Comenius University in Bratislava

Faculty of Mathematics, Physics and Informatics

Updating process code at runtime using GDB

Bachelor Thesis

2023

Filip Koseček

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Updating process code at runtime using GDB

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	Ing. Dušan Bernát, PhD.

Bratislava, 2023
Filip Koseček

38134651

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Filip Koseček
Študijný program:	informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	bakalárska
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Updating process code at runtime using GDB
Aktualizácia kódu procesu pomocou GDB

Anotácia: Odstraňovanie chýb (bezpečnostných, výkonostných, funkčných), alebo zlepšovanie a rozširovanie funkcie programov, známe ako aplikovanie záplat (patch), spravidla vyžaduje zastavenie vykonávania pôvodného procesu a opätovné spustenie upravenej verzie. To prináša nedostupnosť poskytovanej služby počas aktualizácie.
Analyzujte možnosti úpravy kódu procesu v prostredí OS Linux počas jeho vykonávania (bez nutnosti jeho ukončenia) ako aj známe riešenia. Navrhnite a implementujte mechanizmus aktualizácie kódu procesu ako rozšírenie funkcií nástroja GDB. Funkčnosť riešenia overte a vyhodnoťte jeho vlastnosti.

Vedúci:	Ing. Dušan Bernát, PhD.
Katedra:	FMFI.KI - Katedra informatiky
Vedúci katedry:	prof. RNDr. Martin Škoviera, PhD.
Dátum zadania:	27.10.2022
Dátum schválenia:	31.10.2022

doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.... študent

.... vedúci práce

38134651

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Filip Koseček
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Updating process code at runtime using GDB

Annotation: Treating various issues (either security, performance or function related), improvements and extensions of programs, commonly known as applying code patches, requires stopping the executing process and restarting the modified version. This incurs unavailability of a service
during the process of upgrade deployment.
In OS Linux environment, analyze the possibilities and suitable mechanisms involved in modification of a code of a running process
without necessity of its termination. Design and implement a tool allowing for updates of the process code at runtime
as an extension of GDB tool. Verify the functionality of the solution and evaluate its properties.

Supervisor:	Ing. Dušan Bernát, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	27.10.2022

Approved: 31.10.2022 doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

....
Student

....
Supervisor

Acknowledgments: I would like to thank my supervisor Ing. Dušan Bernát PhD. for his ideas and willingness to help during his supervision of the thesis, for everything he taught me during the operating systems course and the interesting topic of the thesis which he came up with.

Abstrakt

Táto práca sa zaoberá modifikáciou kódu bežiaceho procesu bez nutnosti reštartovania, nazývaným aj ako aplikovanie záplat. Analyzujeme a opíšeme možné prístupy aplikovania záplat s využitím nástroja GDB, pričom sa zameriame na operačný systém Linux a procesorovú architektúru x86-64. Vychádzame z existujúcich navrhnutých postupov, pričom navrhneme možné vylepšenia a rozšírime funkcionalitu. Popíšeme navrhnuté a implementované rozšírenie pre nástroj GDB umožňujúce vykonávať záplaty kódu programu počas jeho behu. Rozšírenie zároveň obsahuje mechanizmus umožňujúci sledovať vykonané zmeny počas života procesu ako aj vrátenie procesu do jeho pôvodného stavu. Vyhodnotíme funkčnosť tohto rozšírenia a navrhneme jeho možné vylepšenia.

Kľúčové slová: aplikovanie záplat, GDB, Linux, metadáta

Abstract

This thesis deals with modification of a process code without the need to restart the process which is also known as applying live patches. We analyze and describe ways to perform live patching using GDB debugging tool targeting Linux operating system and x86-64 architecture. We base our work on existing approaches and propose potential improvements and additional features. We implement an extension for GDB allowing to perform process code modification at runtime including function replacement, tracking patching activity throughout the process lifetime and restoring the process to its original state. We evaluate the functionality of the extension and propose potential future improvements.

Keywords: process live patching, GDB, Linux, patch metadata

Contents

Introduction

Once a process has started, it is usually unwanted to modify its code since it may lead to a crash or an undefined behavior. However, in some cases the code modification at runtime might be useful. Imagine a program containing a bug that has been running for a long period of time. Fortunately, the bug has not caused any damage yet but it might in case it is left unsolved. In order to deal with this issue, one could recompile the debugged source code and restart the program. This way, the whole process would start over and the entire progress would be lost. This is when live patching comes into play allowing to inject the fixed function code into the process, while retaining the process's state.

In terms of live patching, we usually have to make some assumptions, the most obvious ones being the target architecture and the operating system. For example, not all architectures allow to modify the code which is being executed. We focus on x86-64 platform which uses von Neumann architecture, meaning that code and data occupy the same address space. Therefore, the code can be accessed in the same way as the program data.

We base our research on existing solutions, most notably *Modification of process code at runtime* [12]. Our thesis deals with design and implementation of a tool on the top of existing GDB tool for x86-64 Linux, capable of live patching process code.

Chapter 1

Linux processes and dynamic linking

1.1 Linux processes

Every time a programmer writes a piece of code and wants to run it, a compiler or an assembler is used to generate a binary file containing executable machine code. Once compiled, the executable can be mapped into memory address space in order to run. A process is a running instance of a program. In UNIX systems, a new process is created using a pair of system calls. Firstly, `fork` creates a copy of the calling process and `exec` replaces the image of the newly created process with the executable which is passed as a parameter. Linux kernel stores data necessary to manage process resources. This data is stored in a structure located inside the kernel space. The structure's name is PCB which stands for process control block. It contains contents of registers, page tables, a process file descriptor table, and so on.

Process virtual address space is split into so called virtual memory areas. These areas then map directly to physical memory. Since x86-64 implements memory paging, a continuous region in virtual memory address space does not necessarily map to a continuous region in physical memory. Since x86-64 platform belongs to von Neumann architecture family, code and data share the same address space. Moreover, every memory page comes with read, write and execute permissions. Operating systems usually allow to set these permissions via system calls. On Linux, this can be achieved by executing `mprotect` system call. Linux offers yet another system call named `ptrace` which provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers [7]. Not to mention, `ptrace` can bypass the lack of write or read page permissions provided the calling process has appropriate privileges. This means that we do not need to bother with setting page access permissions when using `ptrace` to access memory of another process.

1.2 Dynamic linking

The output of a compiler is an object file. This binary file contains machine code, program data and other metadata. Object file format used in UNIX systems is `ELF`. Programs usually consist of multiple object files called modules, with each containing its own code and data. The modules need to be eventually linked together into one executable file. A program called linker, used for this purpose, is usually packed with the operating system. GNU linker can be invoked using `ld` command on Linux. Most compilers such as `gcc` directly invoke it after creating object files, unless explicitly specified not to do so by the user. Modules that are often reused in different programs are called libraries. An example is standard `libc` library available on Linux. There are two main approaches when it comes to linking libraries to the main program.

The first one is called static linking. All relocations of symbol addresses are resolved at compile-time without further need of any modifications. The output file represents the full program image. This, however, results in a larger size of the final executable since all library data and code are packed into the executable file. This approach consumes more disk space since many copies of the same code are stored in the file system if the same library is used by different programs. Fortunately, there is a more sophisticated solution used more frequently in modern operating systems.

Dynamic linking is a technique which links libraries to the executable at runtime. This procedure is carried out by the dynamic linker. The dynamic linker is the first program to be loaded into memory address space working as an interpreter for the main program. It is loaded by a program called loader. Its tasks include determining and loading dependencies, relocate the application and all dependencies and initialize the application and dependencies in the correct order [4]. This approach allows to perform *lazy binding*. Rather than linking libraries at compile-time, the compiler generates a special section named `.plt`. This section contains code used to invoke the dynamic linker and is executed once a function from a shared library has been called for the first time. Table `.got.plt`, stored in a special section, contains an entry for each dynamically linked function referenced in the program. The dynamic linker loads the corresponding object file into process's memory address space and fills in the `.got.plt` table entry with the real address of the newly loaded function. This ensures that the newly loaded function is executed the next time it is called in the program. This approach, however, puts restrictions on the dynamic libraries. They must be compiled as position-independent so that the dynamic linker can place them on any address without the need to perform any relocations except for global variables. Rather than using instructions with absolute address mode, a compiler generates code using instructions using relative address mode. This means that a memory address of operands is computed relatively from the current value stored in the instruction pointer register (RIP) while executing the instruction. Platform x86-64 encourages use of this mechanism because both `jmp` and `call` instructions use relative address mode.

`ELF` format contains several sections which are mapped into memory address space once the module gets loaded. These are `.text`, `.bss`, `.got` to name a few. Moreover, `gcc` allows to define custom sections with the variable attribute [8]:

```
__attribute__((section("section_name")))
```

This attribute orders `gcc` to place the variable in the specified section.

Library `glibc` provides `dlopen` function which loads the object file whose path is specified as an argument. Obviously, the object must be a shared library. A handle for the library is returned or `NULL` in case of an error. The dynamic linker tracks a reference count for each opened library. This value is incremented every time `dlopen` is called for the same library path. On the other hand, once `dlclose` is called with the handle returned by `dlopen`, the reference count is decremented. As soon as the reference count reaches 0, the library gets unmapped from the process address space. Function `dlopen` takes an additional integer parameter containing flag values. Operation `OR` can be used to combine several flags and pass it to `dlopen`. Flag `RTLD_NOLOAD` might be useful since it can be used to test whether a library is loaded. If it is loaded, its handle is returned, otherwise the return value is `NULL`. This could be used to obtain the library handle of an already opened shared object. The problem is, even if the library is already opened, after calling `dlopen` with `RTLD_NOLOAD` flag, the library reference count is incremented nevertheless. Therefore, this approach is not optimal for testing if libraries are opened.

The set of functions for dynamic libraries manipulation contains yet another useful function called `dlsym`. It can be used to retrieve the address of a symbol from the dynamic library once it gets loaded. The arguments are the handle for the library returned from `dlopen` and a pointer to a string representing the symbol name. The return value is either the address of the symbol or `NULL` when the symbol cannot be found or an error occurs.

