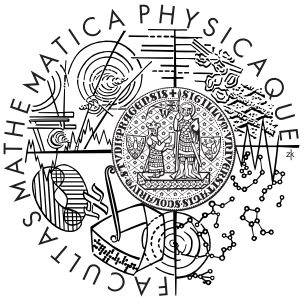
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Michal Demin

Undetectable Debugger

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký
Study programme: Informatics
Specialization: Software systems

Prague 2012

I would like to thank my supervisor, Mgr. Martin Děcký, and all people that supported me in writing of this thesis.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. I understand that my work relates

to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, date Michal Demín

Název práce: Undetectable Debugger
Autor: Michal Demin
Katedra (ústav): Katedra distribuovaných a spolehlivých systémů
Vedoucí diplomové práce: Mgr. Martin Děcký
E-mail vedoucího: decky@d3s.mff.cuni.cz

Abstrakt: Používanie debuggeru je bežný prostriedok na identifikáciu a analýzu malwaru (ako sú vírusy, červy, spyware, rootkity, apod). Keďže, debugger môže byť detekovaný malwarom prostredníctvom pozorovania správania operačného systému, zmien v kóde (napr. breakpoint inštrukcií) a neštandardného správania procesora, urobiť analýzu malwaru môže byť tažké a pomalé. V tejto práci implementujeme základný debugger založený na QEMU emulátore, ktorý dokáže ukryť svoju prítomnosť pred ladenou aplikáciou. Dosahujeme to pomocou QEMU ako virtuálneho stroja a pridaním rozlišovania kontextov do už existujúceho primitívneho debuggeru. Rozlišovanie kontextov je realizované pomocou vstavaného skriptovacieho jazyka Python. Takýto systém nám umožňuje flexibilne implementovať podporu pre rôzne operačné systémy. V tejto práci sme vyvinuli dva príklady. Prvý príklad je pre operačný systém RTEMS, ktorý slúži ako ľahko pochopiteľná referenčná implementácia. Druhý príklad je vyvinutý pre operačný systém Linux, na ukázanie schopností nášho nedetekovateľného debuggeru v reálnejšom príklade.

Klíčová slova: virtualization, debugging, malware

Title: Undetectable Debugger
Author: Michal Demín
Department: Department of Distributed and Dependable Systems
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: decky@d3s.mff.cuni.cz

Abstract: Using debuggers is a common mean for identifying and analyzing malware (such as viruses, worms, spyware, rootkits, etc.). However, debuggers can be detected by malware via observing of the behavior of operating system, changes in code (such as breakpoint instructions) and non-standard behavior of the CPU, making the analysis of the malware can be hard and tedious. In this thesis we are implementing a basic debugger based on the QEMU emulator that hides its presence from the debugged application. This is accomplished by using the QEMU as virtual machine and adding context awareness to the already existing primitive debugger. The context awareness is implemented using an embedded Python scripting engine. Such setup gives us a flexible way of implementing support for various operating systems. In this thesis, we have developed two examples. One example is for the RTEMS operating system, which serves as easy to understand reference implementation. Second example is for the Linux operating system, to show the abilities of the undetectable debugger in a more real scenario.

Keywords: virtualization, debugging, malware

Contents

Chapter 1

Introduction

Today, the main threat to computers is malware. This kind of software can cause data corruption, spying, or play other role in some bigger attack. Debugger is one of the tools that is used in analysis of such software. As with any other software, malware is getting more sophisticated each day - hiding its presence before debuggers, changing its behavior or destroying itself when abnormal activity of user is detected.

In this thesis we are going to discuss usage of modern virtualizing technologies to hide presence of debugger in system - creating natural environment for malware to live in with comfortable debugging support for software analyst. To achieve this we are going to base our work on already existing project - QEMU. QEMU is an advanced virtual machine emulator/virtualizer that can run in various configurations (more in section 2.1).

Main target of this thesis will be Intel's IA-32 (or x86) architecture. We are going to develop on Linux system using standard development tools (GCC, GNU make, etc).

The debugger developed will have only basic debugging capabilities. By basic debugger we understand debugger, that can access (read or write) memory, the state of the CPU and peripherals. Our debugger will also implement support for operating system. The operating system support will include support for listing threads and limiting of the debugger to a single process.

We will prepare support for two different operating systems. This proof-of-concept support will serve as demonstration of the capabilities of such debugger. The first operating system will be RTEMS, which is a full featured operating system with easy to understand source code. Second operating system will be Linux, which is one of the most widespread free operating system with much more complex internal structure.

This thesis is divided into 3 chapters. Each chapter describes different part of development process. In chapter 2 we are going to introduce the existing software, that this work is based upon. In chapter 3 we are going to analyze the problem of undetectable debugger, analyze different problems that need to be solved. In chapter 4 we are going to discuss our implementation, interface of our implementation and discuss two implemented examples of our undetectable debugger.

Chapter 2

Prerequisits

In this chapter we are going to introduce the software, that this project is based upon. This software includes *QEMU* emulator and *RTEMS* and *Linux* operating system.

2.1 QEMU

QEMU is a processor emulator, that supports many different processor architectures. QEMU can act as an emulator and as a virtualizer. When running as emulator, QEMU is emulating the given hardware and can run code build for one architecture on different machine architecture. This is accomplished by dynamic translation of code.

When running as virtualizer, QEMU is using virtualization support of the host system to achieve almost native execution speed. The support is provided either by Xen hypervisor, or by using the Linux KVM kernel module.

QEMU supports many different architecture emulators: IA-32, x86-64, MIPS, Sparc, ARM, PowerPC, etc. Most interesting for us are IA-32 and x86-64 architectures.

QEMU acts not only as a processor emulator. QEMU works as complete systems emulator with various peripherals like video cards, sound cards, network cards, hard-drive, CD-ROM drive, many more.

Modes of operation

QEMU has two different modes of operation:

- User mode emulation
- Full system emulation

In User mode emulation, QEMU can launch programs compiled for one CPU on different CPU. This is usually useful for cross-compiling projects. This mode is not particularly interesting as the guest application is running in native environment.

In full system emulation, QEMU emulates full system (eg. PC-style computer) with possible multi-CPU environment and various peripherals. This is the target

mode for our undetectable debugger as the guest system is running in its own controlled environment. QEMU is able to emulate full PC style computer with all necessary peripherals.

Debug support

QEMU has embedded GDB support. This support is very basic and supports accessing some basic information of the emulated hardware. The support is implemented according to the GDB remote protocol (see [5]). This protocol is supported by GDB which can be used as remote debugger.

The debug interface provides extensive configuration interface for QEMU. This interface is accessible though monitor commands of the GDB protocol. These commands are passed in raw form to the QEMU and QEMU handles the configuration menu for various modules.

By using any GDB client, user can connect to a running instance of QEMU and inspect the state of the machine. In full system emulation user can inspect registers/state of each CPU and the memory space of the guest operating system. Each CPU is presented as single thread - user will see as many threads in GDB client as there are active CPUs in the emulator.

This debug support is provided by "GDB stub" module in QEMU. This stub implements only few basic commands such as reading and writing of registers/memory, handling breakpoints, sending events (triggered breakpoints) back to the connected client. The GDB stub does not do any analysis of the running guest operating system.

KVM

KVM (see [1]) is a virtualization solution for Linux operating systems on x86 based hardware. It provides support for virtualization extensions included in modern CPUs, such as AMD-V or Intel VT. The KVM consists of kernel loadable modules with support for specific hardware implementation, as well as QEMU that is built with KVM support.

KVM provides support to execute virtual machine with native speed. The virtual machine is executed on host CPU in a fully isolated environment.

2.2 RTEMS

RTEMS (Real-Time Executive for Multiprocessor Systems - [9]) is a full featured operating system. RTEMS is a free open-source software distributed under modified GPL license. This system supports following APIs: Classic API (RTEMS API), POSIX, uITRON.

This system has been chosen as an example for our undetectable debugger project. This system provides support for many different hardware configurations from embedded, standard PC to an emulated environment. The fact, that full source code is provided, will help us better understand the inner workings and will provide deterministic environment for our debugger testing.

As there are many configurations possible, we are going to utilize the following configuration:

- RTEMS Version 4.10
- default configuration for PC style computer, x86 architecture
- no interrupt context switch

2.3 Linux

Linux is a Unix-like operating system developed under free and open source model. This operating system has been initially developed by Linus Torvalds in 1990s and since that has grown to be one of the most widespread open-source operating system. This operating system runs on many devices ranging from small embedded devices (such as cell phones, routers) through desktop computers to high-end super computers.

The operating system consists of a Linux kernel ([11]) and user-space applications. The Linux kernel is a monolithic modular kernel, which handles processes, networking subsystem, sound subsystem and others. Device drivers are either integrated into the kernel or loaded during runtime. User-space depends highly on used distribution of Linux operating system. Each distribution have its own choice of the applications.

The kernel itself does not care what applications are executed in the user-space. With this fact in mind, we are going to use lightweight user-space environment and focus our work on the kernel internals. We are going to limit our kernel configuration to following:

- kernel version *3.3.8*
- x86 architecture.
- no SMP support

Chapter 3

Analysis

In this chapter we are going to describe some inner working of QEMU emulator that is relevant for this project, and we are going to discuss various techniques that are used in our basic debugging. These techniques and features are later going to be implemented into already existing primitive debugger that exists in QEMU emulator.

3.1 Adding context awareness

By context awareness we mean ability of a debugger to distinguish between different contexts the system is running in. Context can be an operating system's kernel function or a user application thread, driver's thread, etc. To make the debugging as user-friendly as possible, we will need some mechanism to detect whether we are in a part that user desires to debug (we call this "target thread" or simply "target") or in some irrelevant part (syscall, interrupt handler, other application's thread, etc), not interesting to the user.

