mappings are commonly placed contiguously in that region, making it possible for an attacker to predict the location of the next mapping if he knows the location of previous ones. To avoid this, other sources for randomness have to be used to map each individual mapping at a truly random position. The virtual file "/dev/urandom" on UNIX for example serves as a secure random number generator. The disadvantage of this solution, however, is that the random value is stored somewhere in the process address space where an attacker can potentially leak it and consequently determine the location of the writable memory. Another way to do obtain randomness in a secure way is to use the *RDRAND* CPU instruction on recent Intel and AMD CPUs. This solution is, however, not available on all architectures.

For simplicity this implementation will use the randomness of *mmap*, This can, nonetheless, easily be improved with better architecture specific sources of randomness if necessary.

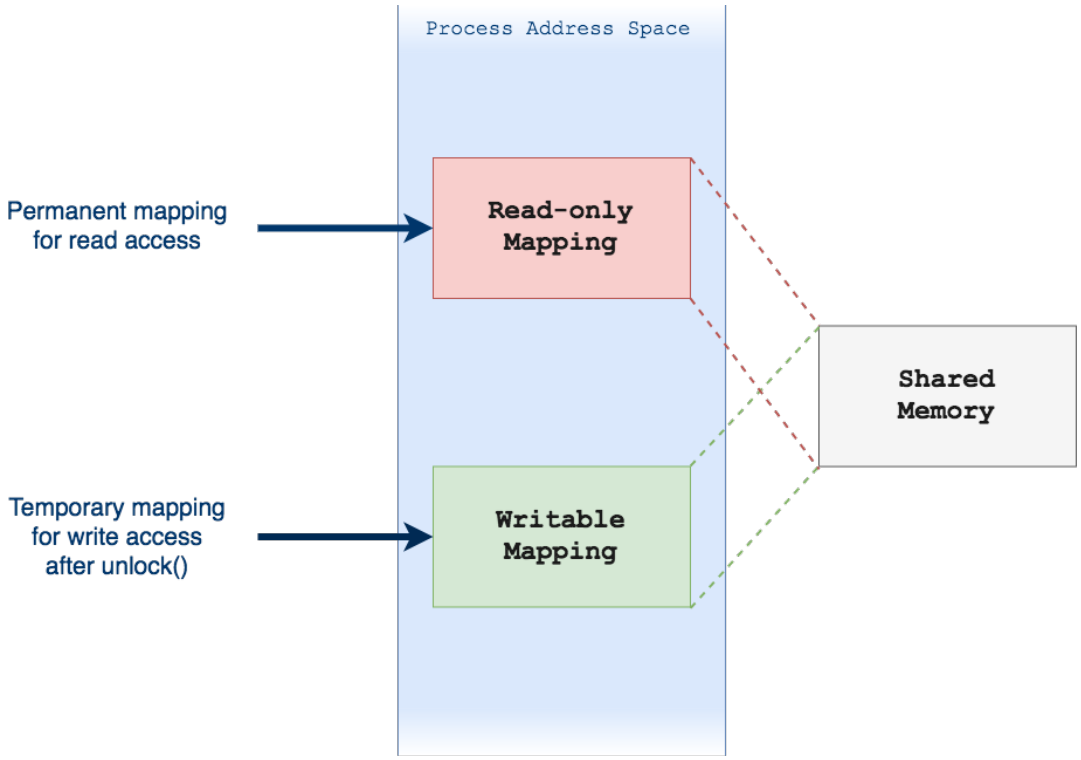

Figure 4.2: Shared memory mapped read-only and writable into the address space

4.2 Secure Heap Allocator

In order to handle dynamic memory allocation requests at runtime most operating systems provide a heap allocator. Since security critical data is potentially created at runtime, a separate secure heap with its own allocator will be designed that uses shared memory as its backing memory and extends its interface to support the *unlock* and *lock* functionality.

A typical heap allocator requests memory from the operating system and manages it internally for handling allocation requests from the application. If the heap allocator runs out of space, the heap is expanded by requesting more memory from the operating system. The application interface to the allocator mainly includes a function to reserve memory (*malloc*) and a function to release the reserved memory (*free*). Further functionality like allocating preinitialized memory blocks or resizing an allocated block can be provided. One of the biggest problems a heap allocator must address is fragmentation to minimize the waste of free memory situated between used memory blocks by reusing previously allocated and freed memory areas. For that purpose additional data structures are required to track which memory areas are in use and which are free to be reallocated.

Starting in 1987 a general-purpose allocator has been implemented by Doug Lea[1] managing memory blocks as so called *chunks* that contain an additional header for meta data. The code for this allocator, called *dlmalloc*, is in the public domain and has been widely used and adapted amongst others for the GNU C library allocator. For this implementation *dlmalloc* will be adapted to provide a separate secure heap

[1] http://g.oswego.edu/dl/html/malloc.html

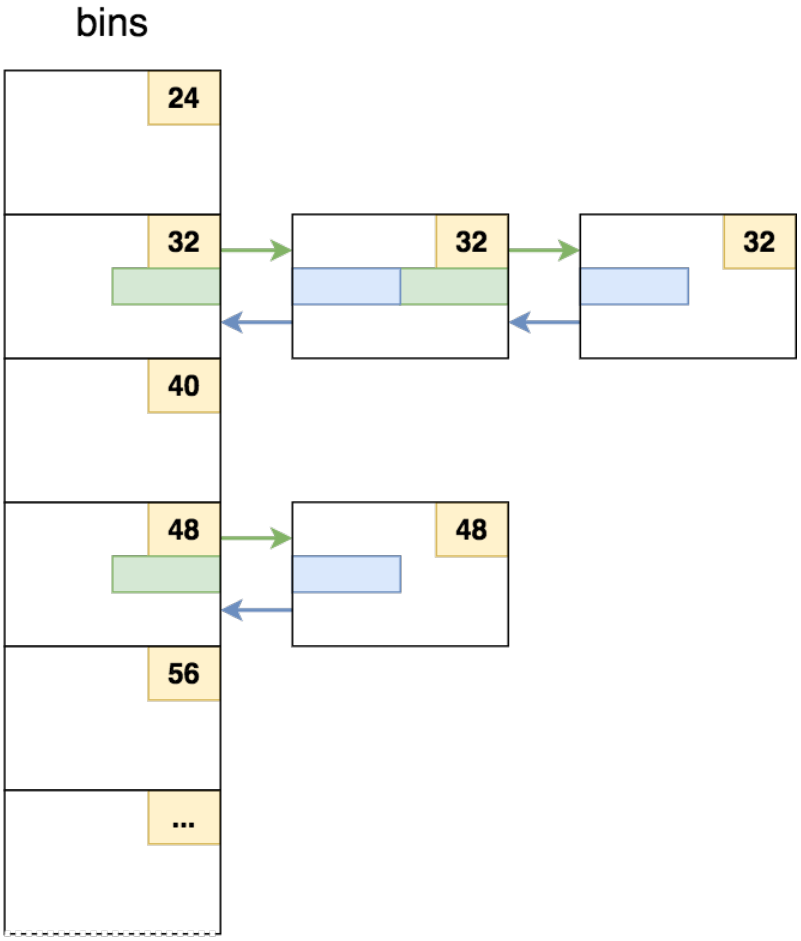

Figure 4.3: Free chunks are stored in double-linked lists which are grouped into bins of different sizes starting with a minimum size of 24 bytes.

that uses shared memory as its underlying memory and provides the *unlock* and *lock* functionality.

