EDB: a GDB-Based Debugger for Ethos

BY

FERNANDO VISCA
Laurea, Università di Roma “La Sapienza”, Rome, Italy, 2011

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2014

Chicago, Illinois

Defense Committee:

Wenjing Rao, Chair and Advisor
Jon Solworth
Paolo Ernesto Prinetto, Politecnico di Torino
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 THE ETHOS OS</td>
<td>2</td>
</tr>
<tr>
<td>2.1 Rethinking OS interfaces</td>
<td>2</td>
</tr>
<tr>
<td>2.2 Ethos on Xen</td>
<td>3</td>
</tr>
<tr>
<td>2.3 Authentication</td>
<td>3</td>
</tr>
<tr>
<td>2.4 fork() and the debug portal file descriptor</td>
<td>7</td>
</tr>
<tr>
<td>3 GDB: THE GNU GENERAL DEBUGGER</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Conceptual overview of GDB</td>
<td>10</td>
</tr>
<tr>
<td>3.2 Command-line syntax</td>
<td>14</td>
</tr>
<tr>
<td>3.3 GDB commands</td>
<td>15</td>
</tr>
<tr>
<td>3.4 GDB Remote Serial Protocol (RSP)</td>
<td>16</td>
</tr>
<tr>
<td>3.4.1 Register- and memory- related commands</td>
<td>18</td>
</tr>
<tr>
<td>3.4.2 Program control commands</td>
<td>19</td>
</tr>
<tr>
<td>3.4.3 Other commands</td>
<td>21</td>
</tr>
<tr>
<td>3.5 Debugging session initiation</td>
<td>22</td>
</tr>
<tr>
<td>3.6 The relation between GDB and EDB</td>
<td>23</td>
</tr>
<tr>
<td>4 NETSTACKGO</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Introduction to netStackGo</td>
<td>29</td>
</tr>
<tr>
<td>4.2 An example ping application</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1 Asynchronous operation and events</td>
<td>31</td>
</tr>
<tr>
<td>4.2.2 Ping client</td>
<td>31</td>
</tr>
<tr>
<td>4.2.3 Ping server</td>
<td>32</td>
</tr>
<tr>
<td>5 THE DESIGN OF EDB</td>
<td>35</td>
</tr>
<tr>
<td>5.1 An in-depth analysis of the design</td>
<td>37</td>
</tr>
<tr>
<td>5.2 An EDB case scenario</td>
<td>39</td>
</tr>
<tr>
<td>6 THE IMPLEMENTATION</td>
<td>41</td>
</tr>
<tr>
<td>6.1 User-space to kernel RPC</td>
<td>41</td>
</tr>
<tr>
<td>6.2 “Attach” and breakpoint insertion</td>
<td>56</td>
</tr>
<tr>
<td>6.3 Remote packets support</td>
<td>63</td>
</tr>
<tr>
<td>6.4 The process proxy</td>
<td>63</td>
</tr>
<tr>
<td>6.5 The fork wrapper</td>
<td>68</td>
</tr>
<tr>
<td>6.6 The GDB proxy</td>
<td>69</td>
</tr>
<tr>
<td>CHAPTER</td>
<td>PAGE</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>6.7</td>
<td>71</td>
</tr>
<tr>
<td>CONCLUSIONS</td>
<td>73</td>
</tr>
<tr>
<td>CITED LITERATURE</td>
<td>75</td>
</tr>
<tr>
<td>VITA</td>
<td>77</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>AUTHORIZATION SYSTEM CALLS</td>
<td>4</td>
</tr>
<tr>
<td>II</td>
<td>GDB COMMAND-LINE OPTIONS</td>
<td>26</td>
</tr>
<tr>
<td>III</td>
<td>GDB MOST FREQUENTLY USED COMMANDS</td>
<td>27</td>
</tr>
<tr>
<td>IV</td>
<td>EDB: SUPPORTED PACKETS</td>
<td>64</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>FIGURE DESCRIPTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example of client/server interaction in Ethos.</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>RSP packet</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Debugging session initiation</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>NetStackGo usage</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>The design of EDB.</td>
<td>36</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
<td></td>
</tr>
<tr>
<td>ELF</td>
<td>Executable and Linkable Format</td>
<td></td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-Process Communication</td>
<td></td>
</tr>
<tr>
<td>LOC</td>
<td>Line Of Code</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
<td></td>
</tr>
<tr>
<td>PKC</td>
<td>Public-Key Cryptography</td>
<td></td>
</tr>
<tr>
<td>PKI</td>
<td>Public-Key Infrastructure</td>
<td></td>
</tr>
<tr>
<td>PL</td>
<td>Programming Language</td>
<td></td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
<td></td>
</tr>
<tr>
<td>RSP</td>
<td>Remote Serial Protocol</td>
<td></td>
</tr>
<tr>
<td>UIC</td>
<td>University of Illinois at Chicago</td>
<td></td>
</tr>
<tr>
<td>UID</td>
<td>User ID</td>
<td></td>
</tr>
<tr>
<td>UUID</td>
<td>Universally Unique ID</td>
<td></td>
</tr>
<tr>
<td>VMM</td>
<td>Virtual Machine Monitor</td>
<td></td>
</tr>
<tr>
<td>VP</td>
<td>Virtual Process</td>
<td></td>
</tr>
</tbody>
</table>
SUMMARY