At the beginning of the debug session user will need to specify his target, and all debugging will continue as when debugging regular program in user-space. To do this we need to make sure, that our debugger is aware of the content being debugged and understands the guest operating systems's state. Guest operating system is the operating system running in the emulated/virtualized environment.

As the guest OS is running in an emulated environment, the GDB server, having raw access to the CPU, has access to all the information about the system. We are going to exploit this fact, and add GDB server the ability to identify and understand various structures of the guest operating system.

Because the GDB server has direct access to the QEMU's CPUs and we want to make the debugging experience as user-friendly as possible, the context awareness features are going to be closely-coupled with the GDB server. This way user can use any GDB client to access the debugger. Also by making the context awareness closely coupled with the GDB server we are minimizing the latency of the analysis of the guest system. If the guest system were to be analyzed over some TCP connection, the network latency could slow down the guest system analysis.

3.1.1 Deciding on the form

Before we begin any programming we need to decide what form the debugger will have. By form we mean either hard-coded features programmed directly into the GDB server or have some scripting engine built into the GDB server.

Hard-coding the debugger's features into the QEMU code makes the debugger less flexible. In this type of implementation user will need to recompile the whole project every time a small change is added or when changing the behavior of the debugger. Hard-coding the debugger would, on the other hand, make the debugger much faster. As we will later see, it is much better to sacrifice speed for the sake of flexibility, as the number of possible configurations of some operating systems is almost unlimited.

Having the operating system specific parts written in scripting language will require to link the existing QEMU project with some, possibly large, scripting engine. This will add another compile-time dependencies to the QEMU project. This approach will require that only small portion of the GDB server is modified. This approach will require to expose some functions of QEMU to the context awareness script. The scripting approach is much more flexible in terms of adding/removing features. User can load and unload scripts at runtime, without affecting the running programs or the guest operating system. Changing to different set of scripts should be not pose any problem.

3.2 Symbol tables

Symbol table is a table that stores location of certain objects in memory. These objects can be functions, variables, markers added by linker, etc.

After building and linking of some project, compiler will embed symbol table into the resulting file. Such file can be in ELF, COFF or other format. Binary files do not embed such information at all. This symbol table contains symbol names, symbol types and locations in the file. Some symbols can have no location at all, and are expected to be dynamically loaded (either at startup or during runtime). These tables can optionally be removed by process called stripping. Running operating system, and release versions of software in common, do not usually contain such symbols, but these symbol tables can be found in development packages (SDKs).

There are few possibilities on how to deliver these tables to the script:

- hardwire values into the source
- create some local symbol storage
- use GDB client query command

Hardwiring the symbol locations is not flexible option. We do not want our context awareness scripts to have hardwired addressed. Different build of the same system can have symbols placed at different locations. This would break compatibility. We want to use symbolic names inside these scripts.

Creating local storage of symbols is possible. QEMU however does not provide any facility for storing symbols and retrieving them from executable files. This would

require to replicate much of the work, that has already been done in many debuggers such as GDB.

Last method is loading symbols from GDB client. GDB client is using BFD library to parse many types of executable files and files that contain symbols. GDB client/server protocol includes facility for symbol retrieval. GDB client asks GDB server (on connect/on symbol table load) whether it wants to resolve any symbols. The GDB client then polls server until all unknown symbols are resolved.

In this method the script will need to keep table of symbols that need to be resolved. This table will need to be filled at the very beginning before the GDB client connects to the server. The context awareness in GDB server is not necessary to be active until user wants to start debugging, so there is no problem with symbol locations being unavailable before the GDB client connects.

3.3 Call-Stack

Call-Stack is data structure used to store local variables, return addresses, function state (CPU registers), arguments passed to functions. What data is stored on the stack, and the exact way of storing them depends on compiler and/or ABI (Applications binary interface).

For x86 architecture the stack is usually placed at the "end" of the address space and filled to the lower addresses. There are two main operations: push and pop. Push operation places values on stack, pop operation removes values from stack.

The call-stack is build when calling functions (also called winding) and "destroyed" when leaving functions (unwinding). Part of stack, that belongs to one function is called *frame*. Each function contains prologue and epilogue part. These two parts are responsible for building and destroying the frame of the function.

When calling a function, prologue of called function allocates frame space on the stack. The size of frame depends on what registers are saved, how many and what type of local variables are used. This value is usually known during compile time. However, for some higher level languages, that support dynamically sized local variables, the exact frame size is not known.

Epilogue on the other destroys the frame, and leaves some values on the stack, when necessary (return value). The frame handling is hidden from programmer in most programming languages.

The size of frame of each function can also be obtained from the debug information such as Dwarf([3]). Most debuggers can extract this information and use it to analyze the function stack.

3.3.1 Stack unwinding

Call-stack provides lots of useful information on currently executing application. From the call-stack we can determine the current back-trace of functions and arguments passed to each one (if arguments are passed by stack). The technique of analyzing existing call-stack is called unwinding.

Unwinding the call-stack is highly dependent on the compiler options used when creating the application. As our main interest is x86 architecture, we are going to limit our analysis on this architecture.

The frame pointer always points to the beginning of the stack frame of the current function. On x86 architecture frame pointer is stored in the EBP register (base pointer register). The frame pointer is managed by the prologue and epilogue. When prologue sets up a new frame, it makes sure that the frame pointer points to this frame. And when epilogue destroys the frame of the current function, it makes sure, that the frame pointer points to the frame of the caller function.

Each frame on the stack contains return address. Address where the execution jumps, when the current function exits. We will threat the stack as linked-list, with frame pointer as the pointer to the next node and return address being the only member of the node. This means that by following the frame pointers we can effectively traverse the frames on the stack and by collecting the return addressed we will create back-trace.

The current frame pointer is obtained from frame pointer register (EBP register on x86 architecture). The only problem with this technique is that back-trace will not give us addresses of the beginning of the functions. But using the right tools these return addressed can be resolved into function names and/or lines of code (if source-code is available).

More difficult situation happens when application is compiled using "-fomit-frame-pointer" option (GCC option, or similar on different compilers). This option will remove use of frame-pointers from the stack management. This is particularly useful when there are many single-line functions calling some other functions.

In standard behavior the code would have to setup stack frame, call the second function and destroy the frame when returning from the function. This could lead to some performance issues and wasting of stack space. With omitted frame-pointers the body of the function would translate to single "jump" instruction. Such function would not be visible in back-trace at all.

Technique used to support such stack unwinding depends on the architecture. For example on MIPS architecture one could scan to the beginning of the currently executing function for any instruction that modifies the stack-pointer register. This can be done due to the fact, that MIPS has constant size instructions and dedicated register for return address. After finding the correct instructions, algorithm would extract the change done to the stack-pointer and calculate the stack pointer of the caller function. Algorithm would then jump to the previous function and repeat scanning of instructions. It would continue until some invalid return address is encountered. In each iteration it would save the return address.

The described algorithm depend on constant size instructions. As x86 architecture has variable instruction size the mentioned algorithm is not usable on this architecture. It is not possible to scan backwards and look for instructions that change the stack-pointer.

If this is the case, we need to look into debug information that is available with the executable file (such as Dwarf format [3]). The debug information is optionally added by the compiler to ease debugging. But it is rarely kept in production releases

of the software, as it increases code size dramatically.

In some cases, like stack corruption or application with nonstandard use of stack, these techniques are useless and some more advanced analysis needs to be done - disassembling the code.

3.4 QEMU

To be able to implement this context awareness we need to understand the architecture of QEMU. We will be looking at QEMU from the GDB server's point of view. Rough architecture of the QEMU emulator can be seen on 3.1.

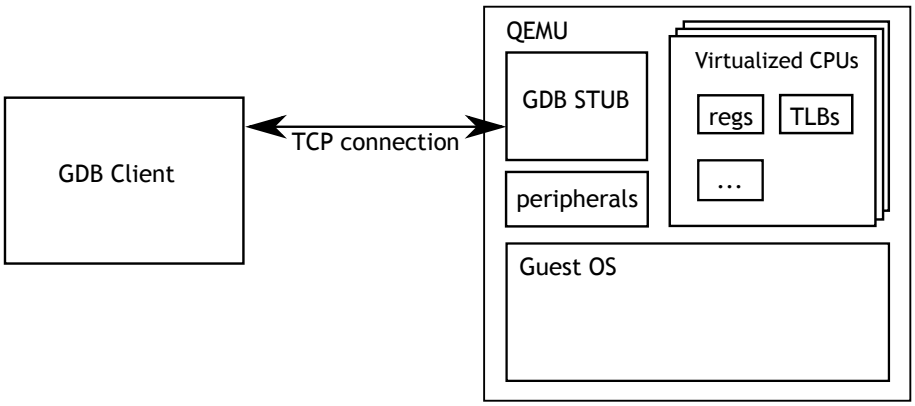


Figure 3.1: Rough QEMU architecture

3.4.1 CPU

Each CPU that is emulated by QEMU is executed in its own thread. These thread then communicate with the rest of the program using message queues. GDB server will receive notification, when any state change of the virtual machine (VM) occurs. This notification is handled by callback function. Such notification can occur in case of CPU trap, VM being shutdown/paused, some internal error, etc. These notifications are used by peripheral modules to start/stop/restart the peripheral when necessary and prevent it from getting stuck (eg. looping sound from sound card).