Dlmalloc uses double-linked list to manage free chunks. Each chunk's header field contains its size and flags with information about its status. In addition, the header of a free chunk stores the forward and backward pointer to the previous and next element in the list. Free chunks are sorted by size into so called *bins* and each *bin* presents the head of one double-linked list. Figure 4.3 shows an example illustrating the different bins which are each 8 bytes apart in size and the free chunks inserted into their respective list. For chunks greater than 256 Bytes *dlmalloc* uses an additional tree data structure. When *dlmalloc* cannot satisfy an allocation request by the application because no chunk large enough for the requested size is available, the allocator requests additional memory from the system. In order not to request more memory every time consecutive allocations are made, *dlmalloc* requests a large block of memory at once and stores the remaining block in the so called *top chunk*. On further allocations smaller chunks can be split from the *top chunk* without the overhead induced when requesting more memory from the system.

The whole process of allocating and freeing memory chunks is shown in figure 4.4. On *malloc*, the allocator first checks if a free chunk is available in the respective bins. If it is, the chunk is unlinked from the list and a pointer to its data field is returned

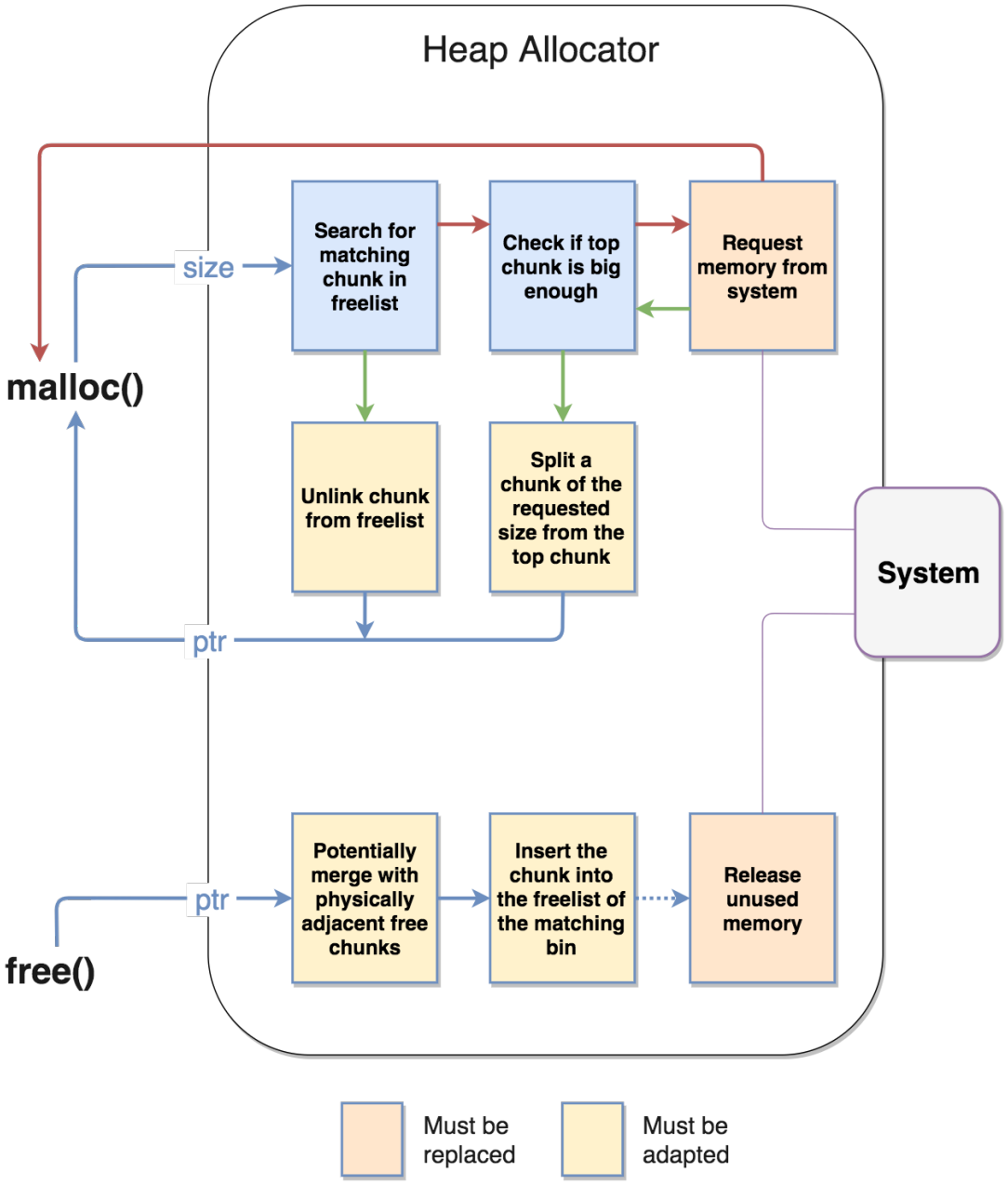

Figure 4.4: Overview of the operations done by the *dlmalloc* heap allocator when calling malloc and free.The functionality that must be replaced or adapted for the secure heap allocator is marked in orange and yellow.

to the application. If there is no matching chunk in the free lists, the *top chunk* is checked to be big enough. If that is the case, a chunk of the requested size is split from the *top chunk* and returned, else, the allocator has to request more memory from the system to increase the *top chunk's* size before continuing the operation. On execution of the *free* operation the physically adjacent chunks to the one to be freed are examined and merged with the current one if they are free as well. This prevents the allocator from accumulating many small free chunks that could fit a larger request if merged. The potentially merged chunks will consequently be inserted as one chunk into its respective bin.