The last function to mention from `dl` family is `dlerror`. It takes no arguments and is used for diagnostic purposes. This function returns a string specifying the error type of the last operation performed by the above-mentioned functions.

Last but not least, `gcc` is able to recognize a constructor function which is called right after the object file gets loaded. This can be achieved by prefacing the constructor function name with attribute `__attribute__((constructor))`. It should be noted that this feature is `gcc` specific.

Chapter 2

Related work

Application of live patches was explored in thesis *Modification of process code at runtime* [12] resulting in an implementation of a tool named *bakatsugi* that can replace both user-defined and library functions with newly defined ones. The project was mainly written in Rust programming language with x86-64 assembly and C modules. This work focused on GNU/Linux operating system running on x86-64 platform. Let us take a look at a brief overview of the implementation of this tool. To stop and seize control of the running process, bakatsugi uses `ptrace` system call. It uses shared memory and a set of protocols to communicate with the target process via UNIX sockets. Function `dlopen` mentioned in section *Dynamic linking* is used to load libraries containing newly defined functions. The tool searches the target process's memory mappings to find this function. When it comes to the function replacement, the whole function's code cannot simply be overwritten by the replacement one. The original function has a certain length and the replacement function may not fit in the space allocated for the original function. Restrictions on a function's length are undesirable which means that this solution is unacceptable. Therefore, a more promising solution is used in bakatsugi - trampolines. The idea behind a trampoline is that we only need to overwrite the first few bytes of the beginning of the original function to pass on the control flow to the replacement function. The x86-64 jump instruction is used for this purpose.

Of course, patching functions in dynamically linked libraries is a very different task. The compiler generates a procedure which allows shared libraries to be linked to the executing binary on the fly described in section *Dynamic linking.* All that needs to be done is to write the address of the replacement function into the corresponding entry of the `.got.plt` table. Additionally, this solution comes with no time penalty as opposed to patching own functions as no additional instructions are needed to redirect the control flow. The mechanisms proposed in *Modification of a process code at runtime* [12] will be described in detail later on since our research is based on the thesis.

Another tool we would like to mention is a tool by SUSE called Libpulp [11]. This framework enables process live-patching by using trampolines. Unlike the previously mentioned toolkit, this one requires the code of the process to have a special format. A special library must be preloaded in the target process and the source code must be compiled with `gcc -fpatchable-function-entry` flag for functions to become patchable. This option basically orders `gcc` to generate multiple `nop` instructions at the beginning of a function to create space for a trampoline. This solves the problem with functions that are shorter than the trampoline. Before applying a patch, Libpulp performs several checks on whether the target function is patchable. Libpulp is also able to revert patches, in other words, to restore functions to their original states. Furthermore, Libpulp uses a special description file format used to store metadata such as names of replaced and replacement functions.

Intel Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools [2]. Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process [3]. Intel Pin provides an API function capable of replacing whole functions in the attached process using a similar approach as the previously mentioned tools. The framework writes a jump instruction at the beginning of the target function. The instruction points to the replacement function. Intel Pin is robust tool which could be used to design a live patching framework. This was accomplished in article *Hot Patching C/C++ Functions with Intel Pin* [10]. As we can observe, the program using Intel Pin is relatively simple and the article demonstrates its functionality.

Chapter 3

GDB tool

GDB stands for GNU debugger. It is a standard debugging tool for Linux operating system. GDB implements a mechanism using `ptrace` system call allowing to attach to a running process and stop its execution. This can be achieved by running a shell command:

```
gdb -p $process_pid
```

GDB provides a command line interface and a scripting language featuring various commands. These commands can either show diagnostic information or manipulate directly with process's resources, e.g. registers or memory. We can also define so called convenience variables which can store a string, a variable's value or an address. Alternatively, GDB offers a built-in python interpreter and API allowing to write scripts which can be imported into GDB. It even enables us to create user-defined command. All we have to do is implement a subclass derived from API-defined python class `gdb.Command` as the documentation specifies [6]. We need to import a `gdb` package in the script to include API functions and launch GDB with option `--command=path/to/my/script`.

One of the most useful things in GDB is a symbol lookup. Every `ELF` binary by default contains a symbol table which basically consists of key-value pairs of symbol name and its offset from the beginning of the corresponding section. Once the sections get mapped into virtual memory address space, GDB can calculate a symbol's address in memory. We use `print &symbol` command for this purpose. In python, the analogous approach is to call API function `gdb.parse_and_eval("&symbol")`. It returns an instance of `gdb.Value` object which works as a wrapper for the underlying value in the inferior process. On the other hand, there is a GNU utility called `strip` which can be used to compress the binary. As the name implies, all redundant information gets stripped off including string names of functions and global variables. The mentioned commands will, therefore, result in an error for a `strip`ped process.

Moreover, GDB allows to type cast the internal value of `gdb.Value` object just as we can do in C. The API defines `gdb.Type` object representing an inferior's type.

Function `gdb.lookup_type` can be used to query the inferior's types. For instance, a query for 8-byte integer would be:

```
gdb.lookup_type("uint64_t")
```

Therefore, a type cast of `gdb.Value` instance to this type might look like:

```
value.cast(gdb.lookup_type("uint64_t"))
```

When debugging symbols for the process code are provided, we take full advantage of GDB ability to look up symbols. We can make use of command `info address` which maps a global symbol to its address. Command `info symbol` might be useful as well as it does the exact opposite of `info address`, in other words, maps an address to the corresponding symbol. It should be noted that these commands cannot be used on locally defined symbols, e.g. local variables. In python API, we can achieve the same results by using function `gdb.lookup_global_symbol`.

A problem regarding symbols arises when two loaded object files use the same symbol name for their respective symbols. When requesting a lookup for one of these symbols, GDB searches all present object files for the symbol. The tricky part is that in case it finds more than one match, only the first one found gets printed. Fortunately, python API offers the following method:

```
    gdb.lookup_object_file("objfile_name")
```

This orders GDB to iterate through loaded object files and returns a `gdb.ObjFile` object representing object file if the name matches. Class `gdb.ObjFile` contains various methods. The most important is `lookup_global_symbol` which only searches the corresponding object file for the specified symbol.

GDB represents the inferior process with object `gdb.Inferior`. This object offers several methods, the most important being `read_memory` and `write_memory`. As the names imply, we can read or write into the process's memory using these methods. These functions take a buffer and an address as arguments. The buffer can be of python types representing arrays of bytes, e.g. `bytearray` or `bytes` while the address can either be `int` or `gdb.Value` instance cast to a pointer to the corresponding type. Additionally, amount of bytes to be read or written is passed as an integer. Function `gdb.Inferior.read_memory` returns a `memoryview` object containing inferior's memory content.

Command `info proc mappings` prints memory mappings of the inferior process in the same way they are stored in `/proc/process_pid/maps` file. One can use this command to inspect the memory address space and keep track of which object files are mapped and which pages are allocated. For this reason, the operation prints the same output as the following shell command:

```
  cat /proc/$process_pid/maps
```

GDB python API comes with function

```
gdb.execute("gdb command", to_string)
```

which executes the GDB command and allows to pass an additional boolean parameter. If this parameter is `True`, the output of the command will be returned as a python string. Memory mappings can then be inspected by using this function with `info proc mapping` and parsing its output in python.

Lastly, we can use the previously mentioned commands to manipulate with registers of the inferior process. To read a value stored in register `register_name`, we can simply execute the following command:

```
gdb.parse_and_eval("$register_name")
```

The return value is a `gdb.Value` instance representing the value stored in the register. To load a value to register `register_name`, we can run the following command:

```
gdb.execute("set $register_name = %d" % new_value)
```

Chapter 4

Design and implementation

Both Libpulp and bakatsugi described in chapter *Related work* perform live patching and implement it from scratch. We explored ways how to perform live patching on the top of GDB. We designed and implemented a tool capable of replacing entire functions similarly to what bakatsugi does. Moreover, we added a mechanism allowing to reapply or revert a patch. We extended GDB tool by using standard commands along with python API and wrote a script declaring a new set of commands which can be imported into GDB sessions. These commands perform operations needed to apply and maintain patches.

To patch functions of the target process, the user creates a shared library containing the implementation of new functions that will replace the old ones in the process. Additionally, the user must specify pairs that identify the functions to be replaced and replacement functions along with the strategy of patch procedure, e.g. if the target function is native or dynamically linked from a library. This information is encoded and embedded in the library itself. The user passes the path of this library to our command and the patch procedure is executed. In case no error occurs, after detaching GDB from the process, the behavior of the target process should be changed as desired.

We implemented a tracking mechanism which allows the user to print and inspect the process patch history. This is useful for debugging so that the user is able to figure out what patches have been carried out and monitor the active ones. Moreover, the user is able to reactivate logged patches overridden by new ones.

On top of the logging mechanism, we designed and implemented a plugin enabling patch reversion. This means that the user is able to revert the impact of a patch and restore the behavior of the function and ultimately the whole process to the original state. The logging mechanism is turned off by default, but it can be activated by the user. The overview of the extension's architecture is illustrated in Figure 4.1.

We implemented the GDB extension as a python script containing classes that define commands as we explained in *GDB tool*. In the early stages of the development, we used GDB scripting language. However, it turned out to lack expressiveness and some basic and important constructs were tricky to implement. Among many, the most obvious ones were loops, string concatenation and exceptions. Python simplified a lot of things, namely error handling by providing API exceptions `gdb.GdbError` which notify the user by printing an error message.