CPU trap is an exception that happened during execution of the code in virtual machine. The reason for these exception can be division by zero, NMI interrupt, illegal opcode, paging error, memory alignment error, and most importantly debug exception. Debug exception is raised when some breakpoint/watchpoint is triggered or after single instruction stepping.

To stay hidden, when our debugger uses breakpoints/watchpoints, we need to restrict access to special feature registers of the emulated CPU. The guest operating system could access these registers and detect that debugger is running on the system. We will need to replace these registers with dummy ones, when accessing these registers from VM and leave original behavior when placing breakpoints from QEMU itself.

To access CPU state, GDB server has access to the QEMU's internal CPU structure. This structure contains all the registers, flags, floating-point registers, breakpoints, etc. The content of this structure is dependent on the architecture that QEMU is compiled for. GDB server implements helper functions to access the registers in unified way for all different architectures.

3.4.2 Memory

To access the memory of the VM GDB server provides memory access wrappers. Semantics of these wrappers depend on the mode QEMU is running in. In user space mode all memory accesses are threated as physical memory accesses. There are no translation tables involved. And all memory accesses touch the application memory directly.

When QEMU is running in full-system emulation, all memory addresses are translated through CPU memory tables. User needs to be sure that correct memory mapping exists in the CPU, when accessing the memory. Access through this method needs virtual address.

To access physical memory in full-system emulation mode QEMU we can use access functions provided by this allocator. This access does not rely on correct tables in CPU, but user needs physical address to access his data.

3.4.3 GDB stub

The GDB stub is the GDB server implemented in the QEMU. This GDB server implements only small subset of GDB remote protocol commands (see [5]). Implemented commands are:

- CPU register read/write
- Memory read/write access
- Thread list, current thread commands
- Step/continue commands + step type setting
- Monitor command

The memory and the CPU access commands are just wrapper commands for already existing functions. Memory access is always using the virtual memory access (or physical when in user-space mode). The physical access in full-system emulation mode is not available through GDB remote interface.

Every command related to thread is dummy. GDB server does not understand what threads are running in the guest operating system, instead is lists each CPU as one valid thread. When accessing thread's registers/memory, we are accessing the state of that CPU. These dummy commands need to be replaced when implementing our context awareness feature. The actual information about the threads needs to be fetched from the Virtual Machine's memory. These commands include the thread list query commands ("qfThreadInfo" and "qsThreadInfo"), extra thread information

command ("qThreadExtraInfo"), current executing thread ID query("qC"), checking whether thread is alive ("T") and reason for halting the target query ("?").

GDB protocol also implements symbol resolving mechanism. This mechanism is not implemented in QEMU. As our context awareness will depend on symbols, we will need to implement some symbol storage to take advantage of this GDB remote protocol command ("qSymbol").

3.4.4 Monitor command

GDB remote protocol includes command that is not processed by the GDB server, but instead is passed to the server-side command interpreter. QEMU utilizes this feature and command framework has been build that allows to configure various parts of QEMU during runtime. These commands can be executed from GDB command-line by prepending *monitor* keyword. List of all commands available is showed after issuing *monitor help* command.

The monitor command framework makes it easy to add new commands. We are going to exploit this functionality and create own set of commands, to ease configuration of the context awareness module. We will add an interface for passing commands to the scripting engine, and to the scripts running inside it.

3.5 Understanding the Guest Operating system

For our context awareness to work, we first need to understand the inner working of the operating system. We will need to know how internal structures of the operating system look. Our context awareness script will need to implement some of the same functionality of the operating system. All the necessary data is already in the memory, we just need the right tools to collect the information.

This work is going to require documentation of the internal structures of the operating systems. For many major operating systems this information can be obtained from either from some official development package or from unofficial sites that gather reverse engineered knowledge. In case of open-source operating system the information can be obtained from the source code of the system or from technical documentation.

For very basic debugger, we need to replace all the dummy functions that are implemented in QEMU's GDB server (see 3.4.3). These commands include creating list of threads in the system, currently executing thread, and manipulation of their states.

First task is to teach the context awareness script what threads are living inside the operating system. Most of the operating systems store all of the threads inside of some internal storage data structure. This structure can be table, linked-list, or some other more advanced structure. Some analysis on the operating system needs to be done to find a way to access this information and locate it reliably in the memory.

When the correct location is found, the data structure needs to be extracted from

memory to create the list of threads. In many reasonable cases the data structure will contain pointers to the thread structures. To satisfy the needs for thread list and thread extended information commands, script will need to extract unique identification (ID) for each thread and also some additional information about the thread. The additional information depends on the user's use case, as the extended information about the thread is just string that is passed to the user interface without any further processing.

Operating systems store reference to currently active thread in some variable. We can use symbol tables to lookup this variable and determine the location of thread structure. As before the thread structure needs to be analyzed for thread ID. This is enough for GDB to know the currently executing thread.

The obtained information is enough for the GDB to work with thread lists. On the other hand user might be interested in state of other objects in the system. Such objects might be timers, mutual exclusions, and other synchronization primitives, etc. All this information can be obtained from the operating system. GDB, however, does not provide commands for requesting and transferring of such information. This needs to be implemented as part of the "monitor command" menu described in 3.4.4.

3.5.1 Examining and altering state of Thread

To alter state of thread, we need to consider two cases. Thread is :

- currently being executed
- not being executed

The second case can happen for example when thread is waiting in queue to be scheduled, thread is waiting on some IO resource to become available, or thread is suspended. In this case the threads virtual memory address space can be unavailable as the mappings in CPU are used for some other's thread virtual memory space. This problem can be overcome by analyzing the internal data structured that manage the virtual address spaces for each existing thread. After doing so, the physical address could be determined easily and direct access to physical memory can be used.

The problem with this technique is that the internal structures for virtual space management might be very complex and reimplementing the virtual space management would be just too hard. The problem might go even further, and on most modern operating systems memory swaps are present. Swaps are used to free physical RAM memory, and move portions of memory that are not being currently needed to external storage media (eg. Hard-disk drive). This would cause that even with available physical address, accessing the memory would cause us to read/write wrong application's data. This would need another method that would allow as to change memory on the external "swap" device.

Instead of finding the virtual address mapping and dealing with the problems above, we might implement "lazy" access functions. For lazy write, when GDB client sends command to write the thread's memory, we will store the modification, and apply the changes when thread becomes active. While this works well for writing, lazy reads are not easy to be implemented in the same fashion.

For reading the memory of thread not being executed, we might wait for the thread to become active and read the memory afterwards. This can be implemented using technique discussed later (see 3.5.2). This will require to continue execution of the Virtual Machine which can cause other side-effects.

Side-effects can include changes in memory, state of peripherals, timer advances. All of these can be noticed from the guest OS point of view and reveal that system has been halted.

When thread is currently being executed, the problem becomes trivial and we can directly access the memory of the thread. Reading and writing of the memory can be done using the functions provided by GDB server.

Accessing the state of CPU (registers and flags) also needs different handling in both cases. In the second case, when thread is not being executed, needs to be handled similar to the memory access. Operating system usually stores state of the CPU inside thread structure or on the stack of the thread. If such information is stored on the stack then previously mentioned access routines can be used. If the state of the CPU is stored inside of the thread structure, this structure is stored inside kernel's memory space. This memory space does not suffer from mapping being unavailable or the space being swapped to different storage media. Our access routines only need to understand the thread structure to read/write the state of the CPU.

When thread is being executed, the CPU state can be altered directly by calling the functions provided by GDB server and/or accessing QEMU's internal CPU structure.

3.5.2 Finding the relevant thread

When our debugger is able to extract relevant information from the operating system, we want to focus on controlling the virtual machine. As our debugger is undetectable, we are not going to choose which thread at which time, instead we are going to let the virtual machine live it's life and stop it in the right moment.

We want to do this as user is interested in single thread and not all the applications being run in the system. GDB client offers commands for selecting desired target thread. After issuing this command, the thread ID is transferred to our debugger. All consequent commands (read/write memory/CPU state, and step/continue) in the GDB should be executed in the selected thread.

To read and write the memory or CPU state, we are going to use the method described earlier. However, when executing step command, we do not want user to step through all the irrelevant parts of code. Instead we need to make sure that code execution continues in the selected thread.

One possibility of achieving such thing is to single step through the execution silently and stop when the current running thread ID matches the selected thread ID. This approach will work. Problem is that each single step will advance the CPU by one instruction. After this instruction is executed, trap signal is generated. This trap signal will trigger callbacks on all listening modules of QEMU and all callbacks require to be processed before we can execute another step. The overhead in this

situation is too large to be feasible.

Different method is to use breakpoints. Breakpoints in QEMU also generate trap signal and have the same overhead as stepping. When placing the breakpoint on the right place, we can skip major parts of the irrelevant code. The right place might be some return address of the target thread or some context switch function.

The context switch function serves the purpose of switching from one executing thread to another. This is usually done inside the kernel and there is usually only single function that does context switch.

This method breaks the property of being undetected, as in most CPUs the breakpoints can be monitored by looking at some special function registers. Target application could check these registers and notice that some debugger has altered the breakpoints. To overcome this shortage we will need to alter the behavior of QEMU's CPU when accessing these registers. We need to provide some dummy register value, that will always show no debugger activity.

Another problem that might occur is when placing the breakpoint in context switch. Symbol tables only provide us with location of certain symbol, not the size of this symbol. Suppose we are placing the breakpoint on the location of the context switch function. When such a breakpoint triggers, we are entering the function. By checking the currently executing thread, we are going to get the ID of thread that is going to be placed to sleep. What we want, is to know the next thread that is going to be scheduled. Effectively we need to place the breakpoint at the end of the context switch. This will lead to problem of finding the correct address of the end of the context switch function.