Figure 4.4, furthermore, illustrates which parts of the original *dlmalloc* allocator must be modified. Evidently, the code that reserves memory from the system must be replaced to work on shared memory. Originally there exist two ways of reserving memory for the heap in UNIX. The preferred way is to use the *sbrk* system call shown in listing 4.2. *Sbrk* extends or shrinks the program's data segment contiguously by *incr*. Calling *sbrk(0)* will return a pointer to the current end of the segment. If *sbrk* is not available or fails the allocator will try to use *mmap* to extend the heap and *munmap* to shrink it. The listing of both is given in listing 4.3.

```
1   /* change data segment size */
2   void * sbrk(int incr);
```

Listing 4.2: The sbrk system call

```
1   /* allocate memory, or map files or devices into
       memory */
2   void * mmap(void *addr, size_t len, int prot,
3               int flags, int fd, off_t offset);
4
5   /* remove a mapping */
6   int munmap(void *addr, size_t len);
```

Listing 4.3: The mmap and munmap system calls

In order to replace this mechanism the *sbrk* call is emulated by a new function that works on shared memory as shown in listing 4.4. Using the file descriptor obtained when opening the shared memory, the *mmap* call in line 6 maps additional memory adjacent to the already mapped secure heap. This view onto the shared memory is only mapped readable and cannot be written by anyone. Likewise, the memory is unmapped when the function is called with a negative argument in line 14.

```
1  static void* morecore(int64_t s) {
2
3      if (s == 0) {
4          return SHEAP_BASE + SHEAP_SIZE;
5      } else if (s > 0) {
6          void* ptr = mmap(SHEAP_BASE + SHEAP_SIZE, s,
               PROT_READ, MAP_SHARED | MAP_FIXED, SHEAP_FD,
               SHEAP_SIZE);
7          if (ptr == MAP_FAILED) {
8              abort();
```

```
9           }
10          SHEAP_SIZE += s;
11          return ptr;
12      } else {
13          int ret = mummap(SHEAP_BASE + SHEAP_SIZE + s, -s);
14          if (ret) {
15              abort();
16          }
17          SHEAP_SIZE += s;
18          return SHEAP_BASE + SHEAP_SIZE;
19      }
20  }
```

Listing 4.4: The morecore function of the secure heap allocator emulating sbrk for requesting more memory from the system

Mapping only readable memory, however, means that heap meta data cannot be written. Therefore, each time a chunk is inserted or removed from the double-linked list or the *top chunk* is modified, the area containing the affected chunks must be unlocked. For that purpose a *chunk_unlock* and a *chunk_lock* function have been implemented. The *chunk_unlock* function depicted in listing 4.5 takes a pointer to a read-only chunk in the secure heap and maps it at a random position representing a view to the shared memory at the corresponding offset. It is, however, only possible to map regions with page granularity. That is to say that at each *unlock* of a chunk, additional adjacent data is possibly mapped writable as well, even though it is not modified.

```
1   static mchunkptr mchunk_unlock(mchunkptr p) {
2
3       size_t size = sizeof(mchunk);
4
5       char* page = (char*) get_page(p);
6       size_t offset = (char*)p - page;
7       size_t alloc_size = page_align(offset + size);
8
9       void* base_w = mmap(0, alloc_size, PROT_READ |
            PROT_WRITE, MAP_SHARED | MAP_POPULATE, SHEAP_FD, page
            - SHEAP_BASE);
10      if (base_w == MAP_FAILED) {
11          abort();
12      }
13      return (mchunkptr)(base_w + offset);
14  }
```

Listing 4.5: Function to create a writable mapping of the page containg the chunk to be modified

Figure 4.5 depicts schematically the process memory layout after unlocking one chunk. In addition to the normal heap, the secure heap containing protected data is mapped as a read-only view onto the shared memory. Furthermore, the page containing the unlocked chunk is mapped writable, representing only the relevant part of the writable shared memory.

When the heap meta data has been written, the page containing the unlocked chunk is unmapped from memory and is no longer accessible. Listing 4.6 shows the *chunk_lock* function, which determines the page containing the unlocked chunk and unmaps it.

```
1  static int mchunk_lock(mchunkptr p_w) {
2
3      size_t size = sizeof(mchunk);
4
5      char* page = (char*) get_page(p_w);
6      size_t offset = (char*)p_w - page;
7      size_t unmap_size = page_align(offset + size);
8
9      int ret = munmap(page, unmap_size);
10     if (ret) {
11         abort();
12     }
13     return ret;
14 }
```

Listing 4.6: Function to unmap the writable mapping created by mchunk_unlock

Listing 4.7 shows an example application of the introduced *chunk_unlock* and *chunk_lock* functions in the secure dlmalloc. In this case the size and status flags of a newly allocated chunk are set. The function takes a pointer to the chunk to be modified in the secure heap. Before writing, however, the chunk is unlocked returning a pointer to a writable mapping on which the write operation is performed. Afterwards the chunk is unlocked again to unmap the writable mapping and and prevent further modifications.

```
1  /* Set size, cinuse and pinuse bit of this chunk */
2  #define set_size_and_pinuse_of_inuse_chunk(M, p, s) {\
3      mchunkptr p_w = mchunk_unlock(p);\
4      p_w->head = ((s)|PINUSE_BIT|CINUSE_BIT);\
5      mchunk_lock(p_w);\
6    }
```