To sum up, the GDB extension provides the following functionality:

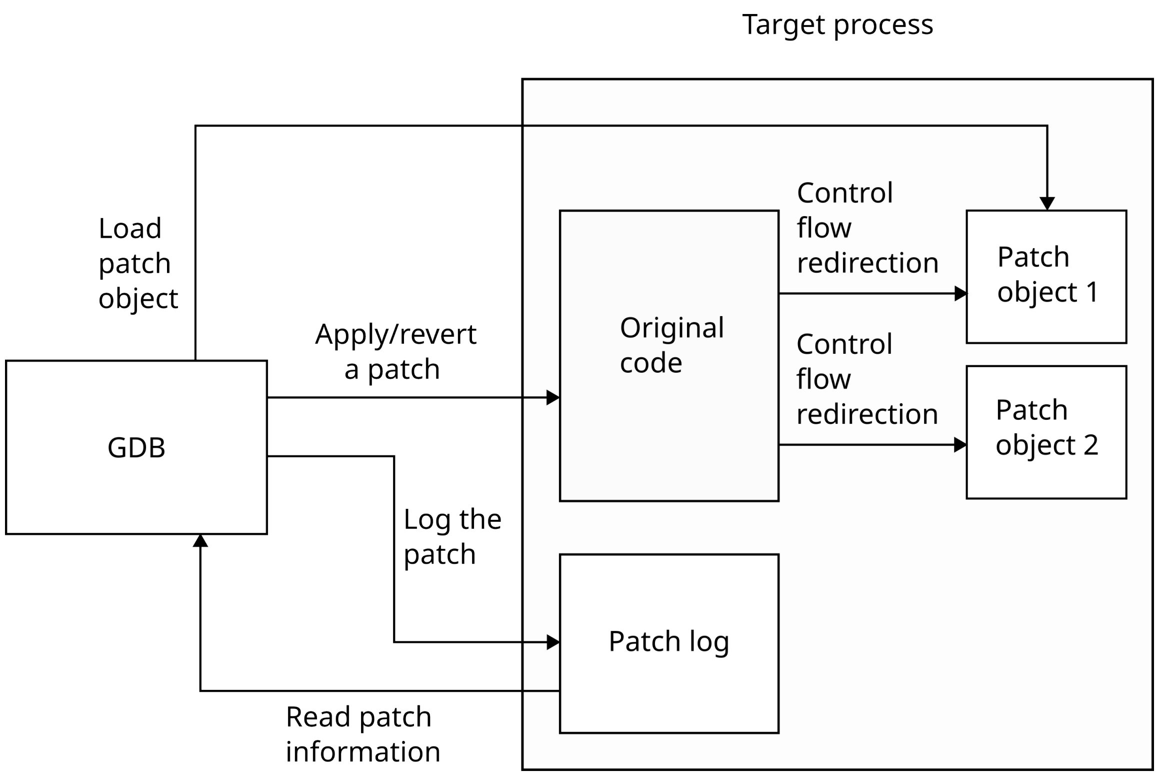
1. replacing entire user-defined or library functions (i.e. functions from shared libraries)

2. logging the applied patches, tracking the active patches and loaded patch objects and providing an interface to print the log and dump it into a file

3. reactivating an inactive patch (e.g. a function was patched again by another patch and the user wishes to activate the previous patch) whose entry is stored in the log

4. reverting a patch whose entry is stored in the log (restoring the original state of the specified function)

Figure 4.1: Diagram of the architecture of the GDB extension



4.1 Patch objects

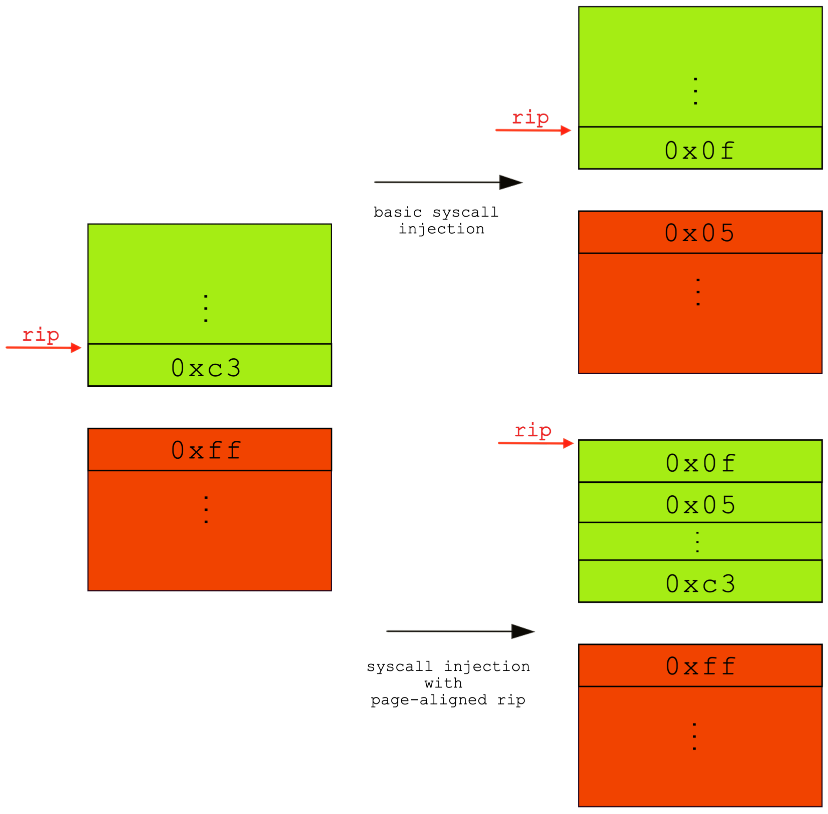
A patch object is a dynamic (shared) library which contains replacement functions. Each patch object contains a special section named `gdbpatch`, which stores patch instructions (i.e. the name of the function to be replaced, the replacement function name and the patching strategy) and structures for patch metadata which are basically arrays of bytes with their respective sizes. The constructor function initializes these structures with predefined values. It is called when the dynamic linker has loaded the object. We explained how to define these components in section *Dynamic linking.*

We provide a C header file which contains macros that generate the special section, the constructor function, the patch instructions and declares metadata structures and other constants such as sizes of the structures and so on. The user must use these macros in the source code of the patch library, otherwise the tool will not be able to locate the metadata. To learn how to build a patch library, see *Appendix B: the user manual.*

4.2 Injecting system calls

When we have seized control of a running process, we can manipulate with its code and registers to achieve a system call execution. The instruction pointer register (`rip` on x86-64) points to the next instruction to be executed. We can obtain its value in GDB python as we described at the end of chapter *GDB tool.* We need to overwrite the memory region pointed to by `rip` with the instruction that triggers a system call execution and make the process execute it. For this purpose, we can use `syscall` instruction available on x86-64 platform. The instruction is encoded as `[0x0f, 0x05]`. The instruction pointer (`rip`) might theoretically point to a 1-byte long instruction (for instance `ret`, encoded as `0xc3`) located at a page boundary. If such an instruction was overwritten, the `syscall` instruction would be split in half between two neighboring pages, see Figure 4.2. The problem would arise, if the page containing the second byte of the `syscall` did not have read or execute permissions. Provided the `syscall` must be executed by the target process, this situation would cause a segmentation violation if the process tried to execute the instruction from a non-readable or non-executable memory page. We decided to solve this issue by simply aligning the `syscall` instruction destination address (the address where the `syscall` will be written) to the beginning of the page to which `rip` points, so that we do not need to worry about the lack of space in the page. The value in `rip` is then aligned in the same way to point to the `syscall`. We described how to load a register value of the target process in python at the end of chapter *GDB tool.* This simple solution prevents any segmentation violation issues. The solution is also illustrated in Figure 4.2.

Figure 4.2: System call injection



To execute the system call, the arguments must be set correctly. Register `rax` is used to store the system call number. Depending on the system call, up to 6 arguments can be passed to it using `rdi`, `rsi`, `rdx`, `r10`, `r8` and `r9` in this order [1]. Firstly, values stored in `rip` and the registers which are used to pass parameters to the system call must be backed up before any modification since the process execution must be resumed after the system call execution. Keep in mind, that `rip` register must also be set to point to the beginning of the page because of the potential issues described above.

To execute the single `syscall` instruction, we make use of GDB command `si` which only executes the next instruction and then returns the control to GDB. With the system call executed, the memory overwritten by `syscall` instruction, `rip` and the registers used for passing the system call number and arguments are restored to their original state. We can now safely resume the process as if nothing happened.

To sum up the procedure, the following algorithm written in python pseudocode is used to inject a system call (functions `backup_registers`, `restore_registers` and `set_arguments_and_syscall_number` are not implemented in the snippet):

```
PAGE_MASK = 0xfffffffffffff000
rip = gdb.parse_and_eval("$rip")
rip_aligned = rip & PAGE_MASK
membackup_bytearray = inferior.read_memory(rip_aligned, 2)
inferior.write_memory(rip_aligned, bytearray.fromhex("0f 05"), 2)
# back up registers
backup_registers()
gdb.execute("set $rip % d" % rip_aligned)
set_arguments_and_syscall_number()
# execute si
gdb.execute("si")
ret = gdb.parse_and_eval("$rax")
# restore registers
restore_registers()
inferior.write_memory(rip_aligned, membackup_bytearray, 2)
```

Our tool contains a function which takes a system call number and 6 system call arguments (or fewer depending on the system call) as parameters and performs the above-mentioned operations to execute system call in the inferior process.

4.3 Patch application

The first command we implemented is named `patch`. It takes a library path as an argument and performs replacement of functions specified inside the patch library. This information is embedded in the patch object and will be described in subsection *Patch instructions*. This command can be invoked inside GDB shell by running

```
patch path/to/patch/library
```

Either an absolute or a relative path might be passed. However, in case of a relative path, `dlopen` might produce an error stating that the patch library file couldn't be found. This occurs when the patch library file is not in the standard library search path such as `/usr/local/bin`. A directory to be searched for dynamic libraries can be added using `LD_LIBRARY_PATH` environment variable.

To apply a patch, GDB must be attached to a running target process, otherwise the patching procedure will throw an exception. We first need to locate `dlopen`, `dlclose` and `dlsym` functions inside the target process. This can be accomplished by running the following commands:

```
    set $DLOPEN_PTR = &dlopen
    set $DLCLOSE_PTR = &dlclose
    set $DLSYM_PTR = &dlsym
```

In python, the same result can be achieved by running commands:

```
dlopen = gdb.parse_and_eval("dlopen")
dlclose = gdb.parse_and_eval("dlclose")
dlsym = gdb.parse_and_eval("dlsym")
```

These API functions return a `gdb.Value` instance which was described in *GDB tool*. The instance represents a function pointer type in this context. Our mechanism relies on the presence of the above-mentioned functions. They are part of standard `libc` library which is mapped in almost every Linux process. However, if a process happens to lack these functions or the tool fails to locate them, the patching procedure is terminated and the process is detached. This is detected by handling an exception thrown by GDB, stating that the symbol is not defined.