3.6 Further analysis

3.6.1 Supporting multi-CPU Virtual machine

QEMU supports multi-CPU environment out-of-box. For our context awareness to work in this multi-CPU environment there are few things we need to keep eye on.

In multi-CPU environment we cannot control which thread of the guest operating system runs on which CPU. Some Operating systems support configuring the affinity of threads. Affinity defines set of CPU where the thread is allowed to be executed. This is a special case and we are not going to assume that our target thread would execute on some single CPU.

We need some way of determining the currently executing thread for each CPU. This might be some table in memory of guest operating system that contains this information for each CPU separately or some other mechanism that is Operating system-specific. This mechanism needs to be implemented in the context awareness script for it to work reliably.

Setting breakpoints as described earlier, will need to be done for each CPU separately. We do not want to miss out target thread, when it is being executed on some different CPU.

To be completely undetectable, we need to be sure that halting one CPU will synchronously halt all remaining CPUs.

Memory access routines (read and write) are going to be CPU specific as well. Only the correct CPU will contain the correct memory mapping inside its tables. If incorrect CPU is chosen, CPU might not contain the memory mapping at all or in worst case, we will write the data into different threads memory (memory corruption might occur).

3.6.2 KVM virtualization

KVM stands for Kernel virtualization. KVM is a virtualization support module for Linux kernel. This module exploit virtualization extension of most recent CPUs. This technology allows virtual machines to be isolated and executed natively on host computer. The speed of emulation is almost native, as only various peripherals need to be emulated.

Most recent versions of QEMU include support for this technology. Our context awareness should have no problem to work in such environment. GDB server provides wrapper functions for low level access for both emulated and virtualized environment.

Chapter 4

Implementation

4.1 GDB Context Awareness

We have expanded the functionality of GDB server by adding custom module to QEMU. The resulting architecture looks like fig. 4.1.

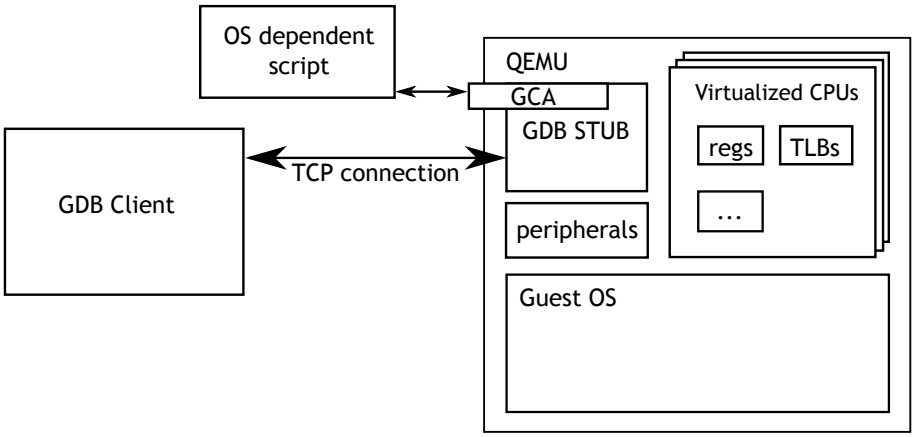


Figure 4.1: QEMU after

We have preserved the default behavior of QEMU and it's debugger. User can use GDB server as before, no semantics of commands have changed at all. This is only true until some context awareness script is loaded.

After loading the correct script into the GDB context awareness module, standard behavior of the GDB server's commands is replaced and execution is redirected to GCA module.

The GCA module consists of scripting engine and bindings that expose the internal functions of GDB server to the scripting engine.

For scripting engine we chose Python ([10]). Python is a high level scripting language that provides easy embedding support and also allows embedding external programs into python modules. In other words, Python can be called from external programming language, as well as call external programming language's functions.

We created a wrapper module that embeds all necessary functions from the GDB stub such as functions to manipulate CPU state, read/write memory functions,

breakpoint management, etc. This module can be loaded from the user context awareness script just like any other python module. The only difference is, when calling function from this module, the actual function is called from the GDB stub. For full list with description see section 4.1.1.

We also created wrapper functions for some expected functionality of the context awareness script. These functions include: functions to create thread-list, functions to access information about threads, accessing memory/CPU state of threads, etc. For complete list see section 4.1.2. These functions replace the original command handling in the GDB stub.

4.1.1 Exposed API

Embedded python implements module GCA. This module contains all functionality exported from the GDB stub. When user wants to use this module inside own script, the "gca" module needs to be imported first. The "gca" module implements following functions:

`gca.target_physical_memory_write(address, data)`

This command writes the physical memory of the emulated system. *Address* argument is the physical address to write to. *Data* is string to be written. Length of written data is the length of string provided. Command returns True when no error occurred.

`gca.target_physical_memory_read(address, length)`

This command reads physical memory of the emulated system. *Address* argument is the physical address to read from. The *length* argument tells how many bytes are read from memory. Command returns *string* containing read data.

`gca.target_memory_write(cpu, address, string)`

The command writes a virtual memory of the emulated system. The *CPU* argument is a pointer to the internal CPU structure of QEMU. Command will write memory using virtual memory mapping of the specified CPU. The *address* is the virtual address to be written and *string* contains the data to be written. Length is specified by the length of the *string*.

`gca.target_memory_read(cpu, address, length)`

The command reads virtual memory of the emulated system. The *CPU* argument is the pointer to the internal CPU structure of QEMU. Command will read memory using virtual memory mapping of the specified CPU. The *address* is the virtual address to be read and *length* is the number of bytes to be read. Function returns *string* with the content of the specified portion of memory.

`gca.target_regs_write(cpu, values)`

This command writes registers of the specified CPU. The *CPU* is pointer to the internal QEMU's CPU structure. *Values* is string with concatenated values of registers to be written to CPU.

`gca.target_regs_read(cpu)`

This Command reads registers of specified CPU. The *CPU* is pointer to the internal QEMU's CPU structure. The command returns *string* containing all register values.

`gca.breakpoint_insert(address, length, type)`

Inserts specified breakpoint on all CPUs in the emulator. The *address* is the address to be watched. The *length* is number of bytes to watch. *Type* specifies type of breakpoint (SW - 0, HW - 1 breakpoint, WRITE - 2/READ - 3/ACCESS - 4 watch-point).

`gca.breakpoint_remove(address, length, type)`

Removes specified breakpoint from all CPUs in the emulator. The *address* is the address of the breakpoint to be removed. The *length* is number of bytes to watch. *Type* specifies type of breakpoint (SW - 0, HW - 1 breakpoint, WRITE - 2/READ - 3/ACCESS - 4 watch-point). All arguments need to be exactly the same as when the breakpoint was inserted.

`gca.breakpoint_remove_all()`

The command removes all breakpoints on all CPUs in the emulator.

`gca.monitor_output(string)`

This command writes text to the connected monitor - connected GDB client. This command can be used to output text messages from the script to connected user. Note: This command should not be called when already processing command from the GDB. Reply generated by this command would falsely be interpreted as reply from the processed command.

4.1.2 QEMU Hooks

The context awareness module requires certain functions to be implemented in the context awareness script. These functions serve as replacement for dummy implementations in the GDB server. The loaded script needs to implement following functions:

Symbol storage functions

The following commands belong to symbol storage group. They are invoked when the remote GDB client is ready to serve lookup of unknown symbols. (GDB protocol "qSymbol" command) When context awareness script wants to utilize symbol resolving from the remote GDB client, context awareness script needs to implement some storage for unknown and resolved symbols. These functions provide the interface to that storages.

`symbol_getunknown()`

This callback is invoked after the GDB client sends query command to the GDB server. The script is then polled for all unknown symbols, one by one. This function returns the name of the symbol the script wishes to resolve.

The GDB server will be polled until returned name is zero-length string.

`symbol_add(name, location)`

This function is called after successfully resolving symbol's location. This function is called with *name* and *location* of the resolved symbol. Context awareness script is expected to store this information inside local storage.

Thread list functions

These commands belong to the thread list group. Commands from this group are invoked when creating responses to "qfThreadInfo" and "qsThreadInfo" GDB remote protocol queries.

`threadlist_create(cpu)`

This command is invoked when first query is received ("qfThreadInfo") and should generate the list of existing threads. This list is to be stored internally in the script. The CPU is the pointer to the CPU structure that should be used during creating the thread list.

`threadlist_count(cpu)`

This command is used to check the count of threads in the thread list. When the returned count is 0, the terminating packet is generated, this indicates that the end of list was reached.

`threadlist_getnext(cpu)`

This command returns the thread ID that is next on the list. The returned ID should be long integer and should be unique for each existing thread. After calling of this function, the returned ID should be removed from the list (thus decreasing count by 1).

The commands in this group are always called in the fashion of this pseudo-code:

```
threadlist_create(...)
while(threadlist_count(...) > 0):
    t = threadlist_getnext(...)
    gdb_send_response(t)
```

Thread information

These functions are involved in getting information about certain thread.

`thread_getinfo(cpu, id)`

This command generates reply to "qThreadExtraInfo" command. This function should generate a printable string. This function can return any extra information that might be useful for the user (for example: name, extra flags, etc). This information is presented to user, when thread list is printed in the GDB (command "info threads")

The CPU is pointer to the CPU state structure, and the id is the ID of the thread.

`thread_isalive(cpu, id)`

This function is used to determine whether thread is still existing in the system. The function should return $True$ if the thread with the ID still exists.

