Studiengang, Kurs: TINF22, IT1
Vorlesung: Moderne Konzepte der Informatik
Betreuer: Bauer, Johannes, Prof. Dr.-Ing.
Abgabedatum: 15.05.2025

Development of a minimal x86-64 Unix debugger in Rust

Florian Hartung (6622800), Marek Freunscht (9604914)

I. Introduction

Debuggers are programs that provide tools for developers to run, monitor and modify the state of other processes. Historically, they have been used primarily to find and eliminate bugs, which are inherent in almost all software developed. Another prominent use case for debuggers is foreign code analysis, sometimes needed during reverse engineering or pentesting. Due to the vast amount of different technologies and languages existing, debuggers must adapt, which is why they come in many shapes and forms: Some debuggers are architecture-specific or support only only a specific set of architectures such as ARM, x86, x86-64 or PowerPC. Other debuggers may require the use of specific programming languages, such as languages that compile to machine code, Java or Python just to name a few.

In this work, we present the bare minimum required to implement a debugger for low-level programs that compile to machine code. We limit our debugger to Unix and Unix-like systems out of personal preference and the x86-64 architecture because of its widespread use. Also we choose Rust as the programming language for our debugger, due to its advantages regarding memory safety and modern approach to software. The debugger is required to implement a certain amount of base features we identify as necessary for a minimum viable product.

First, we present important fundamentals such as the lifecycle of a process in Unix/Unix-like systems. The next section presents our debugger with its initial requirements, development process and application in a test scenario.

II. Fundamentals

This section describes fundamentals for processes on Unix/Unix-like systems and debuggers useful for later development.

A. *Process lifecycle in Unix/Unix-like systems*

Processes are programs during runtime. They contain information such as the program counter, registers, variables, open files or page metadata[1].

In Unix and Unix-like systems, there exist multiple methods for creating new processes, however the most common one is the `fork` syscall[2]. A process may use this syscall to create an identical clone of themselves. Followed by diverging control flow inside this process' program code, the newly created child process can then behave differently from the original parent process. Often the child process then calls a syscall of the `exec` family (or the underlying `execve` directly), which replaces its current memory layout with a completely new one from a specified program.

During this `exec` syscall, the Address Space Layout Randomization (ASLR) technique is initialized, provided that the executed program is a Position Independent Executable (PIE). ASLR randomizes the starting address of memory section such as the text, heap, or stack sections to prevent certain attacks, which exploit the normally fixed memory layout. The mappings of memory sections are stored by the kernel and made available at `/proc/<PROC_ID>/maps` for every process by its process id `<PROC_ID>`.

B. *Signals*

Signals are a feature offered by kernels that comply[1] with the Portable Operating System Interface (POSIX) Application Programming In-

[1]Compliance is meant as partial compliance with the specific signal feature as few kernels are actually fully POSIX-compliant

terface (API) to asynchronously send messages and trigger events between processes. Commonly known signals include SIGTERM, which terminates a process, SIGSTOP, which stops a process, SIGCONT, which continues a stopped process or SIGTRAP, which signals a breakpoint [3].

Most signals can be caught and handled by the receiving process with a signal handler. This signal handler, which can be user-defined, defines a routine which processes the signal and handles it accordingly. If a user-defined signal handler is not present for a specific signal, the default signal handler is invoked. The only exceptions for this are SIGKILL and SIGSTOP, which cannot be caught and respectively kill and stop a process [3].

One use case for signals in a debugger is controlling the execution of the traced process with signals like SIGSTOP, SIGCONT or SIGKILL. Additionally, the signals which target the traced process will be delivered to the debugger first. The debugger can then process these signals and decide which to deliver to the traced process. This includes SIGTRAP, a signal which is sent when a breakpoint is hit in the executable [3].

C. *Debuggers*

Debuggers are programs that can attach themselves to other processes and then monitor and modify them. They are mainly being used by developers for debugging programs, i.e. identifying and tracing bugs/errors in these programs. Although there are other use cases such as reverse engineering.

Widely-known debuggers, namely `gdb` or `lldb`, are terminal Read-eval-print loops (REPLs) with a notable overlap of common commands between them. Commands such as `run` or `continue` are used to create and attach to a new process or resume execution of a process the debugger is already attached to. Breakpoints can be set via commands such as `break` or `breakpoint set` for arbitrary code addresses or addresses of symbols. When the program counter of the process then reaches a previously set breakpoint, execution is preempted and control transferred to the debugger allowing further user input to happen. Watchpoints are similar to breakpoints allowing execution preemption, however they trigger on memory reads and writes of set addresses instead of execution. While execution of a process is halted, the process state including but not limited to call frames or variable contents can be read and modified.

Operating Systems (OSs) usually isolate processes, their resources, memory, etc. from each other. However debuggers require access to other processes to be able to debug them. Thus debuggers require OSs to provide some interface through which they can access and debug these other processes. For Unix and Unix-like systems this is the `ptrace` syscall, which is short for `process trace`. Even though it is a single syscall, `ptrace` combines various different commands useful for debuggers to trace other processes.