Listing 4.7: Example application of the chunk unlock functionality

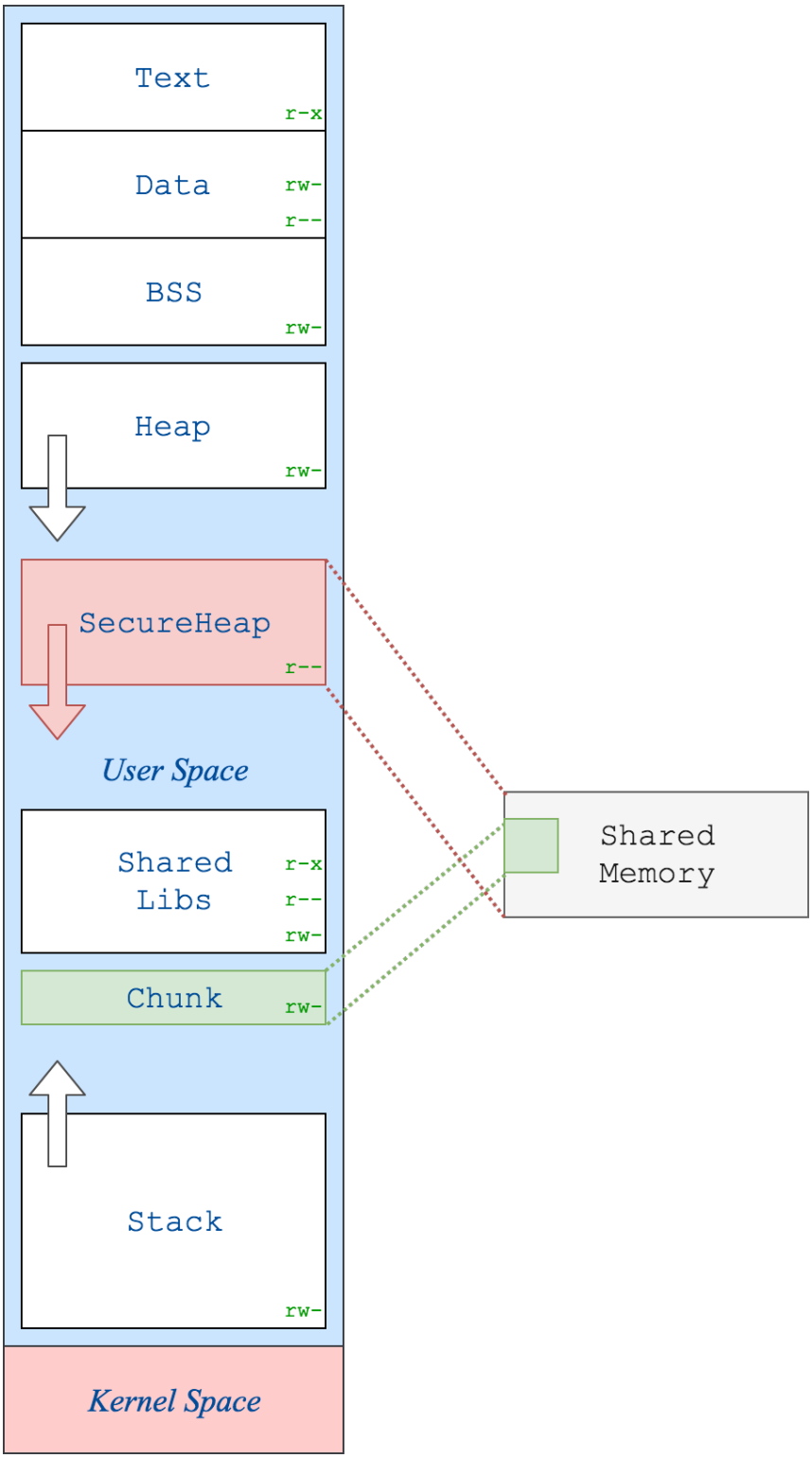

Figure 4.5: Illustration of the process address space while one chunk is unlocked showing the secure heap separate from the normal heap and the writable mapping of the chunk

4.3 Read Validation

Another approach an attacker might pursue to counter the measures presented in the previous sections is to fake existing data structures in a region not protected by the

mitigation. For that, he writes data into writable memory resembling the protected data structure but whose content is fully controlled by the attacker. Then the attacker modifies the pointers that initially pointed to the protected data structure to point to the faked object controlled by the attacker. The program, hereupon, continues execution using the fake data structure.

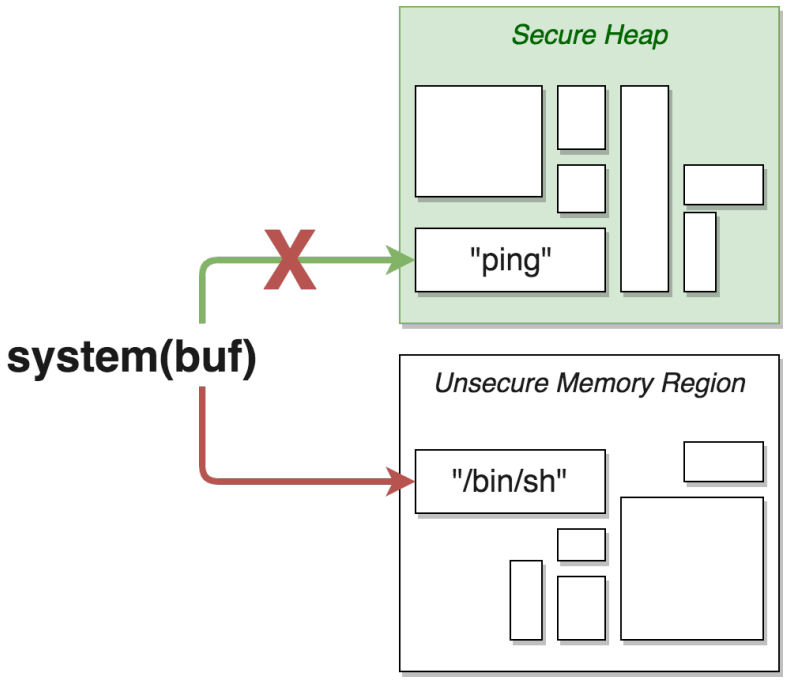

Figure 4.6: Faking of a protected object outside the secure heap

Figure 4.6 shows an exemplary attack where a command buffer is targeted. The buffer is securely allocated on the secure heap where it contains its intended and harmless command string. An attacker with an arbitrary write primitive is, however, able to corrupt the pointer to instead point to a writable memory region where the attacker has previously written his own command to.

In order to counter this attack, read access to the secure heap must be validated. In its simplest form, the validation ensures that the pointer indeed points anywhere into the secure heap. To make this validation efficient, the secure heap is always located somewhere inside a predefined address range which is defined by *SHEAP_RANGE_START* and *SHEAP_RANGE_END*. Listing 4.8 shows the validation function which then consists only of a cheap range check. This operation is implemented as a macro and therefore in-lined at every call inducing very little overhead.

```
#define sh_validate(mem) {\
    if ((uintptr_t)(mem) < SHEAP_RANGE_START ||\
        (uintptr_t)(mem) >= SHEAP_RANGE_END) {\
        abort();\
    }\
}
```

Listing 4.8: Validation function

4.4 Interface

To summarize, this section illustrates the interface provided of the secure heap library by showing an example application protecting a given data structure *ABC*.

First the C interface is summarized in listing 4.9, showing the four essential functions exported by the library.

```
1        const void* shalloc(size_t);
2
3        void  shfree(const void*);
4
5        void* sh_unlock(const void *mem);
6
7        void  sh_lock(void *mem);
```

Listing 4.9: C interface

Listing 4.10 then shows the application of the library to protect the *ABC* data structure. First, an instance of *ABC* is allocated on the secure heap with *shalloc*, taking the object's size as an argument. For initializing the structure's members, it is unlocked with *sh_unlock*. Afterwards all values can be written before the data must be locked again by calling *sh_lock*. Line 22 demonstrates the necessary validation of a pointer before accessing it in line 23. Finally, if no longer needed, the object is freed, using *sh_free*.

```
1  #include "sheap.h"
2
3  struct ABC {
4      int a;
5      float b;
6      int c;
7  };
8
9  int main(int argc, char* argv[]) {
10
11     const struct ABC* abc = shalloc(sizeof(struct ABC));
12
13     // unlock before one more consecutive writes
14     struct ABC* abc_w = sh_unlock(abc);
15     abc_w->a = 1;
16     abc_w->b = 2;
17     abc_w->c = 3;
18     // lock to remove write permissions
19     sh_lock(abc_w);
20
21     // validate before access
22     sh_validate(abc);
23     int x = abc->a;
24
25     shfree(abc);
26     return 0;
27 }
```

Listing 4.10: Example C application

A detailed outline of which operations are done on which mapping of the shared memory can be seen in figure 4.7. The readable mapping of the critical data structure is mapped into memory starting with its allocation, and is only unmapped when it is freed and dlmalloc decides to return memory pages to the operating system. The writable mapping, however, is only temporarily mapped between the call to the *sh_unlock* and *sh_lock* functions. Accessing it afterwards results in an access validation. Before reading a value from the secure heap, the *sh_validate()* function is called on the read-only mapping.