`thread_getcurrent(cpu)`

This function is invoked when the remote command "qC" is received. This function returns the current thread on the selected CPU. The CPU is the pointer to the QEMU's CPU structure.

`thread_regs_read(cpu, id)`

This function is used to read registers and CPU state of some thread. The function should differentiate between active and blocked threads. The CPU is the pointer to the QEMU's CPU structure, this CPU is to be used for the read operation, the id is the thread ID.

`thread_regs_write(cpu, id, values)`

This function is used to write registers and CPU state of some thread. The function should differentiate between active and blocked threads. The CPU is the pointer to the QEMU's CPU structure, this CPU is to be used for the write operation, the id is the thread ID.

`thread_mem_read(cpu, id, address, length)`

This function is used when reading memory space of some thread. The function should differentiate between active and blocked threads. The CPU is the pointer to the QEMU's CPU structure of the CPU to be used for the read operation, id is the unique thread ID. The *address* is the location in the memory, and the *length* is the number of bytes to be read.

`thread_mem_write(cpu, id, address, data)`

This function is used when writing memory space of some thread. The function should differentiate between active and blocked threads. The CPU is the pointer to the QEMU's CPU structure of the CPU to be used for the write operation, the id is the unique thread ID, the *address* is the location in the memory, and *data* is the data to be written.

Other functions and hooks

`monitor_command(command, argument)`

Monitor command is invoked when remote GDB client sends "monitor gca cmd arg" command. Two additional parameters (*command*, *argument*) of the command are passed as arguments of the function call. This command can be used to implement simple menu, either to show some internal data from the operating system, or setting some valued (in system, in the script, etc).

Eg: monitor gca print objects This command will invoke function as: monitor_command("print", "objects")

`hook_signal_trap(cpu)`

This callback is called when the GDB server receives a trap from *CPU*. This can occur when some breakpoint is reached - either when single stepping or hitting some breakpoint.

Return value of this function determines the behavior of the GDB server's trap handler.

- 0 - let the GDB server's handler handle the TRAP.
- 1 - same as 0, and notifies GDB server that GCA script reached configured breakpoint.
- 2 - continue CPU execution.
- 3 - continue CPU execution in single step mode.

`hook_load()`

This callback function is called when the script is loaded into the GCA module. It may be used to load some settings into the memory of the guest operating system, or do some preliminary analysis.

`hook_unload()`

This callback function is called before the script is unloaded from the GCA module. It should be used to cleanup all script specific stuff from the virtual machine (breakpoints, etc.).

`hook_symbols_loaded()`

This callback function is called after the symbol resolving has finished. This callback can be used similarly to hook_load, but for things relying on symbolic names.

`hook_continue(cpu, tid, type)`

This callback function is called when the GDB requests continuation of the program. The *CPU* is the pointer to the QEMU's CPU structure. *Tid* is the target thread ID. *Type* is the type of continuation:

- type = 0 - continue is requested
- type = 1 - stepping is requested

Return value of this function can override the default behavior of the GDB stub. Meanings of return values:

- 0 - default behavior
- 1 - execution will continue
- 2 - execution will step

```
hook_step(cpu, tid)
```

This callback function is called when the GDB requests stepping of the program (command "s"). The *CPU* is the pointer to the QEMU's CPU structure. *Tid* is the target thread ID. Return value of this function can override the default behavior of the GDB stub. Meanings of return values:

- 0 - execution will continue
- 1 - execution will step

4.1.3 Monitor commands

Our GCA module adds two commands to QEMU's monitor menu. First command (gca_script) serves for purpose of loading and unloading of the GCA script. This commands format is following:

```
gca_script (load|unload) [script_name]
```

Load argument requires script name to be present when executing this command. Load command will load the specified script into the GCA module. When unload command is issued, the script in the GCA is unloaded and the whole GCA module is disabled.

Second command is for sending commands to the loaded script. The format is following:

```
gca command [argument]
```

This command will execute the "monitor_command" function inside the script with "command" as first argument and "argument" as second argument (if exists, empty string otherwise). This command can be used to control the behavior of the loaded script during runtime.

These commands can be sent from the GDB client by monitor command:

```
(gdb) monitor gca_script load script_name
```

4.2 Helper library

During the development cycle we have identified some recurring functions. We have isolated these functions into a standalone library. These functions are not OS dependent, some of them might be CPU endianness dependent.

This library provides functions to help extraction of some basic data structures from memory.

```
deref_ptr(cpu, ptr, t = "I")
```

Dereferences pointer *ptr* to the type specified by argument *t*. The argument *t* is optional and defaults to 32bit integer. The type is specified using the format characters as defined for "struct" python module. *Cpu* is the pointer to QEMU's CPU structure.

```
read_table(cpu, ptr, cnt, payload_type = "I")
```

Reads sequence of data from memory of *cpu*. *Ptr* defines the address of table in memory, *cnt* defines number of elements. $Payload_type$ defines the type of element. This argument is optional and defaults to 32bit integer. When multiple format characters are specified, each element of table will be tuple of types specified in the format string.

```
read_linkedlist(cpu, ptr, payload_type = "I")
```

Reads linked list from memory of *cpu*. *Ptr* is the pointer to the first node of linked-list. $Payload_type$ specifies the elements stored inside each node. The algorithm will output array of elements read from the memory. Read continues until *ptr* is null. When multiple format characters are specified, each element of the resulting table will be tuple of types specified in the format string.

This algorithm assumes that *next* pointer of the linked-list is the first element of the node. C language defined structure would look like this:

```
struct node_t {
    struct node_t *next;
    ... rest of the node ...
```

```
read_string(cpu, ptr, limit = 100)
```

This function reads null-terminated string from memory. *Ptr* is the address of the string. *Limit* argument serves as upper boundary for the string length.

4.3 x86 CPU support

Some very basic support class has been implemented for x86 CPU. This class implements all features that are used in the debugger and are specific to x86 architecture.

This class is mainly used to store the state of the CPU and convert the state to different forms. One form is where all register values concatenated into single string. This form is used during receiving and sending register values from/to the GDB. This form is also used when using functions like: thread_regs_read, thread_regs_write and target_regs_read and target_regs_write.

When working with such format it is hard to access single value. Programmer would need to know the exact index of the value, size of the register, etc. This class contains all the names of the registers with their respective sizes. The programmer is presented with interface to access the single values of the registers.

This class should also implement any other CPU specific functions, such as various memory address translations, floating point registers and states, co-processor states, etc.

Some registers of the CPU can by larger than the largest storage type available to the programming language. For this reason all register values are stored in string format and need to be converted when user wants to work with them.

```
i386.__init__(cpu = False)
```

The constructor of the class will initialize the registers to zero values. If user provides CPU during initialization the register values are read from the specified CPU.

```
i386.reg_size(name)
```

Returns the size (in bits) of the specified register. The register *name* has to be specified exactly. If no such register is found, method returns $False$.

```
i386.print_regs(regs = False)
```

This method is used to print the values of registers to the console. If no argument is specified, internal set of registers will be print. If user specifies registers, this method will print these registers. *Regs* is of dictionary type with register names as keys and values as values of these keys.

```
i386.set_gdb_regs(data)
```

This method will parse the *data* string and store the values of the registers inside the class. The string is in the GDB register format.

```
i386.get_gdb_regs()
```

This method returns the GDB register formated string. This string represents values of registers that are stored inside the class.

```
i386.get_reg(name)
```

Returns value of single register. The register value is returned in string format. Function will return *False* if incorrect *name* of register is given.

`i386.set_reg(name, value)`

Sets the specified register to value. The *value* is in string format. If register *name* is not found or the size of the *value* does not match expected size, *False* is returned.

`i386.get_iopl()`

Returns the privilege mode level of the CPU. This 0 being the most privileged and 3 being the least privileged mode.

4.4 RTEMS operating system

4.4.1 RTEMS Internals

To properly support RTEMS system, we need to understand how internal structures are organized.

One of the most interesting table in the system is "Object Information Table". This table contains information about all objects of the operating system (threads, mutexes, queues, periods, tasks, etc.). These objects are sorted by the different API they belong to. Object information table contains as many rows as there are supported APIs. Each row contains a pointer to another table. This table contains information about the objects. Each API table contains as many rows as there are different tables. For example "internal API" contains only 2 different objects: "threads" and "mutexes" and "Classic API" contains many more objects ("tasks", "timers", "semaphores", "queues", "partitions", "regions", "ports", "periods", "extensions", "barriers").

Location of the "Object information table" in memory can be obtained from global symbol "_Object_information_table".

Each entry of the "Object information table" contains table (member "local_table") that stores pointers to all objects of this type. The OI also contains number of objects existing and other additional information.

Each "local_table" object's type is composed of smaller types. Each type has common "header". All types are composed of "Object_Control" class and some object specific members. The Object_Control class contains unique ID of the object, name of the object and some linked-list specific information (next, prev pointers etc).

Threads of the Classic API are of type Thread_Control. From each Thread_Control class we can get all the necessary information about each thread. This class contains information such as: stored register in case of non-running thread (both general purpose and floating-point registers), state of the thread (dormant, suspended, waiting for other object, ...), ID and various priorities, scheduling parameters, and more.

In the default configuration, context switch is not executed from interrupts. Context switch is done on as needed basis, when application calls some system call.

The need for scheduling is determined by scheduler which will set some system flags when some higher priority task has been released. This however highly depends on selected scheduler, which is selectable for each thread individually.

The context switch function's name is "_Thread_Dispatch". This function is called by wrapped function "_Thread_Enable_dispatch" from every function of every API. This fact can be exploited. Each thread that is not currently being executed, will contain the context switch function in its back-trace.