*Ethos* is a novel, security oriented operating system under development at the “Ethos lab”, University of Illinois at Chicago (UIC). Its intended goal is “to make it far easier to write applications which are robust against attack. It is a mammoth undertaking. It involves architecture, software layering, OS design and implementation, and programming language porting”\(^1\).

As a young OS Ethos lacks many of the user applications commonly installed on other systems. It faces the so called *application trap*: there are no applications developed for it because it has no users, and it has no users because there are no applications. One of the troubles a developer will encounter creating an application for Ethos is the lack of a *debugger* (commonly available on any modern OS, obviously).

In what follows, I shall report on how I designed and implemented a user applications debugger for Ethos (supporting the Intel\textsuperscript{®} x86 32 and 64 bits architectures), based on GDB, the GNU General Debugger. The name of this application is EDB.

The interesting part of this work is not only the development of the debugger making use of the *novel user interface* provided by Ethos, but the approach used to minimize development effort and the peculiar aspects of developing a debugger for a secure OS, with its implications. In fact, I shall also highlight which are the key differences between a user applications debugger for a “traditional” OS and for Ethos.

\(^1\)www.ethos-os.org
CHAPTER 1

INTRODUCTION

The objective of this work is that of reporting how I developed a user-space debugger, based on GDB, for the Ethos OS—EDB.

In the next chapters, I am going to introduce the reader to Ethos culture and structure with a particular emphasis on those development tools, their semantics and the kernel internals that more influenced the development of EDB—especially taking into consideration the fact that Ethos is a secure-oriented OS and as such has stricter requirements on authentication and authorization with respect to “traditional” OSs, even when it comes to debugging.

After introducing Ethos, I am going to introduce GDB, the GNU General Debugger. While introducing GDB, I will focus mainly on its Remote Serial Protocol (RSP) which is used by EDB to communicate with GDB remotely.

After introducing GDB, a brief introduction of netStackGo, Ethos’s native networking protocol (MinimaLT, see (1)) port to Linux. netStackGo is used to enable communication between the Xen Dom0 OS (Linux, from where GDB is run) to Ethos (where a software layer manages remote debugging through the RSP and acts as a middle layer between the remote GDB proxy running on Linux and the Ethos kernel).

Lastly, after the reader will have a clearer idea of the whole picture, I will explain the design choices that guided the development of EDB and the motivations behind them, together with a summary of the most relevant aspects of the implementation.
CHAPTER 2

THE ETHOS OS

2.1 Rethinking OS interfaces

The objective of the Ethos project is that of creating a new clean-slate OS design to ease writing and configuring robust applications. This goal is achieved—mainly—through an innovative system call semantics which facilitates—if not eliminates—the risk of security pitfalls.