D. *Symbols*

When source code is compiled by a compiler to native machine code, a lot of information about the original source code is changed or completely lost. Such information may include the names of variables and functions or the layout of stack frames. Debugging programs missing this information is time-consuming and most of the time not feasible in practice. To solve this problem, object/executable formats include sections where compilers can store additional debug information. Debuggers can read this debug information and use to give users meaningful insight into the debugged process.

One crucial category of information compilers produce are symbols. Symbols relate string names to addresses in the final object/executable file. For Executable and Linkable Format (ELF) files, symbols reside in a symbol table and their string names in an additional string table. ELF symbols can be of different kinds, such as functions (`STT_FUNC`), sections (`STT_SECTION`), globals (`STT_GLOBAL`), etc. [4].

III. Requirements

As the scope of this debugger implementation is fairly limited, basic requirements for the debugger are defined:

The set of debuggable programs is restricted to x86-64 ELF binaries for Unix and Unix-like systems. Furthermore, the debugger shall provide the following basic functionalities to run and observe other processes: The debugger must be able to attach to running processes and run new processes. For inspecting binaries, the debugger must allow the user to list all function symbols contained inside of a given binary. Setting breakpoints at arbitrary

```c
#include <stdio.h>
int fn_a() { printf("A\n"); }
int fn_b() { printf("B\n"); }
int fn_c() { printf("C\n"); }
int main() {
    fn_a();
    fn_b();
    fn_c();
    fn_c();
}
```

(a) Source code

```
$ cargo run -- ./<EXECUTABLE>
▯break fn_b
▯continue
A
Hit breakpoint at address 94389373546851
▯continue
B
C
C
Process exited with code 0. Quitting...
```

(b) CLI usage

Figure 1: Exemplary source code and CLI usage for setting a breakpoint and continuing execution

addresses or function symbols must also be allowed. Watchpoints that trigger on reads or writes at arbitrary addresses are also required.

IV. Design

We split the debugger project into two parts for better modularity: a core debugger and a Command-line Interface (CLI). Both are implemented as separate Rust crates with the CLI crate depending on the core debugger crate.

The core crate contains the main logic for the debugger and exposes safe Rust interfaces. Its central part is a `Debugger` Rust struct. It represents a currently debugged process and has the same lifetime as this process. The `Debugger` struct is created when attaching to a process. A created instance of this struct then exposes several methods to interact with the debugged process, for example setting breakpoints or controlling execution.

The CLI crate provides the user interface in form of a REPL while ensuring a consistent output format for messages and error reporting. It also includes logic for displaying the REPL and parsing user input.

V. Implementation

This section explores the implementation details of the various methods needed to fulfil our requirements.

A. *Attaching to processes*

When beginning to debug a process, there are typically two scenarios for a debugger: attaching to a process that is already running and creating a new process. Both of these have available ptrace APIs to use with the debugger.

Attaching to a running process can be done with either PTRACE_ATTACH or PTRACE_SEIZE. When using PTRACE_ATTACH, the attached process is signaled to stop immediately and the debugger should wait until that stop is completed using the `waitpid` syscall. The user can then set breakpoints or obtain information about the process while it is stopped. On the other hand, PTRACE_SEIZE does not stop the attached process and gives the debugger a little more flexibility to do so later with PTRACE_INTERRUPT. PTRACE_SEIZE also allows the debugger to use some other functionality, like PTRACE_LISTEN. However, in our implementation we use PTRACE_ATTACH because it is sufficient for our use case and the flexibility and complexity of PTRACE_SEIZE is not needed [5].

Another use case is the user wanting to debug an executable that is not already running in some process. In this case, the debugger can start the executable and initiate the tracing with ptrace. This is typically done by forking the debugger process, which leaves the programmer in control of what happens in the child process. After forking the parent, the child process initiates the tracing using PTRACE_TRACEME, which turns the parent process into the tracer [5]. Finally, the child process can be turned into the desired executable by executing an `execl` syscall. After this `execl` call executes successfully, a SIGTRAP will be sent to the tracee which stops it and leaves the debugger in control [6].

Both of these cases are supported in our implementation because they are fundamental for a functional debugger.

B. *Setting breakpoints*

There are two main methods to set a breakpoint inside a process that is currently running, software breakpoints and hardware breakpoints.

Software breakpoints use the `int3` instruction of the x86-64 instruction set, which triggers an exception inside the processor [7]. When this exception

```c
#include <stdio.h>
int a = 5;
int before_write() {
  printf("before write: %d\n", a);
}
int after_write() {
  printf("after write: %d\n", a);
}
int main() {
  before_write();
  a = 15;
  after_write();
}
```

(a) Source code

```
$ cargo run -- ./<EXECUTABLE>
□watch 0x404030 write 1
□c
before write: 5
Hit watchpoint Data { condition: Write, length:
OneByte } at address 0x000000404030
□c
after write: 15
Process exited with code 0. Quitting...
```

(b) CLI usage

Figure 2: Exemplary source code and CLI usage for setting a watchpoint that triggers on writes and execution

is encountered, a trap occurs, transferring control to the operating system, which in the case of Unix-like systems will send a SIGTRAP to the running process. To set a software breakpoint in the tracee process, the debugger may write the `int3` instruction directly into the executable text segment of the tracee with the PTRACE_POKETEXT API [5]. The first byte of the instruction at the breakpoint address can be overwritten with `int3`, which has a one-byte opcode (0xCC). When this breakpoint is hit during execution, the debugger has to write back the first byte of the instruction which was in the breakpoint address initially. Execution of the process is then resumed for one instruction, after which the `int3` instruction is re-inserted into the breakpoint, so that it can be hit again.