4.4.2 Finding the correct thread in RTEMS

Let's suppose that user has identified the target thread and knows the ID. The target thread is ready, but not currently running. The back-trace might look like this:

```
#0  0x00114756 in _Thread_Dispatch ()
#1  0x001113a1 in _Thread_Enable_dispatch ()
#2  0x0011140c in rtems_task_wake_after ()
#3  0x0010037e in Task_Relative_Period (unused=3) at tasks.c:167
#4  0x0011e4c3 in _Thread_Handler ()
#5  0x85b45d8b in ?? ()
```

Position 0 is the actual EIP as stored in the Thread_Control structure. All other items are made available using the stack unwinding technique. Item at position 3 is the main loop of he target thread. Position 2 is system API function called from the main loop. This function then called the context switch wrapper - position 1 and the current position of the program counter is number 0.

We know that the RTEMS calls context switch when application thread executes any API function. So in any thread that is not being currently executed the back-trace will look similar. Difference will be, which API function caused the context switch.

To wait for the thread to become ready we need to correctly choose the place for the breakpoint. After the breakpoint triggers, we will need to check that our target thread is really the one being executed.

To place the breakpoint we will choose one address from the back-trace. This will be the place where the execution will eventually return to. If we place the breakpoint at position 0, we will get notified every time, the context switch is done. If we however choose position 2, the chance that an other thread used the same system function is low, thus the overhead of breakpoint handling is much lower. On the other hand, in this case we would need to remove the breakpoint a place it in the next system call the function is going to use. We would have to replant the breakpoint each time it is triggered. In the first case this would not be necessary at all.

Additional check whether the target thread is the current running thread is necessary. More than one threads with the same code can be running on the same system.

Problem might occur when instruction stepping leaves from the current thread and execution of some other thread begins.

This would probably work, as the stepping would eventually hit the breakpoint in the context switch function. This would cause to advance the execution to point where our target thread is again ready to be executed.

Better solution would be to check against some memory range/table of system calls and automatically switch from single stepping to continuing until our target thread is back.

4.4.3 RTEMS parser class

This class implements basic support for the RTEMS operating system. We have implemented parsers for few important structures of the operating system. These include:

- Object_Information_table
- Object_Information
- Object_Control
- Thread_Control

All of these are needed to support the very basic needs of the context awareness module.

This class is also responsible for filling the symbol tables with unknown symbols at the very beginning.

Most important methods are:

`rtems.get_threads(cpu)`

This method extracts all threads from the operating system. Returns a list of the threads. The thread list is also stored inside of the class for later use. Each member of the list is of *dict* type and contains various information about the thread: ID, name, state, saved registers, etc.

`rtems.get_current_threads(cpu)`

Returns the ID of currently executing thread.

`rtems.print_objects(cpu)`

Prints all objects of the operating system. This class will iterate through all objects of all APIs and print ID, name.

`rtems.get_thread_registers(cpu, tid)`

This thread returns the registers of blocked thread. This function will extract the state of CPU of some blocked thread and return string in GDB register format.

4.4.4 Context awareness script

The example script's listing can be found in appendix D. This script implements all the logic necessary for our context awareness to work. This script implements all the QEMU hooks described earlier.

For thread list functions, this script depends on the output of the RTEMS OS parser. The actual list of threads is updated only when checking whether thread is alive and when the thread-list commands are used (more specific "qfThreadInfo" command). All other commands rely on the cached value. This is reasonable trade-off between overhead caused by accessing the memory of guest system, and the freshness of the data. We are also assuming that GDB will check the thread is still alive reasonably often (after each break of the execution, etc).

All remaining thread functions are simple look-ups into the already generated thread-list.

The symbol storage has been implemented in separate class. The symbol storage is filled at the beginning by the RTEMS class, and the symbols are resolved when the GDB client connects. The symbol class also supports calling user specified function that will be executed when all unknown symbols have been resolved. This functionality is used to bootstrap the context awareness script.

We have created a support for multiple breakpoint handling. This support consists of methods for adding and removing breakpoints and dispatcher that can recognize which breakpoint was triggered. When adding a breakpoint, user needs to specify address, size, type of breakpoint and also the callback function. Callback function is called from the dispatcher when the breakpoint is triggered. This framework enables us to have multiple breakpoints planted in the system and each location is used to manipulate different things.

We are using two breakpoints out of total four available on the emulated system. This limits the number of breakpoints that are available for user from remote GDB client. Care should be taken, not to remove the script's breakpoints from the remote GDB client. Otherwise the context awareness script will not be working as expected.

The breakpoints are used for:

- finding our target thread
- lazy memory writes

Each memory write that targets thread, that is not being current one, is stored as a patch inside our script. The breakpoint is set to a location inside the context switch function. When this breakpoint triggers, the ID of the current thread is checked against available patches in the script. When there are any memory patches stored, all are written to the memory at once. The fact that the breakpoint is placed inside the context switch function makes sure, that the thread will run with the modified memory.

Finding our thread is done by analyzing the back-trace of the thread when not being currently executed. This is done in fashion as described in previous section. We are placing the breakpoint at the first item in the back-trace. When the breakpoint is triggered the script checks following conditions:

- target thread is alive
- current thread ID == target thread ID

If both conditions are true, we halt the execution of the virtual machine and user is notified that target thread is again the current one.

The script also implements one sample remote command. This command can be invoked from the GDB using the "monitor" command. Its purpose is to print all objects found inside the Object information table.

4.5 Linux

In this section we are going to look at the Linux operating system and what does one need to understand to be able to implement context awareness script.

Please note that the following information does not apply for every possible configuration of the Linux kernel. Our specific configuration has been mentioned earlier (see 2.3).

4.5.1 Linux internals

The Linux kernel is much more complex than the RTEMS in terms of usage of programming language features. The Linux kernel uses heavily C programming language's macros. These macros make it possible to hide complex expressions and ease work of the programmers. On the other hand understanding these constructs will give many analysts headaches.

The most interesting part of kernel for us is the "task_struct". This structure contains all information about a task running in the system: PID, GID, state of task, priorities, exit states, references to parent/children/sibling task, reference to thread_struct, usage counters and many many more. The size of this structure is about 800bytes for our configuration.

To find the tasks running in the system, we will look at the "init_task". This is the first task that exists in the system. Each task_struct is part of doubly-linked-list, so it is no problem to iterate through this list and extract the information about tasks in the system.

For non-SMP configuration to get to the current executing task we just need to look at the "current_task" variable and it contains pointer to the currently executing task's task_struct. For the SMP configuration this is getting a bit more complex. As there are multiple CPUs running in the system, there are also multiple current tasks being executed. The Linux kernel provides "per-cpu" allocation facility for these purposes. When a variable is declared as per-cpu, there will be as many variables of the same name as maximal number of CPUs supported. This is done by defining a table of variables. This table is initialized at the startup and copied as many times as the maximal number of CPUs. The original table is then discarded. The access to these variables is done by calculating the per-cpu offset and adding the offset of the accessed variable in the per-cpu table.

The thread_struct is a CPU specific structure. This structure contains saved state of the CPU. For x86 architecture these are: stack pointer, stored segment registers, ptrace and debug information, fpu state, etc. If the thread we are inspecting is blocked (not being the current thread), we can read the state of registers of the CPU from the stack. Stack pointer is already provided by the thread_struct. Registers are stored according to the pt_regs structure. To extract this register structure from task_struct there is a macro ("task_pt_regs") that will calculate the correct address and provide pointer to such structure.

To find out when our target thread is active, we are going to hook the context switch function. The context switch function itself is not available in the symbol table of the kernel image. We are therefore going to use the "__schedule" function. This is the main scheduler function, which switches running tasks when necessary. We are going to place the breakpoint at the end of this function. When this breakpoint triggers and we are certain that current task is the target task, we are not quite ready to alert the user. The CPU is very likely in privileged mode and in the kernel context. We can either step until the instruction pointer stored on the stack is accessed, or use second breakpoint, set to the location of the stored instruction pointer. The second breakpoint would only be temporary, as the instruction pointer will be different each time task is scheduled.

If we realize that kernel is running in privileged mode and tasks run in unprivileged mode, we could initiate stepping from the first breakpoint until we reach place where the privilege mode drops. This mode can be checked by looking at the "eflags" register of the CPU (I/O Privilege Level).

4.5.2 Linux support class

This class implements support for extracting certain structures of the Linux kernel from VM's memory. Currently supported structures are:

- list_head
- thread_struct
- task_struct
- pt_regs
- current_task

This is the minimum requirement to support listing tasks available in the system, getting the current task, getting state of blocked tasks.

```
linux.get_threads(cpu)
```

Returns the list of threads extracted from the system. This list contains both kernel threads and user-space threads.

```
linux.get_current_threads(cpu)
```

Returns the PID of the current task.

`linux.get_thread_registers(cpu, tid)`

This function extracts register values from the task, while this task is blocked. The registers are returned in GDB format.

`linux.set_thread_registers(cpu, tid, regs)`

This function replaces the register values of the blocked thread with the values specified. *Regs* is a dist object, containing the register names as keys and register value's as values of the keys. Registers that are not available in the dictionary are not altered at all.

4.6 Problem with data structures

Some common problems emerged when trying to implement the support for any operating system. Even though the user has access to the source code, the binary representation of the actual code and data structures highly depends on compiler's optimizations and configurations.

If we take two examples of structures in C programming language:

```
struct first
{
    char one;
    long two;
    char three;
};

struct second
{
    char one;
    char two;
    long three;
};
```

When program with such structures is compiled, the size of first structure will be different from the size of the second. The first structure's size will be 12, while the size of the second structure will be 8! This is also valid only for 32bit architecture. For 64bit architecture, sizes will be twice as much (24 and 16 respectively).