Writing secure application can be a daunting, oftentimes unattainable task due to:

- Complexity and poor composition of existing system APIs
- Need for a deep knowledge of security threats and how to protect from them

In particular, without proper semantics, developing huge software systems would inevitably expose to security pitfalls. Moreover, current system APIs leave the implementation of many security aspects to the application (e.g. encryption, authentication, authorization\footnote{For example, \texttt{connect} and \texttt{accept} on POSIX leave encryption and user authentication to the application.}).

Ethos improves on interface semantics through better naming conventions, aliases avoidance and higher abstraction levels using types\footnote{}\footnote{\texttt{}}. The system calls set is minimal and so is complexity. Security properties are universally guaranteed—encryption, cryptography and authorization are all managed at kernel level and Ethos provides them transparently to the
programmer through its interface \(^1\). In Section 2.3 I will introduce some of the Ethos system calls and their use to establish a network connection.

### 2.2 Ethos on Xen

Over time, programming languages (PLs) have become more abstract, adding features such as *type safety*, *garbage collection* and improved *modularization*. On the other hand, operating systems have evolved much more slowly (3) due to backward compatibility issues and—in big part—to the *intrinsic complexity* of their realization.

To relieve the developers from the burden of more mundane portions of OS construction and allow them to focus on providing new interfaces, Ethos was built on top of the Xen virtual machine (4). The advantages to targeting a specific Virtual Machine Monitor (VMM) are debugging support, profiling support, device support and backward compatibility (5).

Many components of OSs require huge effort to develop from scratch, even for commercial systems. The approach adopted to create Ethos made it possible to minimize this effort. Thus, for some tasks Ethos relies on services provided by Xen or the Dom0 OS (Linux) as in the case of *device drivers* or the *filesystem*.

### 2.3 Authentication

An in-depth discussion on the Ethos authentication model can be found in (6) and (7). In what follows I am going to talk on how authentication is designed and used in Ethos in much

---

\(^1\)The programmer can thus concentrate on building better and more functional applications, because security is taken care of at a lower level.
broader terms. My objective is to explain the authentication mechanism to the reader so that he will more straightforwardly understand the design choices behind EDB (see Chapter 5).

The authentication facilities are embedded in Ethos and provided at the kernel level. A list of system calls involved in the authentication process is provided in Table I. We are interested mainly in the network authentication properties. Ethos networking and local Inter-Process Communication (IPC) use the same set of system calls: advertise, ipc and import. All network communications are encrypted and protected against tampering by cryptographic checksum.

<table>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTHORIZATION SYSTEM CALLS</td>
</tr>
<tr>
<td>Category</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Process management</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Local authorization</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Network authorization</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Ethos authentication mechanisms are centered around its virtual processes (VPs)—per-user on-demand processes. A VP is invoked by sending it a tuple of file descriptors. The system calls directly involved in the management of VPs are:

- \( \text{fdSend}(\text{fd[]}, u, \text{program}) \)

- \( \text{fds} \leftarrow \text{fdReceive}() \)

Basically, a program (the distributor) can send a tuple of file descriptors \( \text{fd[]} \) to the virtual process \( \text{program} \) belonging to user \( u \)—thus invoking the creation of process \( \text{program} \), if necessary. The virtual process receives incoming file descriptors with the \( \text{fdSend} \).

A typical usage of VPs (which is similar to the one used for EDB) is that of remote-user virtual processes. In this scenario, the distributor advertises a specific service and then accepts incoming network connections from users connected remotely. The distributor then identifies the user as, let us say, \( u \) and eventually sends (using \( \text{fdSend} \)) the imported network file descriptor to the VP belonging to \( u \). Such VP will ultimately take care of interacting with user \( u \) remotely.