In contrast, hardware breakpoints use the hardware debug registers that are a part of the x86-64 architecture. These registers make it possible to specify breakpoints at 4 different addresses inside the process. However, the breakpoints that are stored in hardware registers are more powerful than software breakpoints, as they can also be triggered on memory reads and writes, as opposed to just execution. This may be specified in the debug control register (DR7) for each address individually, which is another debug register. While the direct access of these registers via the `mov` instruction requires a privileged process, they can also be accessed with the PTRACE_WRITE_USER API [5], that allows the tracer to write fields of the `user` struct[2] in the tracee process, which we use in our implementation [7].

[2]see glibc source: https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/unix/sysv/linux/x86/sys/user.h

Our debugger implements both types of breakpoints, as they have unique strengths and weaknesses. Software breakpoints are used as the primary execution breakpoint mechanism, because there is no limit to the number of software breakpoints. Hardware breakpoints, on the other hand, are more flexible in the functionality that they provide, as they can be used to monitor memory access rather than just execution. However, the number of hardware breakpoints is limited to 4 by the processor architecture.

C. *Instruction Stepping*

Instruction stepping is vital for the user to have fine grained control over the program execution after hitting a breakpoint. This allows the user to advance the execution by a single instruction at a time and inspect the program state after each step.

To implement this, ptrace provides the PTRACE_SINGLESTEP API [5]. The tracee will be stopped after executing one instruction. This is done internally by the kernel, which sets the trap flag in the x86-64 FLAGS status register. The CPU will generate a trap after execution which yields control back to the debugger [7]. In order to ensure that the debugger does not initiate invalid ptrace calls while the tracee is still running, a call to `waitpid` is necessary to wait for the tracee to stop.

VI. Debugger Usage

Our debugger CLI can be executed through cargo, Rust's official package manager. Figure 1 and Figure 2 show the source code of C programs and the REPL interaction for setting break- and watchpoints and continuing execution.

In the following other commands or syntaxes not shown here are presented: the `info functions` command is implemented to list all function symbols of a binary. The `step <STEPS>` command is used to advance execution by a number `STEPS` of instructions. The suffix `hard` can be used with the `break` command for setting hardware instead of software breakpoints, e.g. `break 0x1234 hard`. Furthermore, the `watch` command already presented in Figure 2 allows to configure it to trigger on writes (`write`) or reads and writes (`read_write`). Also the length of the watchpoint may be specified as one of 1, 2, 4 or 8 bytes.

VII. Outlook

Looking forward, there are multiple features that could be additionally implemented to improve the debugging experience for the user. One feature that most debuggers provide is disassembly of machine code at the current or any other arbitrary address. This in an integral process during debugging in general, because it allows users to inspect machine code while it is being executed.

Also users should be able to read and write register contents and memory at arbitrary locations.

Another feature is stepping into and over functions, allowing more fine-grained execution control. While our debugger currently provides the functionality to step a single instruction, this mechanism is not aware of function boundaries, which is required to step through individual source code statements.

Furthermore, stack unwinding is the process of iterating through call frames to generate a snapshot of the current backtrace, providing insight to the user about the current function call hierarchy. This was also not implemented for our debugger due to the high complexity needed to parse stack frame information and unwind the stack based on that.

VIII. Conclusion

In this work, design techniques for debuggers were explored. Then, a debugger was implemented, providing basic functionality including breakpoints, watchpoints, listing function symbols and instruction stepping. Two examples demonstrate the the debugger's usage. While these show that the debugger works for simple examples, various functionalities are still missing for real world usage.

Bibliography

[1] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Pearson, 2014.

[2] "fork(2) - Linux man page." Accessed: May 12, 2025. [Online]. Available: https://www.man7.org/linux/man-pages/man2/fork.2.html

[3] "signal(7) - Linux man page." Accessed: May 10, 2025. [Online]. Available: https://www.man7.org/linux/man-pages/man7/signal.7.html

[4] "elf(5) - Linux man page." Accessed: May 12, 2025. [Online]. Available: https://www.man7.org/linux/man-pages/man5/elf.5.html

[5] "ptrace(2) - Linux man page." Accessed: Apr. 28, 2025. [Online]. Available: https://www.man7.org/linux/man-pages/man2/ptrace.2.html

[6] "exec(3) - Linux man page." Accessed: May 10, 2025. [Online]. Available: https://www.man7.org/linux/man-pages/man3/exec.3.html

[7] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 1–4.