The next step is to use `dlopen` to open the patch library. The function can directly be invoked in GDB by using commands (GDB language and python):

```
print dlopen("$PATH_TO_LIBRARY", $FLAGS)
dlopen(path_to_lib, flags)
```

The function is executed in the context of the target process. There is, however, a slight complication with passing arguments to the functions from python. The user passes a string representing an absolute path of the patch library to `patch` command. The console argument is parsed and the path is stored as a python string. Since API allows to execute the inferior's function, we can simply call `dlopen` and pass the python string representing the path as an argument. As the documentation specifies, any arguments provided to the call must match the function's prototype, and must be provided in the order specified by that prototype [6]. This means that `gdb.Value` instance passed as an argument must be cast to a type matching the signature of the function. Python's representation of strings is different than that of C, specifically missing the notorious null terminating byte. If we pass the python string without any conversion to `dlopen`, it might lead to an undefined behavior which may ultimately result in a crash of the process. We solved this problem by wrapping the string by `gdb.Value` object and appending a null terminating byte to it. Additionally, this instance needs to be type cast to `char[]` to match the C string representation. We can now safely call `dlopen` function inside the target process to load the patch library. We save the handle returned by `dlopen` for later use.

With the module and the patch functions loaded, we now have to redirect control flow from the target function to the patch function. This is done in two ways depending on the fact if the target function is present in the main program or dynamically linked to it.

4.3.1 Function replacement

When dealing with patching user-defined functions, we first need to determine the address of the target function (the one being replaced). We use the same procedure as we did with library handling functions (GDB language and python):

```
set $TARGET_FUNCTION_PTR = print &target_function
target_function_ptr = gdb.parse_and_eval("target_function")
```

Another component is the patch (replacement) function address since we do not know beforehand where the dynamic linker maps the patch library. Initially, we intended to use `gdb.Objfile.lookup_global_symbol` method which we described in chapter *GDB tool*. We could search the corresponding object file for the global symbol (non-static function symbol names are global by default). This way, we could avoid symbol name collisions across multiple patch libraries since GDB would only search the specified object file. Yet, this method did not work as intended since we encountered a bug during the development. If more object files contain the same symbol name, only one of them is returned every time regardless of the searched object file. So instead, we use `dlsym` to search for a symbol in a patch library. We pass the library handle and the symbol name to it and invoke the function the same way we explained with `dlopen`. The symbol name must again be converted to the C string representation. If the symbol is defined in the library, its address in the process memory is returned. Otherwise, `NULL` value is returned. The trampoline which redirects the control flow to the replacement function can now be constructed and written inside the process's memory with `gdb.Inferior.write_memory()` method mentioned in *GDB tool*.

The first type of trampoline is the absolute trampoline:

```
movabs $patch_func_ptr, %r11
jmp *(%r11)
```

This piece of code is written at the beginning of the original function which ensures that the control flow is passed to the replacement function once the original function is called. We decided to use `r11` register for this purpose because it does not have to be preserved across functions calls [12]. When encoded into machine code, the above code snippet is 13 bytes long. We cannot directly use x86-64 relative `jmp` instruction for trampolines since it uses 32-bit relative address mode which means that the jump is performed relatively from the next instruction pointed to by the instruction pointer (`rip`). We rely on `dlopen` and the dynamic linker, therefore, we cannot make any assumptions about the replacement function's address. In other words, the distance between the original function and the replacement function could theoretically be larger than $2^{31} - 1$ which would not fit into 32-bit relative offset. Moreover, x86-64 does not provide any absolute jump instruction which would effectively solve the problem. We have no other option than to use the above code.

The second type is the relative trampoline. As mentioned, the dynamic linker might place the replacement function too far from the replaced one. This rules out the direct use of relative `jmp` instruction. To overcome this, we use the same approach introduced in bakatsugi [12]. We first search the process address space for the closest free page to the trampoline destination (the beginning of the original function). This can be achieved by parsing the output of GDB command

```
info proc mappings
```

explained in *GDB tool*. We need to allocate this memory page inside the target process. We use `mmap` system call which is injected into the target process. System call injection was explained in subsection *Injecting system calls*. The allocated memory page puts no restrictions on trampoline's length, which means an intermediary absolute trampoline can be written there. A short relative trampoline pointing to the beginning of the corresponding intermediary trampoline is then constructed. We must ensure the allocated page is close enough to the target function since we use `jmp` which uses 32-bit relative address. In other words, a free page is searched for in the following range:

$$[\texttt{target\_function\_address} - 2^{31} + 1, \texttt{target\_function\_address} + 2^{31} - 1]$$

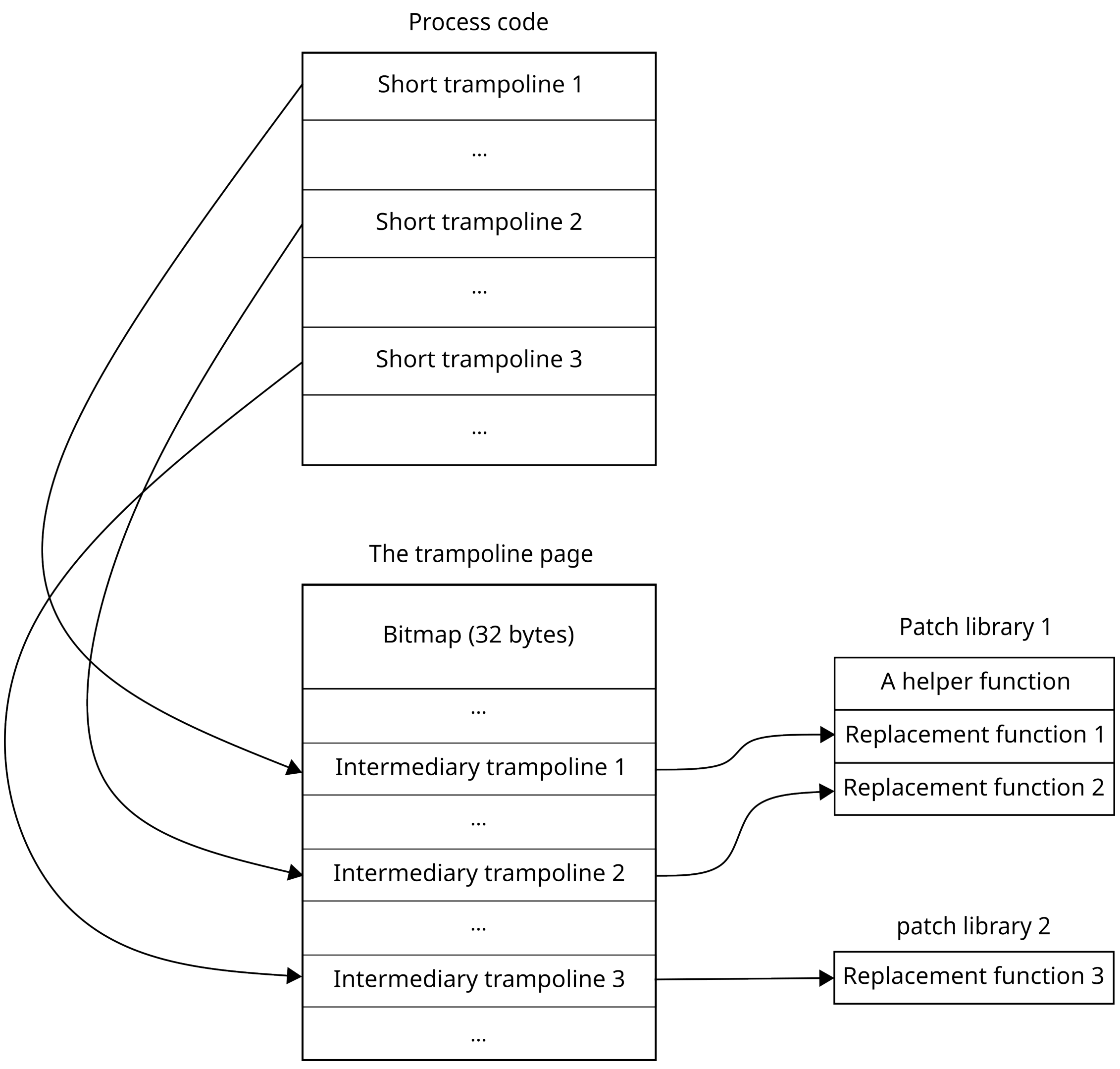
It is highly likely that we will always be able to find a free page in the search range. Even if no free page in the range is found, the script falls back to the absolute trampoline. This approach comes with its drawback, the most obvious being the jump overhead since an additional instruction must be executed as opposed to the absolute trampoline. On the other hand, the most notable advantage is the relative jump instruction length which is only 5 bytes long as opposed to the absolute trampoline which is 13 byte long. Moreover, most functions use function prologue and epilogue demonstrated by the following code snippet:

```
#function prologue
push %rbp
mov %rsp, %rbp
#function body
...
#function epilogue
leave
ret
```

This code itself without a function body is 6 byte long. This means that there is no need to worry that code of a function located right beyond this one will overlap with the written trampoline.

As we explained, a short trampoline jumps to the beginning of the corresponding intermediary trampoline which finally jumps to the replacement function. As suggested, to store these intermediary trampolines, we allocate a memory page inside the target process. From now on, we will call this page the trampoline page. A memory page allows to accommodate 315 absolute trampolines. We decided to align the absolute trampoline structure to 16 bytes padding them with 3 `nop` instructions (these will never be executed nevertheless). Hence, the trampoline page can fit 256 aligned absolute trampolines. For the reasons which we will explain later in chapter *Repeated patching*, the tool implements intermediary trampoline allocation and freeing. This is done by placing a 32-byte long bitmap in the beginning of the trampoline page. The bitmap indicates whether a trampoline entry is allocated or free. The final trampoline count the page can accommodate is 254 which is highly sufficient. The short trampoline mechanism is illustrated in Figure 4.3.