On top of the C interface, the C++ interface additionally provides two smart pointer classes. First the *secure_pointer* class representing a pointer to the secure heap and, second, the *unlocked_pointer* class representing an unlocked version of the object. These classes facilitate the necessary operations significantly. On the one hand, each read access to a secure pointer automatically validates the pointer before dereferencing it. This is done by overloading the -> operator and can be seen in listing 4.12, in lines 16 and 21. In line 21 the -> operator on the *secure_pointer* additionally performs the range check to ensure that it does indeed point into the secure heap before dereferencing the actual pointer.

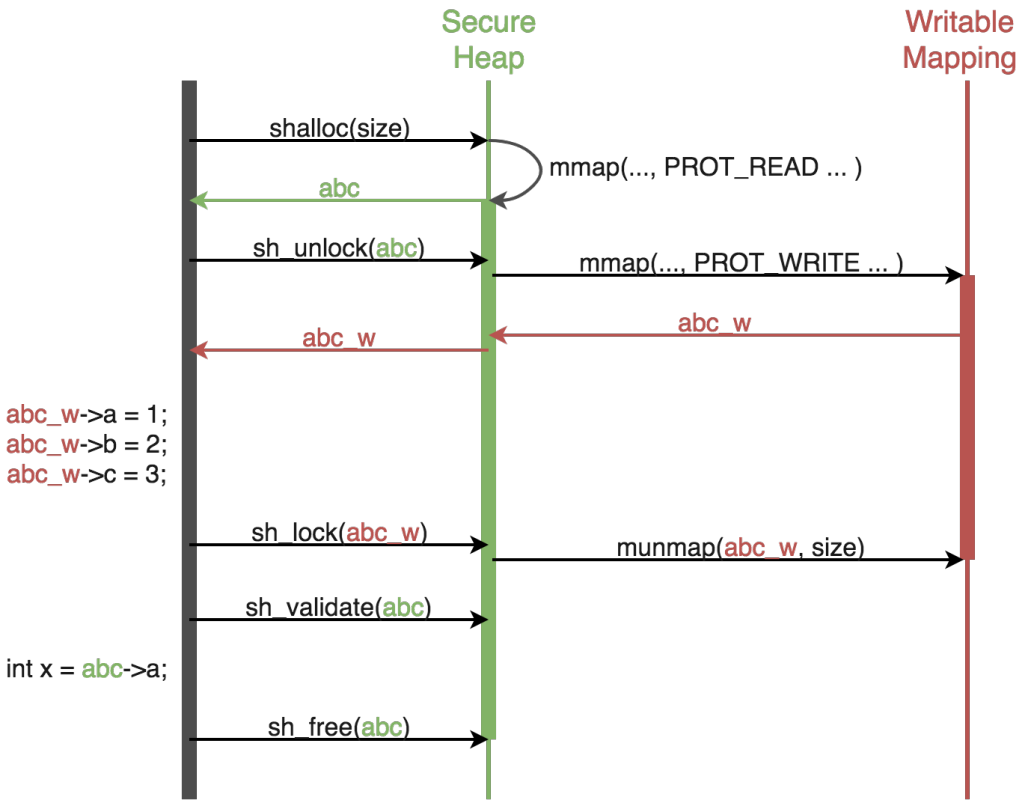

Figure 4.7: Sequence diagram the interactions with the protected data structure in listing 4.10.

```
1 template <typename T, typename... Arg>
2 secure_pointer<T> make_secure(Arg&&... args) {
3         const void* r = shalloc(sizeof(T));
4         void* w = sh_unlock(r);
5         new (w) T(std::forward<Arg>(args)...);
6         sh_lock(w);
7         return secure_pointer<T>(reinterpret_cast<const T*>(
              r));
8 };
```

Listing 4.11: Implementation of the *make_secure* function to construct protected objects in C++

On the other hand the object's members can be automatically initialized with the *make_secure* function, which securely allocates, unlocks, and constructs the object in the unlocked memory before locking it again. The implementation of *make_secure* is shown in listing 4.11 and an example of its usage can be seen in listing 4.12 in line 13.

Most importantly, the *unlocked_pointer* is automatically locked when the destructor is called, that is in particular when leaving its scope. Therefore, the programmer must not remember to call the *sh_unlock* function. This can also be seen in listing

4.12, where the *unlocked_pointer* is destroyed when leaving the block scope in line 17. At that point the writable memory is unmapped as well.

```
1  #include "sheap.hpp"
2
3  struct ABC {
4      int a;
5      float b;
6      int c;
7
8      ABC(int a1, float a2, int a3) : a(a1), b(a2), c(a3) {}
9  };
10
11 int main() {
12
13     secure_pointer<ABC> abc = make_secure<ABC>(11, 12.1, 13)
           ;
14     {
15         unlocked_pointer<ABC> abcW = abc.unlock();
16         abcW->a = 3;
17     } // abcW is automatically locked on leaving scope
18
19
20     // -> validates pointer before access
21     abc->a;
22 }
```

Listing 4.12: Example application in C++

4.5 Summary

This chapter presented the design and implementation of the mitigation designed to protect against non-control data attacks. The mitigation is implemented in form of a shared library and relies on page-level memory permissions by mapping critical data read-only into memory. In order to protect critical data, the programmer uses the interface provided by the implemented secure heap allocator which leads to the data being allocated on a separate protected heap. Furthermore, the allocator adds a *unlock()* and *lock()* functionality to its interface. These functions that *map* and *unmap* protected data writable is required to allow the program to make legitimate modifications to protected-data. For this purpose shared memory is used. Finally, the data on the secure heap must be validated at each read access. This is performed with an additional range check before each read. The library provides both a C and a C++ interface.

5. Evaluation

This chapter evaluates the mitigation introduced in the previous chapters. The evaluation is composed of two parts - A security evaluation and a performance evaluation. The security evaluation first covers the security in single-threaded applications lists necessary measures to take in order to guarantee that the mitigation can reliably protect critical data in section 5.1.1 and 5.1.2. It then goes on with depicting an unsafe usage pattern that has to be avoided in section 5.1.3 and outlines a type confusion problem that is not entirely solved by the proposed mechanism in section 5.1.4. Finally the security in multi-threaded applications is evaluated in section 5.1.5. Afterwards, the performance of the implementation is evaluated in section 5.2.

5.1 Security Evaluation

The proposed mitigation is based on CPU memory protections which guarantees by construction that read-only memory cannot be corrupted. Nevertheless, the scheme only works if it can be guaranteed that the memory permissions cannot be extended by an attacker. For this purpose Control Flow Integrity comes into consideration by preventing the attacker from executing arbitrary code that adds write permission to protected memory pages. This means that as long as CFI is not bypassed, the memory read-only memory protection represents a valid security boundary. It must further be noted that an attacker that is able to tamper with memory protections is able to execute more severe attacks since he can for instance simply overwrite the code section which originally is also mapped non-writable. Therefore, this attack vector will not be taken into further consideration.