This is due to alignment of the native types on some boundary. Such optimization will reduce the number of unaligned accesses which are often slower. Some architectures do not allow unaligned access.

Such behavior can be overcome by using special directives of the compiler. When using the GCC compiler, user can change the attribute of the type to packed. This will make sure, that members are placed in the same order and no padding is inserted among them.

```
struct __attribute__ ((__packed__)) first
{
    char one;
    long two;
    char three;
};

struct __attribute__ ((__packed__)) second
{
    char one;
    char two;
    long three;
};
```

Now the size in both cases is equal to 6.

Getting the actual size and binary structure of the data type can be tricky. We either need some good documentation, in case of binary distribution of the program. Or we can compile the sources and make few debug outputs, that will tell us the size and offset of the interesting members. Such thing can be accomplished using the "offsetof" macro. This macro requires the actual data type and the name of the member. The output will be offset in bytes from the beginning of the structure.

Chapter 5

Conclusion

In this thesis we have showed how to change the QEMU emulator into a basic undetectable debugger. We created a simple, yet powerful framework, that allows user to define support for virtually any operating system available. Two examples for two different operating systems have been created to demonstrate the undetectable debugger. Each example provides the debugger with all necessary information about the running operating system, that are needed to limit the debugging session to single target.

This debugger provides user with same comfort of debugging as when debugging using standard user-space debuggers. The developed debugger is virtually undetectable from the operating system running in the virtual machine, as our debugger is in full control of the state of the emulated computer. The debugger has only basic features like: selecting target thread in the system, skipping over unrelated parts of code, altering state of the target thread or any other thread existing in the system, altering the CPU state. Our created debugger is very flexible in terms of adding new features. User could modify the existing examples and change the behavior quickly.

One problem is that the debugger is highly volatile to changes in configuration of the system. The provided examples are proof-of-concept and have hard-coded internal structures of the operating system. The first example has been developed for the RTEMS operating system, this example serves as easy to understand reference implementation of the context awareness. The second implemented example is for the Linux operating system. This example is for more common operating system that can be found in this real world.

Future updates of this project should include dynamic parsing of source codes and automatic generation of the structure parsers. Such work would make it easier to implement context awareness scripts for other operating systems and would help to provide support for different versions of already implemented ones. Future versions of the debugger should also include support for other wide-spread operating systems such as Windows, *BSD systems and OS X systems.

Appendix A

CD content

- thesis - LaTeX source of this thesis, with compiled text in PDF
- sw
  - source-qemu - sets correct environment path for QEMU
  - source-rtems - sets correct environment path for RTEMS development environment
  - rtems-run - RTEMS system testing environment
    - hda/rtems-grub.cfg - grub configuration file
    - hda/triple_period.exe - RTEMS example
    - boot.img - FD boot image with grub
  - linux - build environment for Linux test image
  - linux-run - Linux system testing environment
    - .gdbinit - init file for GDB
    - vmlinux - kernel image, before stripping
    - openwrt-x86-kvm_guest-vmlinuz - Linux image with embedded ramdisk
    - run.sh - script to start the VM
    - run-console.sh - script to start the VM with serial console
  - scripts
    - rtems.py - RTEMS support class for GCA
    - i386.py - x86 support class for GCA
    - gca_symbol.py - symbol storage class for GCA
    - gca_util.py - GCA helper library
    - run.sh - script to execute the testing environment
    - gca_rtems.py - GCA script with RTEMS support
  - build-qemu - build script for QEMU
  - qemu - QEMU installation directory
  - qemu-source - QEMU source directory (with GCA support)

  - rtems - RTEMS installation directory (compiler and libraries)
  - rtems-install
    * tool - toolchain sources + build scripts
      · build - build script for RTEMS toolchain
      · gdb-7.3.1.tar.bz2
      · gcc-g++-4.4.6.tar.bz2
      · binutils-2.20.1.tar.bz2
      · gcc-core-4.4.6.tar.bz2
      · newlib-1.18.0.tar.gz
      · newlib-1.18.0-rtems4.10-20110518.diff
      · gcc-g++-4.4.6-rtems4.10-20110829.diff
      · gcc-core-4.4.6-rtems4.10-20110829.diff
      · gdb-7.3.1-rtems4.10-20110919.diff
      · binutils-2.20.1-rtems4.10-20100826.diff
    * rtems - main sources + examples build directory
      · build - RTEMS build script (requires correct toolchain)
      · build_examples - RTEMS examples build script (requires built RTEMS and toolchain)
      · examples-v2-4.10.2.tar.bz2
      · rtems-4.10.2.tar.bz2

- readme.txt - content of the CD

Appendix B

Building modified QEMU

Provided on the enclosed CD is the source code of the modified QEMU software. This source code contains all the necessary parts to build QEMU with GDB context awareness implemented in this thesis.

To build this source you can follow these instructions:

```
PREFIX=/usr/local/
TARGETS="i386-softmmu"

cd qemu-source
./configure --prefix=${PREFIX} --target-list="${TARGETS}"
make
make install
```

This will install QEMU into the /usr/local/bin directory on your system. If other directory is needed, you may alter the PREFIX to your needs. Make sure that after executing the configure command you get following line in the output:

```
gca support       yes
```

If everything finished without problem you should end up with working qemu emulator, as used during the development of the context awareness module.

Appendix C

Example Usage

In this chapter we are going to show two example sessions. One for each implemented system. We are going to show basic commands that are implemented and how the GCA should be used. Before we begin with the examples, we are going to show, how to use GDB in remote environment.

C.1 GDB

For each example we created ".gdbinit" file. This file gets loaded by GDB on startup and GDB will execute every command from this file. This file can also serve to define macros in GDB. This is rather useful when repeating some complex command sequences.

Our example .gdbinit file from Linux test environment looks like this:

```
target remote 127.0.0.1:10000
monitor gca_script load gca_linux
add-symbol-file vmlinux 0
info threads
```

First line commands GDB to connect to remote target. User should provide IP address or host-name with correct port, where the GDB server is listening. When connecting to the QEMU user needs to be sure to enable the GDB server in QEMU by specifying the host-name and the port where QEMU's GDB server will listen. In this example, we are executing QEMU using the following command line:

```
qemu-system-i386 -gdb tcp:localhost:10000 -kernel vmlinuz
```

The second command will tell QEMU to load "gca_linux" script into the GCA module. The third command will load symbols from external file into the GDB. Last command will show the threads that are available in the system.

After executing the GDB in the same directory as this .gdbinit file, the GDB will connect to the already running QEMU and print the threads from the system. After this the debugging session can start.

C.2 RTEMS

In this section we are going to show example of the RTEMS context awareness script. We assume that user already has QEMU with the proper GCA support installed. After executing the example in rtems directory, user will see example program from RTEMS operating system.

To debug this example program. We will connect to the QEMU using standard GDB client. We created simple gdbinit file, that will help to setup the context awareness module.

```
target remote 127.0.0.1:10000
monitor gca_script load gca_rtems
add-symbol-file hda/triple_period.exe 0x00100000
info threads
```

This will command the GDB to connect to remote target and load the proper symbol file and the GCA script.

To see the threads that are recognized by the operating system, user can use the GDB command for listing threads.

```
(gdb) info threads
```

This command will trigger the GCA module, to reload the current list of threads and pass them through the GDB server to the GDB remote client:

```
(gdb) info threads
  Id   Target Id         Frame
  5    Thread 151060481 (IDLE 0000) 0x00116b9e in _CPU_Thread_Idle...
  4    Thread 167837698 (TA1  0010) 0x00114756 in _Thread_Dispatch ()
  3    Thread 167837699 (TA2  4000) 0x00114756 in _Thread_Dispatch ()
  2    Thread 167837700 (TA3  0008) 0x00114756 in _Thread_Dispatch ()
```

In this example we can see that the operating system contains three threads and one idle thread. All of the threads are blocked in the context switch of the Operating system. This makes the idle thread an active thread. (Not yet pictured in the listing, as the GDB does not refresh the current thread on demand)

To select a target thread, we need to use the following command:

```
(gdb) thread number
```

The number is not the ID of the thread recognized by the system, but the index number in the listing of threads in GDB. If we were to select thread named "TA3", we would need to issue following command:

```
(gdb) thread 2
[Switching to thread 2 (Thread 167837700)]
#0  0x00114756 in _Thread_Dispatch ()
```

This commands selects the thread as target thread, and stores the selection inside the GDB. This command does not transfer the selection to the GDB server, so no action is taken at the QEMU side.

At this point user either want's to continue the execution, or start stepping of the thread. When continuing the execution, no action is taken by the GCA script. The virtual machine will continue to operate as usual.

```
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
[Switching to Thread 151060481]
0x00114756 in _Thread_Dispatch ()
(gdb) info thread
  Id   Target Id         Frame
* 5    Thread 151060481 (IDLE 0000) 0x00116b9d in _CPU_Thread_Idle...
  4    Thread 167837698 (TA1  0010) 0x00114756 in _Thread_Dispatch ()
  3    Thread 167837699 (TA2  4000) 0x00114756 in _Thread_Dispatch ()
  2    Thread 167837700 (TA3  0008) 0x00114756 in _Thread_Dispatch ()
```

As we can see in this example, the thread number 5 is the current thread. (star next to the thread number)

So again we select thread two as the target thread and this time use the step command. The GCA script will override the step command with continue and advance the execution until the target thread is not current thread.

```
(gdb) thread 2
[Switching to thread 2 (Thread 167837700)]
#0  0x00114756 in _Thread_Dispatch ()
(gdb) step
Single stepping until exit from function _Thread_Dispatch,
which has no line number information.
Cink - Your thread is ready!
Task_Relative_Period (unused=3) at tasks.c:160
160     status = rtems_clock_get( RTEMS_CLOCK_GET_TOD, &time );
(gdb)
```

As we can see, we are in the main loop of the thread TA3.