Figure 4.3: Redirection using intermediary trampolines



In terms of replacing library functions, we need to overwrite the corresponding `.got.plt` entry (see section *Dynamic linking*). A `.got.plt` entry address is encoded inside the indirect jump instruction used to invoke the dynamic linker when the library function is called for the first time. To introduce an example, see the following instruction:

```
jmp *0x2fe2(%rip)
```

This instruction is encoded as `ff 25 e2 2f 00 00` in hexadecimal. When executing this instruction, the `rip` register always points to the next instruction which is located right beyond this one. We can now calculate the address of the corresponding `.got.plt` entry (which is `[rip + 0x2fe2]` in this example) and edit it accordingly. This strategy is less error prone when compared to the trampoline approach since we do not modify any code. This means that even if the function was on the call stack, the patching procedure would not cause any error.

4.3.2 Patch instructions

Patch objects (libraries) contain section `gdbpatch` which stores instructions for the patch procedure. They specify the type of operation to be performed, the function to be replaced, the replacement function and what type of trampoline to use. The format is based on the one used in bakatsugi [12] with slight modifications. To introduce an example, to replace function `f_old` with `f_new`, the user must declare and implement `f_new` function and specify the following macro in their patch library source code (depending on whether `f_old` is a native or library function and what type of trampoline is to be used):

```
PATCH(PATCH_OWN_SHORT(f_old, f_new))
PATCH(PATCH_OWN_LONG(f_old, f_new))
PATCH(PATCH_LIB(f_old, f_new))
```

The inner macro represents the patch instructions and is expanded as follows:

```
O:S:f_old:f_new;
O:L:f_old:f_new:
L:N:f_old:f_new;
```

Option `O` means an own function is about to be patched while `L` represents a library one. Option `S` stands for short trampoline, `L` for a long trampoline and `N` for no trampoline which can only be used with library function patching. These instructions are extracted and parsed when the patch object is opened. Before any patch application, a check is performed on whether the patch instructions are valid, e.g. if the original and replacement functions can be located or whether it contains valid options. If any error occurs, the library is closed and the patching process is terminated.

The `PATCH()` macro declares the `gdbpatch` section, the constructor function for the patch library and places the patch instructions into the `gdbpatch` section.

Project directory `examples` contains several patch libraries demonstrating use of patch instructions. See *Appendix B: the user manual* to learn how to compile and build patch libraries.

4.4 Repeated patching

This section deals with tracking and maintaining patches, in other words tracking states of applied patches (e.g. whether they are active), loaded patch objects and logging the patch history throughout the process lifetime. We will explain how patches can be reverted and how the tool implements this feature. Additionally, we will describe how to store metadata which is necessary to provide the aforementioned features. The biggest challenge of repeated patching was to come up with a mechanism that would be consistent. The following questions had to be answered. What happens if an already patched function gets patched once again? What to do with the libraries when they become unused? How could the user inspect the current state of the process? Where to store metadata? How to perform patch reversion and free used resources? We decided to come up a following workflow.

Each time a new function patch is applied and the user wishes to log it, we mark an entry to a so called patch log which contains various items such as the patch time stamp, the original function address and the replacement function address. The user can print the log using GDB command which we will describe in *Patch log*. When a patch of a function is applied, we linearly search the log to fetch the active entry representing the patch of the same function. The entry is set as inactive and changes are written to the log.

Furthermore, every shared library holds a value representing its reference count. In other words, how many functions from the target process are currently patched by functions from the library. Whenever a function gets patched again by another library, the former library reference count is decremented. The library is then kept open until its reference count reaches zero. The user might wish not to close the library even despite the library not being used, i.e. its reference count is equal to zero. In other words, a future improvement might enable the user to specify that a certain library remains in the process's memory despite not being used. Additionally, an option to explicitly close a library could be added.

The logging may sometimes complicate and slow down the patching process, so it is turned off by default and only carried out when explicitly ordered by the user. It should be noted that if a patch is not logged, it will not be possible to restore the original state of the original function.

4.4.1 Patch log

We implemented GDB command `patch-log` which simply prints the log entries sorted by date of their application and highlights the active patches. The following example demonstrates what a log instance may look like:

```
$ gdb -p $pid --command=project_path/src/commands.py
(gdb) patch-log
[0] revert
[1] 2023-03-22 12:02:18: target_function -> /tmp/patch.so:dec
[2]* 2023-03-22 12:02:18: printf -> /tmp/patch.so:pretty_printer
[3]* 2023-03-22 12:02:47: target_function -> /tmp/mult.so:mult
```

The zeroth item is a special one used for patch reversion which will be described in section *Patch reapplication and reversion*. The first item indicates that the function at address `target_function` was patched at the specified time and replaced by function `dec` from `/tmp/dec.so` library. The `*` character means that the entry represents an active patch.

In addition, we implemented `patch-dump` command which can be used to dump the patch log into a file. This may be useful when the user is about to restore the initial process state and wants to keep track of changes made. The log is then written to the file whose path is passed as a parameter to the command.

4.4.2 Storing patch metadata

The biggest challenge of storing metadata was persistence. In other words, how to ensure that necessary metadata structures remain intact and persist in memory even after detaching GDB from the process. Furthermore, GDB must be able to locate these structures when attaching to to the process. The patch metadata must exist throughout the whole life of the process in case another action were to be performed later. The only exception is when the original process state is restored, e.g. by reverting all patches. When we detach from the process and quit GDB, all data stored in GDB will be lost. This means that we cannot store any metadata in GDB itself. One solution that naturally springs to mind is storing the data in a separate file as Libpulp does [11]. This would mean that the user would always have to pass the path to the database file as a parameter. In addition, error handling would have to be more complex if, for instance, the file got deleted.

A more promising approach is to allocate a memory page inside the inferior process using `mmap` system call injection which was described in *Injecting system calls*. Unlike the previous idea, this one does not require any external resources. Anyway, this approach still does not meet the criterion of metadata persistence. We cannot assume the page address beforehand since it is up to the operating system where the page will be allocated. It is possible to specify the address to be allocated in `mmap` system call, but it is only used as a hint to the kernel. Furthermore, this page might, of course, already be occupied. In such a case, the kernel picks whatever address it finds suitable. This means that the address of the metadata page cannot be assumed as the memory address space layout varies depending on the process. This leads to the conclusion that a pointer to the page must be stored somewhere as we need to ensure metadata persistence. A good candidate that can be used as a storage for the pointers seems to be the special section of a patch library. It is a memory region which is available to the process as long as the corresponding patch library remains opened. Patch libraries store patch instruction inside a special section as we explained in *Patch instructions*. This space can be used to allocate structures for patch metadata besides patch instructions. This is the approach we chose to implement in our tool.

Each loaded patch library has its own special section mapped to the target process memory. Only one of them actually contains the patch metadata (or the pointers to them) mentioned above. We call this library the master library. Every time we attach to a process with GDB, we need to scan every loaded object and identify the master library. When quitting GDB, we lose all the information about shared libraries such as their handles returned from `dlopen` or path to the corresponding file. This data is crucial for the subsequent process manipulation in case we reattach to the process. We worked around it by using the following GDB API function:

```
gdb.objfiles()
```

This function returns a list of `gdb.Objfile` objects which represent loaded object files. This allows to iterates through all loaded objects including the patch libraries. To sum up, each time we attach to a process and the user invokes one of our commands, we search the loaded shared objects for the master library. We filter out non-patch libraries by checking if they contain the special section. If they do, we perform a further check by reading the magic value from the beginning of the special section. These checks increase the likelihood that the object file represents a patch library. In case none is found, the process is in its original state and no patches are active. Furthermore, if a special flag in the library header is set, it means that the master library has been found. Not to mention, it is possible that the master library reference count reaches zero which indicates that it should be closed. However, other patch objects may still be loaded. If we simply closed the master library, the pointers to the metadata pages and other crucial metadata would be lost. Therefore, if the master library is to be closed, we iterate through the loaded patch objects as described above, select a new master library and copy the process-wide metadata (e.g. the log page pointer) from the former master library to the new one.

4.4.3 Metadata Structure

Each patch library contains a patch header which is a simple data structure with size of 48 bytes holding information about the patch library itself. This is what the structure declaration would look like in C:

```
struct patch_header{
        uint32_t magic_const;
        void *libhandle;
        void *trampoline_page_ptr;
        void *log_page_ptr;
        void *patch_data_page_ptr;
        uint16_t refcount;
        uint16_t is_master_lib;
        uint16_t log_entries_count;
        uint16_t patch_data_array_len;
        uint32_t commands_len;
};
```

- `magic_const` - contains a special randomly generated constant, which is used by the python script to identify whether the shared object represents a patch library or not
- `libhandle` - stores the handle to this library returned by `dlopen` and is passed as a parameter to `dlsym` when requesting a lookup of the library symbol
- `trampoline_page_ptr` - a pointer to the trampoline page which we explained in subsection *Function replacement*
- `log_page_ptr` - a pointer to the memory area containing the patch log
- `patch_data_page_ptr` - a pointer to the page containing the additional information to the corresponding log entry, we will explain this in more detail in subsection *Patch log structure*
- `refcount` - used to track references (how many functions are currently patched by this library)

- `is_master_lib` - a boolean value indicating whether the library is the master library (the one containing pointers to the trampoline page, log page and other patch data)
- `log_entries_count` - the count of the entries in the log
- `patch_data_array_len` - the length (in bytes) of the array of the additional patch information
- `commands_len` - the length of the patch instruction string

We could use `dlsym` function to locate the header inside a patch library. There is, however, a problem which we had to bypass. To use `dlsym` for a library, we need a handle to it returned by `dlopen`. Since we have to take persistence into account, the handle is lost every time we detach GDB from the process. The handle is stored inside the header and we need the handle to locate the header. To locate the header, we firstly intended to use the following method:

```
gdb.Objfile.lookup_global_symbol("patch_header")
```

As we mentioned earlier in *Function replacement*, this function contains a bug and we kept receiving the address of the header from another library if more patch libraries were loaded. We bypassed this error using similar function

```
objfile.lookup_static_symbol("symbol_name")
```

which worked but forced us to declare the header as a static symbol which was no concern in our case since the inferior process does not use the header outside of the library constructor function. Moreover, this function requires debugging symbol in order to work. For this reason, patch libraries must be compiled with debugging symbols.