However, the following situations in which problems arise have been identified and must be kept in mind by the programmer to ensure a secure application of the mitigation. This list also serves as a guide for the developer which code patterns to avoid to guarantee that an attacker has the smallest attack surface possible.

5.1.1 Blacklisting

Current CFI implementations prevent an attacker from changing the control flow to arbitrary address. However, whole functions can still be called, possibly enabling an

attacker to call the *unlock* function to gain write access to protected data. In order to prevent abuse of legitimate functions CFI schemes use blacklists for managing functions that must never be called indirectly. Consequently, the *sh_unlock* function must be added to this list. Another solution is to implement the *unlock* functionality as a macro so that it will be always inlined. This means that no callable *unlock* function exists. However, evaluating the feasibility of implementing the function as a macro is part of future work.

5.1.2 Omitted Unlock

Similarly, after calling *sh_unlock*, the omission of the corresponding *sh_lock* functions leads to severe problems. If a developer using the secure heap interface forgets to call the *lock* function, the writable mapping exists until the process terminates. Therefore it is essential for the protection of critical data to *lock* it as soon as the relevant modifications have been made.

5.1.3 Indirect Calls Before Lock

A problem that can easily occur in C++, if the programmer does not pay close attention to the scope of unlocked data, is shown in listing 5.1. The *perform_critical_operation()* function unlocks a protected data pointer and writes a value to it. However, the object is only unlocked when its current scope is left, which happens only in line 13. Therefore, the data is unprotected while a virtual function call is executed in line 12. In contrast to non-virtual function calls, where the destination is known at compile time, virtual function calls result in indirect calls since the function pointer has to be loaded from the object's *vtable* first. Depending on the CFI scheme an attacker might be able to call a function of his choosing at the virtual call site by corrupting or replacing the *vtable*. If he is now able to call a function that writes controlled data to an address of his choosing, he can corrupt the unlocked protected data. This attack is partially mitigated by creating the writable mapping at a random location.

```
1  class Foo {
2          virtual void some_unrelated_operation();
3
4          void perform_critical_operation(secure_pointer<
               ProtectedData> data)
5          {
6                  auto dataW = data.unlock();
7
8                  dataW->some_field = 42;
9
10                 // Virtual (indirect) function call.
11                 // Note, data is still unlocked here
12                 this->some_unrelated_operation();
13         }
14 }
```

Listing 5.1: Code making a virtual function call during critical data is unlocked

A better way of writing this code, which does not suffer from the described problem, is shown in listing 5.2. The time in which the writable data is mapped is minimized by creating a new scope especially for this purpose which is explicitly closed before performing the unrelated virtual function call. When the virtual function call is executed now, the unlocked data is already unmapped.

```
1  void perform_critical_operation(secure_pointer<ProtectedData
       > data)
2  {
3          // encapsulate unlocked data in its own small scope
4          {
5                  auto dataW = data.unlock();
6                  dataW->some_field = 42;
7          }
8
9          this->some_unrelated_operation();
10 }
```

Listing 5.2: The correct way to perform the critical operation

The same problem can occur with the C interface, although its occurrence is more obvious since the *unlock* operation is performed explicitly and should normally be called right after the write operation. An example for this can be seen in listing 5.3.

```
1  void perform_critical_operation(const struct ProtectedData*
       data) {
2
3          struct ProtectedData* data_w = sh_unlock(data);
4          data_w->some_field = 42;
5
6          // the follwing two lines should be executed in
               exchanged order
7          some_function_pointer();
8          sh_lock(data_w);
9  }
```

Listing 5.3: Occurence of the indirect call problem in C

In general, developers are advised to keep the time window in which data is unlocked as small as possible to avoid interference from unrelated operations.

5.1.4 Internal Type Confusions

Section 4.3 already introduces a simple, yet effective, method to prevent an attacker from faking objects outside the secure heap and causing the application to treat the fake object as the critical data structure. By prepending range checks to each read access to the critical object, the mitigation ensures that it actually resides in the secure heap.

However, this does not prevent an attacker from confusing objects of different types that both reside in the secure heap, since in that case the range check will still succeed.

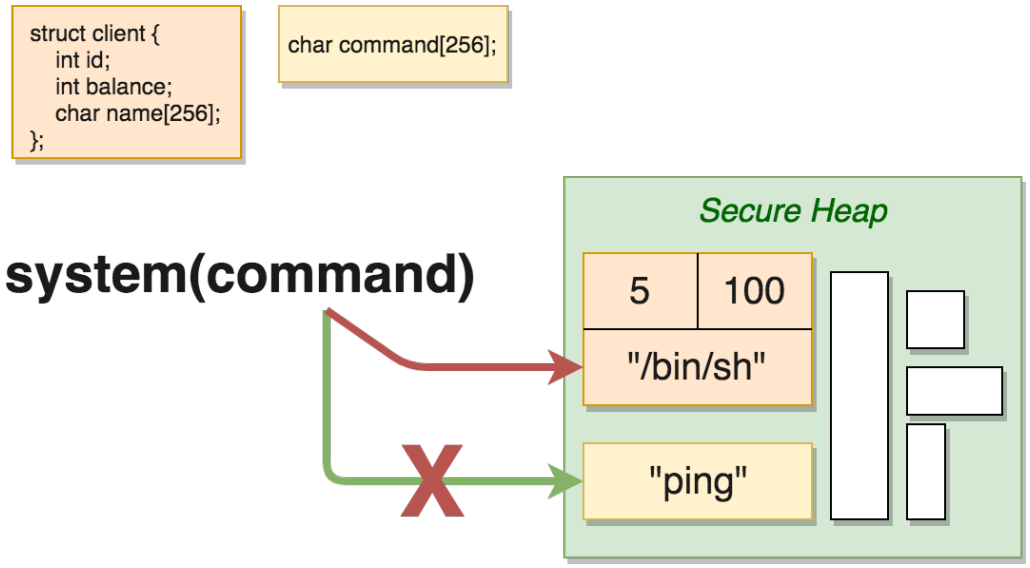

Figure 5.1: Confusion of two different object types on the secure heap

Figure 5.1 illustrates an example of the confusion of two different object types. The first type is a command buffer that is later executed and is stored on the secure heap as a simple character buffer. The second type is a structure representing a client of the application containing different structure members including a character buffer for the client's name. The name field is potentially controlled by the attacker if he can create his own profile and could thus be set to the string "/bin/sh". The attacker is also able to corrupt the pointer to the argument of the *system()* call with his arbitrary write primitive and replaces the argument pointer with a pointer to the character buffer of the client structure. As a result he achieves code execution.

The solution to this is to use isolated heaps, that is to say that each object type must be assigned its own heap. When accessing the protected data structure, the validation check must be extended to check if the queried heap actually contains the right object type. Furthermore, the access must be aligned by object size to avoid object-internal type confusions.

Further problems occur with dynamically sized objects and C++ inheritance, since the assignment into the different heaps may be ambiguous. This refined protection mechanism against internal type confusions is subject for future work.