In 2.3 an example of a client making a connection to a distributor is illustrated\(^1\). The remote client requests a connection to the distributor using the \text{ipc} system call on a previously advertised service. The distributor invokes \text{import}, waits and, once it receives the \text{ipc} call, it

\(^1\)Picture taken from (6).
accepts the connection \textit{if and only if} the remote client is authenticated and authorized to access that service\footnote{Again, authentication and authorization are taken care of inside of the kernel, so the programmer does not need to care about it. All the developer needs is \texttt{ipc/import}, thus code complexity and probability of error are reduced and so are security pitfalls.}.

After a connection is established, the distributor \textit{sends the IPC file descriptor} to the relevant user’s VP through the \texttt{fdSend)—as described above. Eventually, the VP will interact with the remote client using the \texttt{read} and \texttt{write} system calls on the provided file descriptor (similarly to Linux).
Ethos uses a Local Authentication Service (LAS)—based on password—for physical login. For remote login a Remote Authentication Service (RAS) is used. Ethos remote authentication is based on Public-Key Cryptography (PKC) so users can create their own key pairs, and public keys are guaranteed unique. Thus, public keys are Universally Unique IDs (UUIDs) that can serve as UID, even if the real world identity is not known (8; 9; 10). To associate a name and thus a real-world identity with a UID, a custom Internet-scale PKI is being developed for Ethos, which is introduced in (11).

2.4 fork() and the debug portal file descriptor

The fork() system call, similarly to Linux, forks the current process creating a child process having the same memory content (shared using a copy-on-write policy) and separate virtual memory address space. Father and child processes are identified using the returned process ID in the same way as one would do on Linux. The signature of the fork() system call is:

\[
\text{status, terminatePortalFd, debugPortalFd} \leftarrow \text{fork(level)}
\]

The fork returns a status and to the parent a terminate portal and a debug portal (which are basically two file descriptors). A read may be called on the terminate portal. This will result in the calling process blocking until the child process exits. The value read is the child process exit status. In addition, the process terminate portal is sent to the process terminate virtual process\(^1\).

\(^1\)level is the process groups of the child to change. Process groups level...MaxProcessGroups1 of the child is set to the childs process ID.
When a process is created, two portals are returned to the parent—two interfaces to manage the process itself. These two portals are the terminate portal and the debug portal. The terminate portal allows performance information to be collected, and the process to be terminated. The terminate portal file descriptor is sent by the Ethos kernel to the terminate portal virtual process. (Note that different users, and the system as a whole, each have their own terminate portal virtual process). The terminate portal thus allows to:

- Get execution statistics
- Kill a process
- Determine whether the process still exists
- Get process group information

Process groups therefore are specified at fork time. The terminate portal virtual process will manage the processes of a user. It is responsible for providing kill semantics and ps semantics to other processes. The debug portal enables interaction with the process to take place over an IPC. The respective portal semantics are defined by the RPC interface supported by that IPC. All IPC endpoints are always named.

I have developed Ethos’ debugging facilities as RPCs to the debug portal file descriptor (as I will describe in Chapter 6). I have defined an interface supported by the IPC channel that basically consists of two data structures (GdbProxyCall and GdbProxyReply) designed to represent a debugging request and relative reply. From user space, it is possible to generate an
encoder and decoder for the RPC based on the debug portal file descriptor. By doing so, the request will be forwarded directly inside the kernel.

Enabling RPCs from user-space to the kernel, encoding on/decoding from the debug portal file descriptor associated with the “debuggee” process has been one of the major blocks of EDB development, allowing for debugging requests to be performed and responses to be collected.
CHAPTER 3

GDB: THE GNU GENERAL DEBUGGER

GDB, the GNU debugger, is the standard debugger for the GNU operating system. However, its portability and flexibility have made it available on many UNIX-like systems and for different programming languages. The definitive reference for GDB is its official documentation (see (12) and (13)). In what follows, I am going to introduce the reader to those aspects of GDB that are important for the remaining of the dissertation—most importantly the GDB Remote Serial Protocol (RSP).

[The content of sections 3.1, 3.2 and 3.3 has been adapted from Chapter 17 of (14)].

3.1 Conceptual overview of GDB

A debugger is a program that lets you run a second program, which we will call the "debuggee." The debugger lets you examine and change the state of the debuggee, and control its execution. In particular, you can single-step the program, executing one statement or instruction at a time, in order to watch the program's behavior.