4.4.4 Patch log structure

In the earlier stages of the development, we used master library's special section to store the whole log since it could easily be stored inside the patch library using `dlsym` function. We declared an array of log entries in the special section of each patch library. Only the master library contained the real log as long as any of its function was active. This approach was valid and worked. There were two major drawbacks. Firstly, the special section was larger, since we allocated three pages for the log. Because every patch library contains the special section, three pages per every library were wasted. This was not so much of a concern considering the fact that the average user would only need to apply few patches. Secondly, in case the master library was to be closed, the log would have to be copied to the section of another page library. This lead us to change the implementation and store the log in a separate memory area of the inferior process just like the trampoline page. A pointer to it is stored in the header of the master library. See `log_page_ptr` member in the header structure in subsection *Metadata Structure*.

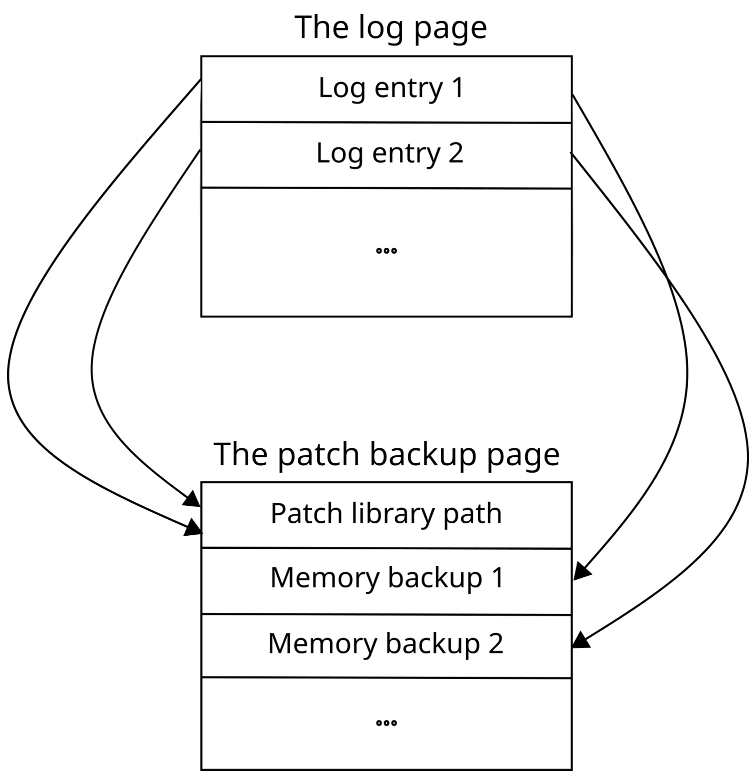
The log page contains an array of log entry structures. This structure's size is 32 bytes and is defined as follows:

```
struct log_entry{
        void *target_func_ptr;
        void *patch_func_ptr;
        char patch_type;
        uint32_t timestamp;
        uint8_t is_active;
        uint32_t path_offset;
        uint32_t membackup_offset;
        uint16_t path_len;
        uint16_t membackup_len;
}
```

- `target_func_ptr` - a pointer to the original function
- `patch_func_ptr` - a pointer to the replacement function
- `patch_type` - indicates what patching strategy was used (whether the original function is native or from dynamic library)
- `timestamp` - timestamp of the patch application
- `is_active` - indicates whether the patch is active

The log is stored in the log page and a pointer to it is stored in the master library header, see `log_page_ptr` in the header structure which was described at the beginning of subsection *Metadata Structure*. The patch metadata contains data with variable sizes, namely the path of the patch library file and backup of memory overwritten by a trampoline or the original `.got.plt` entry data (this memory content is stored so that the patch reversion mechanism which will be described in section *Patch reapplication and reversion* can be implemented). These data are stored separately from the rest of the log entry items in a special page a pointer to which is stored in the header of the master library, see `patch_backup_page_ptr` in the header structure. The log entry contains two members `path_offset` and `membackup_offset`. These represent the offset of path and memory backup respectively in the page. Members `path_len` and `membackup_len` represent the length of the path and the memory backup. This design allows to share references. In other words, if two log entries represent patches from the same library, they point to the same string representing their patch library path. See Figure 4.4 for an illustration. Memory backup can be shared as well. This is made use of when applying a new patch instance to a function that has already been patched. The script searches the log for an old patch of the currently patched function and copies the reference to its memory backup.

Figure 4.4: Log structure



4.5 Patch reapplication and reversion

Our tool enables users to reapply (reactivate) logged patches. After printing and inspecting the log (see subsection *Patch log*), the user might wish to return to a previous state of a function assuming it has been patched more than once. We implemented `patch-reapply` command for this purpose. The usage is as follows:

```
$ gdb -p $pid --command=project_path/src/commands.py
(gdb) patch-log
[0] revert
[1] 2023-03-22 12:02:18: target_function -> /tmp/dec.so:decrement
[2]* 2023-03-22 12:02:37: target_function -> /tmp/mult.so:multiply
[3]* 2023-03-22 12:02:47: print_value -> /tmp/printer.so:pretty_print
(gdb) patch-reapply 1
(gdb) patch-log
[0] revert
[1]* 2023-03-22 12:02:18: target_function -> /tmp/dec.so:decrement
[2] 2023-03-22 12:02:37: target_function -> /tmp/mult.so:multiply
[3]* 2023-03-22 12:02:47: print_value -> /tmp/printer.so:pretty_print
```

The user specifies a number from the log list which is interpreted as an index of the log entry to be reapplied. Internally, the patch script reads the metadata for the entry. This includes the target and patch function addresses. All we have to do is recreate the corresponding trampoline or `.got.plt` entry and trigger the patching procedure described at the beginning of the chapter *Design and implementation.*

The zeroth item represents patch reversion. The user must specify `0` as a parameter along with a list of functions to be reverted. If no function is specified, every patch is reverted and the process is returned to the state it was before applying the first patch. The following example demonstrates the command usage:

```
(gdb) patch-reapply 0 function1 function2 function3
```

The implementation is straightforward. Firstly, the active log entry patching the input function is searched for. Next, the memory backup mentioned in subsection *Patch log structure* is fetched. If the trampoline was used, it has to be detected whether it was a short trampoline or a long one. This can easily be detected by reading the memory occupied by the trampoline and read the instruction prefix (`0xe9` is used in the short and `0x49` in the long trampoline). If a short trampoline was used, the long intermediary trampoline to which the short one jumps gets freed in the trampoline page bitmap. It should be noted, each time the first patch of a function is applied, we back up 13 bytes of the beginning of the function regardless of what type of trampoline is to be used. Imagine a situation that a short trampoline is used for the first patch. The second patch of the same function is about to use a long trampoline. If we only backed up the first 5 bytes of the function when applying the first patch, we would have to make a longer backup storing 13 bytes overwritten by the long trampoline. Hence, we always directly back up 13 bytes. Lastly, we restore the memory modified by the patch execution (the trampoline or `.got.plt` entry overwriting), mark the log entry as inactive, decrease the corresponding library's reference count and detach from the process.

Chapter 5

Verifying the results

We have tested our tool in several test cases. All examples can be found in `examples` directory of the project, see appendix *Appendix A: the source code.* Each test case contains a `Makefile` that builds the whole testing program. The tests have been carried out for every component.

Firstly, we introduce a simple example similar to those used in bakatsugi [12] to test patch application and repeated patching mechanisms. Attaching with GDB is required and the python script must be imported as well. For further details, see *Appendix B: the user manual.*

5.1 Simple counter

The first simple demonstration is the following program:

```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>

void target_function(uint64_t *x){
        *x += 1;
}

int main(void){
        uint64_t x = 0;

        while(1){
                print_value(x);
                target_function(&x);
                sleep(1);
        }

        return 0;
}
```

As we can see, the program increments a counter and prints it to the standard output every second without any formatting. We created two separate patch libraries. The first patches native `target_function` and `printf` from standard `libc`, while the second only patches the `target_function`.

The first patch library is as follows:

```
#include "patch.h"
#include <stdint.h>
#include <stdio.h>

void patch_function(uint64_t *x){
        *x -= 1;
}

void pretty_printer(
        const char *old_format_string,
        uint64_t x
){
        printf("The current value is: %lu\n", x);
}

PATCH(
        PATCH_OWN_SHORT(target_function,
                                        patch_function)
        PATCH_LIB(print_value, pretty_printer)
)
```

This patch ensures that the counter is decremented in each iteration and is printed using a different formatting. In `pretty_printer` function, the original formatting string passed from the original `printf` call is ignored and the integer is printed using a new formatting string. To sum up, we are patching a user-defined function with a short trampoline and a function from a dynamic library. Since our patching tool only replaces functions in the main executable and every loaded object has its own `.got.plt` table, we can use ordinary `printf` function in our patch library.

The second patch library is as follows:

```
#include "patch.h"
#include <stdint.h>

void mult(uint64_t *x){
        *x *= 2;
}

PATCH(
        PATCH_OWN_SHORT(target_function, mult)
)
```

This patch function multiplies the counter value by 2 in each iteration. The short trampoline approach is to be used.