5.1.5 Multi-threading

Additional problems occur in multi-threaded applications, particularly when an attacker controls two ore more threads of the program.

5.1.5.1 Time Of Check Time Of Use

The first problem that occurs is a *time of check time of use (TOCTOU)* problem, that arises if data from untrusted memory regions is copied into the secure heap. A simple example for illustrating this problem is input validation: when user input is handled by an application, it is often sanitized before further processing. This ensures that the input does not contain malicious data that would pose a security risk for the application. If these validation checks are executed while the data resides in untrusted memory, for instance the stack, an attacker can interfere with another thread before the data is stored in the secure heap. After the validation is done, but before the data is moved onto the secure heap, the other thread overwrites the validated data with malicious content which is then stored on the secure heap and treated as trusted data by the application. A simple solution to counter this problem is to execute the validation only when the data is stored on the secure heap and can not be corrupted anymore.

5.1.5.2 Race Conditions

Multi-threaded applications are generally susceptible to race conditions. The problem describe next is caused by the short time frame in which the protected data is unlocked. While the thread unlocking the data only executes the instructions intended by the programmer, another thread controlled by the attacker with an arbitrary read and write primitive could take advantage of the writable mapping and corrupt the data. The secure heap allocator complicates this type of attack by always choosing a random location for the writable mapping. However, this does not guarantee that the location cannot be found out by an attacker. For instance, the address of the writable mapping might be spilled onto the stack since the CPU registers are need for other values. The attacker can thus retrieve the needed address from the stack. Apart from being a non-trivial attack, this procedure is not risk-free for the attacker. If he loses the race, the write access results in an access violation, thus causing the whole application to crash.

To counter race condition attacks the developer should keep the time window in which protected data is unlocked as small as possible. To prevent the attacker from guessing or determining the location of the writable data, the mapping must be mapped at a truly random location of which the address must never spill onto the stack. Functioning solutions for this would need compiler modifications to ensure that the location is always kept in register and never stored in untrusted memory. A different solution, which would, however, require per-thread memory protections, something that none of the major operating systems support at this time, could be possible as well. In that case the problem would cease to exist as the writable mapping would only exist in one thread. The feasibility of such a mechanism is subject to future work.

Non-crashing writes

Above, the possibility of attacker determining the location of the writable mapping and writing to it has been illustrated. It has also been discussed that this poses a certain risk for the attacker since the program might crash. Listing 5.4 shows a way for an attacker to avoid this risk.

```
1  void* data = ...;              // attacker controlled data
2  size_t data_len = ...;
3  void* addr = ...;              // address to which to write
4                                 // without risk of crashing
5
6  // Goal: write |data| to |addr| without risk of crashing
7
8  int fds[2];
9  pipe(fds);
10 write(fds[1], data, data_len);
11 read(fds[0], addr, data_len);
```

Listing 5.4: Write invalid memory without crashing

Instead of directly writing to the desired memory address, the attacker can use certain system calls that do not crash if the access is invalid. Examples include the *read* and *write* system calls writing data from or into a file. With the *write* syscall the attacker can put controlled data into the pipe and write it with the *read* system call to the desired address. The important observation is that both system calls do not crash when the memory access fails but simply return an error. The disadvantage of this is that it takes longer since two system calls have to be executed instead of one memory access. The attacker can also use this method to search the address space for writable mappings without crashing. However, he would have to linearly search multiple terabytes of memory which would likely exceed his time constraints. To generally protect against non-crashing system calls, *seccomp filters* can be used, which is essentially a programmable system call filter that, based on the system call number and arguments, can decide to prevent the system calls from executing. In particular, it would be possible to disallow a set of predefined system calls such as *read* and *write* when they are used with an address that is located in the secure heap region. This would prevent the described attacks but might also impact the application to some degree.

5.2 Performance Evaluation

This section evaluates whether the performance of the mitigation meets the requirements listed in chapter 4. The key objective was to induce little overhead for read access for data. The design chapter describes that overhead is generally only induced for data that is marked by the programmer. Therefore, both read and write access to non-critical data structures do not increase the execution time. The allocation

and freeing of critical data, as well as write access to it are expected to induce noticeable overhead as memory pages have to be mapped and unmapped. Read access to the secure heap, however, is expected to induce little overhead since the memory access is only augmented by a range check which should only add comparatively few additional CPU cycles.

Table 5.1 shows the measured durations of data creation and access on the secure heap compared to the operations on the regular heap. The measurements were performed on Ubuntu 16.04 with glibc version 2.23 running on an Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz with 64 GB of RAM.

The benchmark compares the *malloc* and *free* functionality of the glibc heap allocator with the corresponding implementation provided by the secure heap allocator. The values depict the average duration of one million iterations. These allocations were divided into smaller groups that were executed successively to avoid out-of-memory errors. Allocating memory on the secure heap is expected to create the biggest overhead compared to the other operations because on every *sh_alloc()* two *mmap* and two *munmap* system calls are required to manage the internal heap data structures explained in 4.2. Each *mmap* system call takes approximately 650 ns and each *munmap* system call takes 5050 ns. As a result the *sh_alloc* operation is measured to take 2500 ns compared to 28 ns which is the longest duration of all operations. However, it should be noted that the overhead occurs only once per allocated critical object and is thus not as relevant. Securely freeing an allocated object on average takes 1 *mmap* und one *munmap* system call each. Therefore the secure free operation takes only approximately half the time the secure allocation needs, that is to say 1300 ns compared to 8 ns.

	Normal	**Secure**
alloc	28 ns	2500 ns
free	8 ns	1300 ns
read	40 ns	40 ns
write	40 ns	1100 ns

Table 5.1: Performance of normal and secure memory operations

Furthermore, read and write access to normal and protected data structures has been measured. On write access to the secure heap one *unlock()* and on *lock()* operation must be performed, which are essentially one *mmap* and one *munmap* system call plus the actual write access. As expected the execution time of 1100 ns compared to 40 ns is similar to the one of the *free* operation since they require the same amount of system calls. An important point that should be noted is that the overhead for write access can be reduced if multiple successive writes to the same data structure are performed. In that case the *unlock()* and *lock()* operations only have to be performed once for all write instructions, thus decreasing the overhead significantly.

Finally the performance of read access has been measured. Reading non-protected memory requires 40 ns just like normal write access. For accessing the secure heap the validation range check has to be performed additionally. The measurements show that the overhead for the validation is non-negligible as the secure read access takes 40 ns as well. These access times were measured with uncached memory, which explains why these operations take longer than *alloc* and *free*.

The overall overhead of the mitigation strongly depends on the context and functionality of the program. Critical data that is only rarely written, like for example configuration parameters parsed from a file at startup, result in low overhead whereas data structures that have to updated regularly induce larger overhead.