The RTEMS GCA script has been designed to override the commands of the GDB to always keep the target thread current thread. Whenever the execution reaches context switch and the current thread variable is changed, the GCA will advance the execution back to the target thread. This is accomplished by analyzing the call-stack, and placing breakpoint at the correct instruction.

Appendix D

Example GCA script listing

```
from struct import pack, unpack, calcsize
import gca
from gca_util import *

import mempatch
import symbol
import rtems
import i386


thread_list = {}

symbol_storage = symbol.symbol()
target_os = rtems.rtems(symbol_storage)

# pointer to the QEMU's first cpu
first_cpu = 0

# this is the thread, user wants to debug
target_tid = 0

breakpoint = {}

mpatch = mempatch.mempatch()

def hook_unload():
  breakpoint_removeall()

def hook_symbols_loaded():
  print "Symbols loaded"

def hook_load():
  print "RTEMS script loaded"
```

```
def hook_continue(cpu, tid, t):
  # if continue requested, just continue
  if t == 0:
    print "continue: continue as requested"
    return 0

  # if the thread is the current thread, just
  # step as requested
  ctid = thread_getcurrent(cpu)
  print "Current thread %x" % ctid
  if ctid == tid:
    print "continue: stepping as requested"
    return 2

  # the thread requested to step is not current, make it current
  thread_settarget(cpu, tid)
  print "continue: continuing to target"

  return 1

def hook_step(cpu, tid):
  ctid = thread_getcurrent(cpu)
  print "Current thread %x" % ctid
  #
  # if we are stepping the current thread, just step
  if ctid == tid:
    print "step: stepping as requested"
    return 1

  # if not in the current thread, we need to set breakpoint
  # and continue
  thread_settarget(cpu, tid)
  print "step: continuing to target"
  return 0

def hook_signal_trap(cpu):
  global breakpoint

  if len(breakpoint) == 0:
    return 0

  # check which breakpoint triggered
  x86_cpu = i386.i386(cpu)
  eip = x86_cpu.get_reg("eip")
  eip = unpack("I", eip)[0]
```

```
  print "hook: EIP = %x" % eip
  print breakpoint

  for bp in breakpoint.keys():
    if bp[0] == eip:
      print "Found breakpoint"
      callback = breakpoint[bp]
      return callback(cpu, bp)

  return 0

def mempatch_callback(cpu, bp):
  global mpatch

  if mpatch.empty():
    return

  ctid = target_os.get_current_thread(cpu)
  if mpatch.exists(ctid):
    mpatch.apply(cpu, ctid)

  return 2

def thread_callback(cpu, bp):
  global target_tid

  ctid = target_os.get_current_thread(cpu)

  # need to check that thread exists
  if not thread_isalive(cpu, target_tid):
    gca.monitor_output("Thread %08x no longer exists" % target_tid)
    thread_settarget(cpu, 0)
    return 0

  # user doesn't care about other threads
  if ctid != target_tid:
    return 2

  gca.monitor_output("Cink - Your thread is ready!")

  breakpoint_remove(bp)

  return 1

def breakpoint_removeall():
```

```
  for bp in breakpoint.keys():
    gca.breakpoint_remove(*bp)

def breakpoint_exists(bp):
  global breakpoint
  return bp in breakpoint.keys()

def breakpoint_exists_cb(callback):
  global breakpoint

  for bp in breakpoint.keys():
    if breakpoint[bp] == callback:
      return bp

  return False

def breakpoint_remove(bp):
  global breakpoint

  del breakpoint[bp]
  gca.breakpoint_remove(*bp)

  print "Breakpoint removed"

def breakpoint_add(addr, size, typ, callback):
  global breakpoint

  bp = (addr, size, typ)
  if bp in breakpoint.keys():
    print "Breakpoint exists"
    return

  breakpoint[bp] = callback

  if gca.breakpoint_insert(*bp):
    print "Breakpoint inserted"
    print breakpoint
  else:
    raise "Unable to insert breakpoint"

def thread_settarget(cpu, tid):
  global target_tid

  bp = breakpoint_exists_cb(thread_callback)
  if tid == target_tid:
    return True # nothing to do
```

```
  if not bp == False:
    breakpoint_remove(bp)

  if tid == 0:
    target_tid = 0
    return True

  print "set target = %08x" % tid

  if not thread_isalive(cpu, tid):
    print "not alive"
    return False

  if thread_getcurrent(cpu) == tid:
    print "not implemented yet"
    return False

  # ebp is a "framepointer"
  fp = unpack_int(thread_list[tid]["registers"]["ebp"])

  # read the backtrace
  backtrace = read_linkedlist(cpu, fp)
  print "backtrace is " + str(backtrace)

  breakpoint_add(backtrace[2], 4, 1, thread_callback)

  target_tid = tid
  print "Target Thread %08x" % tid

  return True

# invoked from GCA
def thread_regs_read(cpu, tid):
  if not thread_isalive(cpu, tid) or thread_getcurrent(cpu) == tid:
    regs = gca.target_regs_read(cpu)
  else:
    regs = target_os.get_thread_registers(cpu, tid)

  return regs

# invoked from GCA
def thread_regs_write(cpu, tid, regs):
  if not thread_isalive(cpu, tid):
    return False
```

```
  ret = 0
  if thread_getcurrent(cpu) == tid:
    ret = gca.target_regs_write(cpu, regs)
  else:
    ret = target_os.set_thread_registers(cpu, tid, regs)

  if ret > 0:
    return True

  return False

# invoked from GCA
def thread_mem_read(cpu, tid, addr, length):
  if not thread_isalive(cpu, tid):
    print "Thread %d is no longer alive" % tid

  return gca.target_memory_read(cpu, addr, length)


# invoked from GCA
# TODO: check cpu/tid is correct
def thread_mem_write(cpu, tid, addr, data):
  if not thread_isalive(cpu, tid):
    print "Thread %d is no longer alive" % tid
    return False

  ret = 0

  if thread_getcurrent(cpu) == tid:
    ret = gca.target_memory_write(cpu, addr, data)
  else:
    ret = mpatch.store(cpu, tid, addr, data)

  if ret > 0:
    return True;
  return False

# thread alive - gdb Txx cmd
def thread_isalive(cpu, t_id):
  threadlist_update(cpu)

  return t_id in thread_list

# current thread - gdb qC cmd
# invoked from GCA
def thread_getcurrent(cpu):
```

```
  return target_os.get_current_thread(cpu)

# extra thread info - gdb qThreadExtraInfo cmd
# invoked from GCA
def thread_getinfo(cpu, t_id):
  s = "%x" % (t_id)

  if t_id in thread_list:
    t = thread_list[t_id]
    s = "%s %04x" % (t["name"], t["current_state"] & 0xffff)

  return s

def threadlist_update(cpu):
  global thread_list

  thread_list = target_os.get_threads(cpu)

  return 1

# thread list interface - gdb q[fs]ThreadInfo cmd
# invoked from GCA
def threadlist_create(cpu):
  global first_cpu

  if first_cpu == 0:
    first_cpu = cpu

  threadlist_update(cpu)

  threads = []
  for t in thread_list:
    threads.append(t)

  threadlist_getnext.tlist = threads
  return 1

# invoked from GCA
def threadlist_count(cpu):
  if not hasattr(threadlist_getnext, 'tlist'):
    return 0

  return len(threadlist_getnext.tlist)

# invoked from GCA
def threadlist_getnext(cpu):
```

```
  if not hasattr(threadlist_getnext, 'tlist'):
    return 0

  out = 0
  if threadlist_count(cpu) > 0:
    out = threadlist_getnext.tlist.pop()

  return out


#symbol storage
# invoked from GCA
def symbol_getunknown():
  return symbol_storage.get_unknown()

# invoked from GCA
def symbol_add(symbol, value):
  symbol_storage.add(symbol, value)

def monitor_command(command, arg):
  if first_cpu == 0:
    return ""

  if command == "print":
    if arg == "objects":
      target_os.print_objects(first_cpu)

  return ""
```

Bibliography

[1] Kernel Based Virtual Machine, http://www.linux-kvm.org

[2] QEMU - a generic and open source machine emulator and virtualizer, http://www.qemu.org/

[3] DWARF 4, DWARF Debugging Standards, http://www.dwarfstd.org

[4] OpenOCD, Free and Open On-Chip Debugging, In-System Programming and Boundary-Scan Testing, http://openocd.sourceforge.net/

[5] Debugging with GDB, Appendix E gdb Remote Serial Protocol, http://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Protocol.html#Remote-Protocol

[6] Intel 80386 Reference Programmer's Manual, Chapter 17 80386 Instruction Set, http://www.read.seas.harvard.edu/ kohler/class/aosref/i386/c17.htm

[7] Operating Systems Design and Implementation, Tandenbaum, Woodhull

[8] Josh Jackson, An Anti-Reverse Engineering Guide, http://www.codeproject.com/KB/security/AntiReverseEngineering.aspx

[9] RTEMS Operating System, http://www.rtems.com/

[10] Python Programming Language, http://www.python.org/

[11] The Linux Kernel Archives, http://www.kernel.org/

[12] Intel® 64 and IA-32 Architectures Developer's Manual: Combined Volumes, http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html