The output of the test program is shown in Figure 5.1. We attach to the process with

```
gdb -p $pid --command=project_path/src/commands.py
```

We apply patches from the first library with logging turned on, print the log and detach from the process:

```
(gdb) patch /tmp/patch.so --log
(gdb) patch-log
[0] revert
[1]* 2023-05-10 12:32:56: target_function ->
/tmp/patch.so:patch_function
[2]* 2023-05-10 12:32:56: printf -> /tmp/patch.so:pretty_printer
(gdb) detach
```

Lines 1-5 in Figure 5.1 were printed before the patch execution. The counter is now decremented in each iteration and the printing format has also changed as we can see in lines 6-10. Now, we apply the second patch with logging turned on.

```
(gdb) patch examples/more/mult.so --log
(gdb) patch-log
[0] revert
[1] 2023-05-10 12:32:56: target_function ->
/tmp/patch.so:patch_function
[2]* 2023-05-10 12:32:56: printf -> /tmp/patch.so:pretty_printer
[3]* 2023-05-10 12:35:07: target_function -> /tmp/mult.so:mult
(gdb) detach
```

The `mult` function patches the same function as the `patch_function`. This resulted in deactivation of the `patch_function` which can be seen in the changes in the printed log. We can now see that the counter is multiplied by two in each iteration and the printing function is preserved (Lines 10-19).

We can now print the log and reactivate `patch_function`:

```
(gdb) patch-log
[0] revert
[1] 2023-05-10 12:32:56: target_function ->
/tmp/patch.so:patch_function
[2]* 2023-05-10 12:32:56: printf -> /tmp/patch.so:pretty_printer
[3]* 2023-05-10 12:35:07: target_function -> /tmp/mult.so:mult
(gdb) patch-reapply 1
```

We can see that the counter is again decremented in every iteration in lines 19-25. Lastly, to restore the process to its original state, we run:

```
(gdb) patch-reapply 0
(gdb) patch-log
Couldn't find the log. No patch applied.
(gdb) detach
```

We can notice that all log data is deleted. This is due to the fact that all the patch libraries became unused and were unmapped along with pages containing the patch metadata. We can verify that the original process behavior is now restored based on the program output in lines 26-30.

```
0
1
2
3
4
The current value is: 5
The current value is: 4
The current value is: 3
The current value is: 2
The current value is: 1
The current value is: 2
The current value is: 4
The current value is: 8
The current value is: 16
The current value is: 32
The current value is: 64
The current value is: 128
The current value is: 256
The current value is: 512
The current value is: 511
The current value is: 510
The current value is: 509
The current value is: 508
The current value is: 507
The current value is: 506
507
508
509
510
511
```

Figure 5.1: The output of the testing program

5.2 SSH daemon

SSH daemon is a system process running with root privileges. SSH, short for secure shell, is a protocol allowing users to connect to a remote machine and execute commands. UNIX systems use OpenSSH implementation [5]. SSH daemon is a service that listens and waits for the incoming requests. An authentication is needed when a user makes a request. This can be done via a user password. Once the user wishes to establish a connection and makes a request, listening SSH daemon executes `fork` system call to create a child process that takes care of the communication between the user and the server. SSH daemon contains `auth_password` function which authenticates users. It works with users' passwords in plain text and simply returns a boolean value indicating whether the user was authenticated. In this example, we use GDB to replace the function. The replacement function receives a password in plain text as an argument and writes it into a file with proper read permissions so that we can later inspect it. Moreover, the function authenticates every user regardless of the provided passwords by always returning a non-zero value. An unaware user types in their password and it gets written into the file and the user is authenticated as if nothing happened. It should be noted that this example should rather be considered as an anti-example of how to use this tool.

For security reasons, the newly `fork`ed process executes `exec` system call. This means that even if we patch SSH daemon, its forked child will not preserve the changes made during the patching procedure since `exec` replaces the process image with a new one. We, therefore, have to initiate a connection which results in a new process creation and then attach to it with GDB and perform the changes.

Furthermore, SSH daemon is protected by SELinux which is a security enhancement to Linux which allows users and administrators to have more control over access to processes or files [9]. To be able to patch the SSH daemon, we have to set SELinux roles of the patch object file (compiled library) to match those of SSH daemon process. This is performed in the `Makefile` in `examples/ssh/` directory. Lastly, we need access to a root account (or a `sudoer` account and run GDB with `sudo` command) to attach to a running SSH daemon instance with GDB since it runs with root privileges. We created a fake user account for the demonstration. The user name for this account is `user` and the password is `user` as well.

In Linux distributions using systemd, SSH daemon can be started with one of the following commands:

```
(debian) systemctl start ssh
(redhat) systemctl start sshd
```

First, we send request from the local machine using:

```
ssh user@localhost
```

We are prompted to type in the password. This is the time to replace the authentication function. We launch GDB from another terminal.

```
sudo gdb -p $sshd_child_pid --command=project_root/src/commands.py
(gdb) patch examples/ssh/evil_auth.so --log
(gdb) patch-log
[0] revert
[1]* 2023-05-14 10:38:21: auth_password ->
/tmp/evil_auth.so:evil_auth_password
(gdb) detach
```

With the function patched, we can now enter whatever password we want and should get authenticated.

```
$ ssh user@localhost
user@localhost’s password: password
Last login: Wed May 10 13:23:38 2023 from ::1
[user ~]$ whoami
user
```

Lastly, we can print the content of the file where the passwords are saved. Path `/tmp/passwords.txt` is hard-coded in the patch library.

```
$ cat /tmp/passwords.txt
password
```

This example including the patch library and the `Makefile` can be found in directory `examples/ssh` in the project repository. Further examples can be found in `examples` directory, see *Appendix A: the source code.* The tool performed the patches successfully on these examples. However, there are cases in which the tool might not work correctly, therefore, it should not be used in critical applications. We will describe the limitations of the tool in chapter *Known problems.*

Chapter 6

Known problems

6.1 Patch library name collision

There is one issue we encountered when working with `dlopen`. This function takes a file path as an argument, loads the library and increases its reference count. This means that even if the file gets unlinked in the file system, the file will remain on the disk. If `dlopen` gets called with the same path which is now pointing to a new library, the reference count of the old library gets increased and the new one does not get loaded at all. This is not a bug but a feature of `dlopen`. A solution to this issue is introduced in bakatsugi [12] where the injector (GDB in our case) opens the file and only passes the corresponding file descriptor to the inferior process via a UNIX socket which is then opened using `/proc/self/fd` path. This is hard to implement in GDB so we did not include this feature in our tool. Therefore, it is up to the user to track names of used libraries to avoid name collisions.

6.2 Symbol lookup

We decided to analyze possible ways to use GDB to implement a live patching tool. The main reason was its ability to search the inferior process for symbols. The library function we relied on such as `dlopen`, `dlsym` or `dlclose` can be looked up by running:

```
print &dlopen
gdb.parse_and_eval("dlopen")
```

This simple approach, however, requires the user to have `glibc` debugging symbols downloaded. Moreover, the above commands search for the symbols locally, in other words in the current program scope. This means that if a file other than `libc.so` contained `dlopen` function declaration and the execution was inside this file, the address of this user-defined function would be returned. Furthermore, if the main program was compiled with debugging symbols, even locally defined symbols (e.g. local variables in the current scope) would be found first. For example, if someone defined a local variable with name `dlopen`, its address would be returned instead of the desired function. These scenarios are, however, not likely to happen. There is a possible improvement involving using yet another API function:

```
gdb.lookup_global_symbol("symbol_name")
```

This function searches the entire process for a global symbol. Library `libc.so` exports `___dlopen`, `__dlclose` and `___dlsym` symbols which represent the corresponding functions. These symbols can be passed as arguments to the aforementioned API function which returns their address. This approach might be more reasonable since it avoids collision with locally defined symbols, e.g. local variables. Our implementation allows to easily change the way the `dl` functions are searched for.

Another problem arises, when the user declares a function in a patch library whose symbol name is already declared in the main program. After loading the patch library into the memory, GDB might return the address of the newly loaded function instead of the one in the main program when doing a lookup. Hence, the user must guarantee that every symbol name used in patch libraries must not be used in the main executable. This limitation was no concern in bakatsugi [12] thanks to the fact that the main program executable was parsed every time a symbol from the main binary was being searched for. We could overcome the issue by using an `ELF` parser in our python code. This approach would add complexity to our solution which is undesirable since we chose to use GDB to simplify our work. Due to the issues mentioned above, we strongly advise users to retain the name of the original functions and only preface it with a prefix or append a suffix to it to create the name for the replacement function. To introduce an example, a pair of functions could be named as follows using suffix `new`:

```
find_shortest_path -> find_shortest_path_new
```

However, symbol names can be shared across the patch libraries since their addresses are retrieved using `dlsym` function which is only restricted to the shared object scope.

6.3 Trampoline length

The absolute trampoline is 13 byte long. It is not hard to write such a function that would be shorter than 13 bytes (the shortest function on x86-64 can theoretically be just 1 byte long containing only a single `ret` instruction), although, compilers usually align functions to 16 bytes. If the patched function happened to be shorter than the critical value and its beginning got overwritten by a trampoline, the memory beyond the function would be modified as well. This might result in a crash if another function located right beyond the patched one got called. The short trampoline mechanism explained in section *Patch application* reduces the risk of the process crash due to the shorter length of the short trampoline. Not to mention, detecting a function's length is a non-trivial task. Therefore, our tool does not perform any checks regarding this problem.

The next issue is caused by the fact that we do not make any assumptions about when the process execution is interrupted. If the function about to be patched was being executed and we overwrote its memory region, it might lead to an undefined behavior. If we overwrote a memory region with a trampoline and the instruction pointer pointed to the region and we resumed the process, the executed instruction would likely be corrupted and the process would get terminated. This can, however, be checked with ease simply by reading the instruction pointer value and comparing it to the trampoline destination address.