5.3 Summary

In this chapter the introduced mitigation was evaluated for security and performance. The security evaluation first showed that the *sh_unlock* function has to be blacklisted by the CFI scheme in order to prevent an attacker from calling it to create a writable mapping. Furthermore, it stressed that every *unlock* operation must always be followed by a *lock* operation as soon as possible to prevent security implications. Furthermore, an insecure usage pattern regarding indirect function calls has been laid out illustrating that unlocked data must be locked before performing such a call. Next, the security evaluation showed that the validation functionality does not protect from internal type confusions. This has been declared subject to future work. Finally, race conditions in multi-threaded applications have been detected that cannot be prevented at this point. The exploitation of those is however non-trivial. The performance evaluation confirmed that the mitigation meets the requirement to provide low overhead for read access of critical data. The measured overhead is negligible. The allocation and freeing of critical data as well as write access is performed with noticeable overhead. This is, however, negligible if the data structure is rarely read. All in all, this result matches the expectations to the mitigation.

6. Summary and Future Work

In this work a secure heap allocator to protect against non-control data attacks has been presented. The primary goal was to protect critical data structures against corruption by an attacker with the ability to read and write to arbitrary memory addresses. This was achieved by relying on page-level memory protections and implementing a secure heap manager that marks its data as read-only to prevent modifications by an attacker.

The main challenge with this approach was to enable legitimate write access to protected data by the program itself. In order to allow write access the memory has to be remapped with write permissions. This was accomplished by using shared memory and mapping both a read-only and a writable view onto it. While the read-only mapping exists permanently throughout the lifetime of the object, the writable mapping is only available during legitimate write operations on the protected data structure. For this reason the basic heap interface, that is *alloc* and *free*, was expanded with a *unlock* function for mapping the object writable and a *lock* function for unmapping it again.

Since critical data structures tend to be read frequently but only written rarely, a major focus of the mitigation was to induce little overhead for read access. The performance evaluation indeed showed that this overhead is negligible.

The implementation successfully prevents an attacker from modifying critical data in single-threaded applications through the use of page table permissions but assumes a mechanism to enforce Control flow integrity. In multi-threaded applications a fundamental problem exists in the form of race conditions. This mitigations is designed to make these race conditions hard to exploit, however, fully solving this problem likely requires hardware or operating system support and is subject to future work.

Another problem portray type confusions, where an attacker causes an application to treat partially controlled memory as a critically data structure without ever corrupting the secure heap. A basic countermeasure has been implemented by providing a *validate* functions that ensures that a critical data structure indeed resides in the secure heap. However, fully resolving this problems requires type isolated heaps and is again for future investigation. Finally some unsafe usage patterns of the secure heap allocator have been identified and recommendations on how to avoid them have been given.

All in all the introduced mitigation meets the requirements laid down in chapter 3 by providing a low-overhead protection mechanism against non-control data attacks.

Bibliography

[ABEL09] M. Abadi, M. Budiu, Ú. Erlingsson und J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security* 13(1), 2009, S. 1–40.

[BeAA06] B. Bershad, M. ACM Digital Library. und T. ACM Special Interest Group in Operating Systems. Securing software by enforcing data-flow integrity Miguel. In *OSDI*, 2006, S. 407.

[Bial18] J. Bialek. The Evolution of CFI Attacks and Defenses. 2018.

[CAGN17] Q. Chen, A. M. Azab, G. Ganesh und P. Ning. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[CaPa17] S. A. Carr und M. Payer. DataShield: Configurable Data Confidentiality and Integrity. *ASIACCS '17 Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security*, 2017.

[CBPW+15] N. Carlini, A. Barresi, M. Payer, D. Wagner und T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. *USENIX Security Symposium*, 2015, S. 161–176.

[Corp16] I. Corporation. Control-flow Enforcement Technology Preview rev. 1.0. In *Intel Spec*, Nr. June, 2016, S. 1–136.

[CXSG+05] S. Chen, J. Xu, E. C. Sezer, P. Gauriar und R. K. Iyer. Non-control-data attacks are realistic threats. In *USENIX Security*, 2005, S. 12–12.

[DeTT09] J. C. Demay, E. Totel und F. Tronel. SIDAN: A tool dedicated to software instrumentation for detecting attacks on non-control-data. In *Post-Proceedings of the 4th International Conference on Risks and Security of Internet and Systems, CRiSIS 2009*, 2009, S. 51–58.

[DGLS17] L. Davi, D. Gens, C. Liebchen und A.-R. Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Proceedings 2017 Network and Distributed System Security Symposium*, 2017.

[GHJM05] D. Grossman, M. Hicks, T. Jim und G. Morrisett. Cyclone: A type-safe dialect of C. *C/C++ Users Journal* 23(1), 2005, S. 112–139.

[HCAS+15] H. Hu, Z. L. Chua, S. Adrian, P. Saxena und Z. Liang. Automatic Generation of Data-Oriented Exploits. *USENIX SEC*, 2015, S. 177–192.

[HSAC+16] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena und Z. Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, 2016, S. 969–986.

[KiPy12] K. Kim und C. Pyo. Securing heap memory by data pointer encoding. *Future Generation Computer Systems* 28(8), 2012, S. 1252–1257.

[KSPC+14] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. S. OSDI und U. 2014. Code-Pointer Integrity. *usenix.org*, 2014.

[NeMW02] G. C. Necula, S. McPeak und W. Weimer. CCured. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '02*, Band 37, New York, New York, USA, 2002. ACM Press, S. 128–139.

[PaGZ08] K. Pattabiraman, V. Grover und B. G. Zorn. Samurai: protecting critical data in unsafe languages. *ACM SIGOPS Operating Systems Review* 42(4), 2008, S. 219–232.

[Samp18] N. Sampanis. WINDOWS 10 RS2/RS3 GDI DATA-ONLY EXPLOITATION TALES. 2018.

[Secu17] Q. P. Security. Pointer Authentication on ARMv8.3 Design and Analysis of the New Software Security Instructions. 2017.

[Shac07] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). *Proceedings of the 14th ACM conference on Computer and communications security* 22(4), 2007, S. 552–561.

[SPSW+14] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker und B. Zorn. Modular protections against non-control data attacks. *Journal of Computer Security* 22(5), 2014, S. 699–742.

[SuXZ17a] B. Sun, C. Xu und S. Zhu. The Power of Data-Oriented Attacks: Bypassing Memory Mitigation Using Data-Only Exploitation Technique Part I. 2017.

[SuXZ17b] B. Sun, C. Xu und S. Zhu. The Power of Data-Oriented Attacks: Bypassing Memory Mitigation Using Data-Only Exploitation Technique Part I. 2017.

[Team15] P. Team. RAP: RIP ROP. *Editor & Publisher*, 2015.

[TEBJ+11] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh und P. Ning. On the expressiveness of return-into-libc attacks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Band 6961 LNCS. Springer, Berlin, Heidelberg, 2011, S. 121–141.

[TRCC+14] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano und G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*, 2014, S. 941–955.

[VDSCB12] V. Van Der Veen, N. Dutt-Sharma, L. Cavallaro und H. Bos. Memory errors: The past, the present, and the future. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Band 7462 LNCS, 2012, S. 86–106.