There is another issue which is non-trivial to detect. For this reason, we do not implement any countermeasures. We use Figure 6.1 to demonstrate the potential problem. Imagine we stopped the process while executing the loop on lines 6-9. Even if we applied the short trampoline, the process would likely crash. The function prologue (lines 2-3) is 4 byte long and the trampoline would overwrite the first byte of `test` instruction. When the process is resumed and the next loop iteration is executed, CPU will fetch a corrupted instruction.

```
1   prologue:
2   push %rbp
3   mov %rsp, %rbp
4
5   loop:
6   test %rsi, %rsi
7   jz epilogue
8   inc %rdi
9   dec %rsi
10
11  epilogue:
12  mov %rbp, %rsp
13  pop %rbp
14  ret
```

Figure 6.1: An example of a problematic function

6.4 Trampoline page distance

The trampoline page allocation is deferred for the first patch application. The script tries to find the closest free page to the currently patched function as we explained in subsection *Function replacement*. If found, the long intermediary trampolines are always placed inside it. If all patched function were close to one another, this approach would be valid. However, the process code might be very large and the distance between the trampoline page and other functions to be patched may be higher and this distance would not fit in the short trampoline 32-bit relative offset. We will introduce a following solution which is not implemented in our tool. Every time a patch is about to be applied, a check on whether the function is close enough to the trampoline page would be performed. If not, a new trampoline page would be allocated. This would require to store a list of pointers to trampoline pages instead of a single trampoline page pointer which is not much of a complication. Currently, the check on distance is performed and the script falls back to the absolute trampoline when the trampoline page is too far from the function to be replaced. The user might wish not to perform the patch at all if anything fails. A future improvement might be to make this option configurable. In other words, the user could set that no operation will be performed in case of a failure.

6.5 Global variables

As long as the target process lists its global variables in the dynamic symbol table, these variables can be used by loaded shared objects. They simply need to be declared as `extern` in the library source code. The dynamic linker performs needed relocations when loading the object. Compiler `gcc` and linker `ld` do not export global variables by default. This means that in such a case, the replacement functions will not be able to access the global variables of the target process [12].

Conclusion

We explored and analyzed ways to perform live patching using GDB and built a tool described in the thesis. We analyzed the existing solutions to live patching, proposed improvements and added new functionalities. We based the thesis on *Modification of a process code at runtime* [12] and implemented the functionality of bakatsugi as a GDB extension and evaluated its performance and properties. Moreover, we designed and implemented new features, most notably storing patch metadata and patch reactivation and reversion mechanisms. We analyzed and described their properties and suggested potential future improvements. The shortcomings of the tool were described in chapter *Known problems.* The extension currently only supports x86-64 architecture and Linux operating system.

Bibliography

[1] Ryan A. Chapman. Linux system call table. Available at `https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/` (visited on May 20, 2023).

[2] Intel Corporation. Pin - a dynamic binary instrumentation tool. Available at `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html` (visited on May 20, 2023).

[3] Intel Corporation. Pin user guide. Available at `https://software.intel.com/sites/landingpage/pintool/docs/98718/Pin/doc/html/index.html` (visited on May 20, 2023).

[4] Ulrich Drepper. How to write shared libraries. Available at `https://www.akkadia.org/drepper/dsohowto.pdf` (visited on May 16, 2023).

[5] OpenBSD Foundation. Openssh. `https://www.openssh.com/` (visited on May 20, 2023).

[6] Inc. Free Software Foundation. Gdb python api. Available at `https://sourceware.org/gdb/onlinedocs/gdb/CLI-Commands-In-Python.html#CLI-Commands-In-Python` (visited at May 20, 2023).

[7] Michael Kerrisk. ptrace(2) — linux manual page. Available at `https://man7.org/linux/man-pages/man2/ptrace.2.html` (visited on May 20, 2023).

[8] GNU project. Gcc - common variable attributes. Available at `https://gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html` (visited on May 20, 2023).

[9] SELinux Project. Selinux. Available at `https://selinuxproject.org/page/BasicConcepts` (visited on May 20, 2023).

[10] Dan Sporici. Hot patching c/c++ functions with intel pin. Available at `https://codingvision.net/hot-patching-functions-with-intel-pin` (visited on May 16, 2023).

[11] SUSE. Libpulp - user space live patching. Available at `https://github.com/SUSE/libpulp` (visited on May 20, 2023).

[12] Samuel Čavoj. Modification of a process code at runtime. 2022. Source code at `https://github.com/sammko/bakatsugi`.

Appendix A: the source code

The source code of the GDB extension described in the thesis is attached electronically as the archive of the git repository. The repository contains a readme file `README.md` which is used as a project description and the user manual. Its content is copied into *Appendix B: the user manual.* Directory `src` includes the extension script `commands.py` and the C header file `patch.h` for patch libraries. Directory `examples` contains various examples of programs and patch libraries. Moreover, a `Makefile` is provided for each of them to simplify the building process. The root directory contains a bash wrapper `patch.sh` which can be used to simplify the patching procedure. The wrapper is described in more detail in *Appendix B: the user manual.*

The git repository is available at
`https://github.com/filipkosecek/gdb-livepatch`.

Appendix B: the user manual

Linux process live patching using GDB

The project focuses on Linux process live patching using GNU debugger tool targeting x86-64 architecture. It can be used to patch whole functions of the target process as well tracking and logging changes made. This extension is also able to revert these changes, i.e. restoring the original functions.

The extension has its limitations, therefore it is not recommended to use it for critical software.

Repository structure

Directory `src` contains two main components:

- `commands.py` - the python script defining new GDB commands which perform the patching
- `patch.h` - C header file containing macros and metadata structures definitions which every patch library must use

Directory `examples` contains various examples of target processes as well as patch libraries each in a separate subdirectory. Every example contains a `Makefile` which builds the target process and libraries from the source code. The repository contains a bash wrapper to simplify the patching process.

Installation

GDB

You must have at least GDB 7 installed in your system. GDB is often installed by default in many Linux distributions. Even if it is not, you can always install it using a package manager:

```
(Debian) apt install gdb
(Redhat) dnf install gdb
```

Python

The extension is a python script which can be imported into GDB. GDB contains embedded python interpreter. The extension is tested using GDB with python3 interpreter. There are GDB versions that use Python 2 which is not supported by the extension. To verify what version of python interpreter you are running, you can visit this page to learn how to do it directly in python interpreter. Any version of Python 3 should be sufficient. In a GDB session, Python interpreter can be invoked using `python-interactive`.

There are two ways to import the extension script. Firstly, you can launch GDB with the path (either relative or absolute) to the script as a parameter.

```
gdb -p $pid --command=porject_path/src/commands.py
```

Alternatively, you can import the script directly from GDB shell using source command:

```
source src/commands.py
```

To automate this task, you can place the script at a fixed path and insert the above command in your initialization GDB script `.gdbinit` in your home directory. This will ensure the script will get loaded every time GDB is started.

Glibc debugging symbols

To make the extension usable, you must install debugging symbols for `glibc`. These symbols are available in all Linux distributions. They can be installed via package managers on most Linux distributions.

```
(Debian) apt install libc6-dbg
(Redhat) dnf debuginfo-install glibc
```

Alternatively, debuginfod service can be used on newer distributions to download the debugging information on the fly from a debuginfod server during a GDB session. Debuginfod can be enabled with the following command in GDB:

```
set debuginfod enabled on
```

This command can be added into `.gdbinit` script as well.

Visit this page for more details about debuginfod service.

User manual

Writing a patch library

Patches are represented as shared libraries with declared functions which are to replace the old ones. The libraries must contain special instructions and metadata declared in C header file. In `examples` directory, you can see various examples which use the defined instructions and metadata definitions and contain `Makefile` files for building the patches. Using this building process is strongly advised so we recommend to make a copy of such a `Makefile` when writing a patch. The most important requirements for the building process are:

- patch libraries must contain debugging symbols
- patch libraries must be compiled as shared libraries

The extension cannot handle symbol name collisions correctly. In other words, if you declare a function in your patch library and its name is already used in the target process, the patching will likely not work correctly and might crash the process. Symbol names can be shared across the patch objects as the search scope is limited to the patch object. For this reason, it is recommended to name replacement functions after the functions they are going to replace with slight modifications, e.g. adding `new_` prefix to the name of the original function.

Attaching to a process

To attach to a running process, you have to start with `gdb -p $process_pid`. Alternatively, you can start GDB with `gdb` and run `attach $process_pid` in GDB shell.

Executing a patch

To carry out a patch, you have to be attached to the target process and run command

```
patch /path/to/your/patchlib
```

We recommend to use the absolute path if the patch library is not in your standard search path. This will invoke the patching procedure with logging and tracking turned off. If you want to turn on logging, you have to specify `--log` switch as the second parameter, i.e.

```
patch /path/to/your/patchlib --log
```

Error handling in the extension is not perfect so you might encounter errors during the patching. The most common are the following:

- relative path to the patch library
- the target process missing both the symbol table and debugging symbols
- missing `glibc` debugging symbols
- missing patch instruction definition, e.g. no definition of the function to be replaced and the replacement function
- C macro file (`src/patch.h`) not included in your patch library
- macros for patch instructions not used in your patch library
- typographical errors in original or replacement function names

Printing patch history

When logging was turned on for a patch instance, an entry for it was created and written to the log. You can print the log with

```
patch-log
```

taking no parameters. It contains patch information, most notably the original and the replacement function and information whether the patch is active or not. The output of the command should look something like this:

```
[0] revert
[1] 2023-05-11 20:00:00: old_function -> my/patch/lib:new_function
[2]* 2023-05-11 20:04:00 old_function -> my/patch/lib2:new_function
```

If the log cannot be found, an error message is printed stating that the log couldn't be found.

Dumping patch history

It is possible to dump the patch log into a specified file. This might be useful when you are about to revert some patches and you would like to preserve all changes made. The usage is as follows:

```
patch-dump /path/to/dest/file
```

Reapplying patches

It is possible to reapply a patch whose entry is marked in the log. The `patch-log` outputs a sorted list of marked entries. You can reapply any patch from the log with `patch-reapply` command which takes the index of the patch entry in the log. This patch is then reapplied in case the library containing the replacement function has not already been closed. A special parameter is 0 which represents patch reversion.

Reverting patches

You can revert an active patch with `patch-reapply` command with parameter 0. If no additional parameter is specified, all active patches are reverted. Otherwise, a list of original functions to be restored is specified. The list can contain either a hex address of the function or its name. A patch could be reverted like this:

```
patch-reapply 0 old_function
```

The bash wrapper

A bash script is provided which simplifies the patching procedure. GDB does not need to be invoked manually. The script takes PID of the target process as the first argument followed by the command to be executed in GDB. For example, to apply a patch with logging turned on, one could use the following command:

```
./patch.sh $pid patch examples/more/patch.so --log
```