Debuggers come in two flavors: instruction-level debuggers, which work at the level of machine instructions, and source-level debuggers, which operate in terms of your program's source code and programming language. The latter are considerably easier to use, and usually can do machine-level debugging if necessary. GDB is a source level debugger; it is probably
the most widely applicable debugger (portable to the largest number of architectures) of any current debugger.

GDB itself provides two user interfaces: the traditional command-line interface (CLI) and a text user interface (TUI). The latter is meant for regular terminals or terminal emulators, dividing the screen into separate “windows” for the display of source code, register values, and so on.

GDB provides support for debugging programs written in C, C++, Objective C, Java\(^1\), and Fortran. It provides partial support for Modula-2 programs compiled with the GNU Modula-2 compiler and for Ada programs compiled with the GNU Ada Translator, GNAT. GDB provides some minimal support for debugging Pascal programs. The Chill language is no longer supported.

When working with C++ and Objective C, GDB provides name demangling. C++ and Objective C encode overloaded procedure names into a unique “mangled” name that represents the procedures return type, argument types, and class membership. This ensures so-called type-safe linkage. There are different methods for name mangling, thus GDB allows you to select among a set of supported methods, besides just automatically demangling names in displays.

If your program is compiled with GCC (the GNU Compiler Collection), using the \(-g3\) and \(-gdwarf-2\) options, GDB understands references to C preprocessor macros. This is particularly

\(^1\)GDB can only debug Java programs that have been compiled to native machine code with GJC, the GNU Java compiler (part of GCC, the GNU Compiler Collection).
helpful for code using macros to simplify complicated struct and union members. GDB itself also has partial support for expanding preprocessor macros, with more support planned.

GDB allows you to specify several different kinds of files when doing debugging:

- The *exec file* is the executable program to be debugged—i.e., your program.

- The optional *core file* is a memory dump generated by the program when it dies; this is used, together with the exec file, for post-mortem debugging. Core files are usually named `core` on commercial Unix systems. On BSD systems, they are named `program.core`. On GNU/Linux systems, they are named `core.PID`, where PID represents the process ID number. This lets you keep multiple core dumps, if necessary.

- The *symbol file* is a separate file from which GDB can read symbol information: information describing variable names, types, sizes, and locations in the executable file. GDB, not the compiler, creates these files if necessary. Symbol files are rather esoteric; they’re not necessary for run-of-the-mill debugging.

There are different ways to stop your program:

- A *breakpoint* specifies that execution should stop at a particular source code location.

- A *watchpoint* indicates that execution should stop when a particular memory location changes value. The location can be specified either as a regular variable name or via an expression (such as one involving pointers). If hardware assistance for watchpoints is available, GDB uses it, making the cost of using watchpoints small. If it is not available, GDB uses virtual memory techniques, if possible, to implement watchpoints. This also
keeps the cost down. Otherwise, GDB implements watchpoints in software by single-stepping the program (executing one instruction at a time).

- A catchpoint specifies that execution should stop when a particular event occurs.

The GDB documentation and command set often use the word *breakpoint* as a generic term to mean all three kinds of program stoppers. In particular, you use the same commands to enable, disable, and remove all three.

GDB applies different statuses to breakpoints (and watchpoints and catchpoints). They may be *enabled*, which means that the program stops when the breakpoint is hit (or *fires*), *disabled*, which means that GDB keeps track of them but that they don’t affect execution, or *deleted*, which means that GDB forgets about them completely. As a special case, breakpoints can be enabled only once. Such a breakpoint stops execution when it is encountered, then becomes disabled (but not forgotten).

Breakpoints may have conditions associated with them. When execution reaches the breakpoint, GDB checks the condition, stopping the program only if the condition is true.

Breakpoints may also have an *ignore count*, which is a count of how many times GDB should ignore the breakpoint when its reached. As long as a breakpoints ignore count is nonzero, GDB does not bother checking any condition associated with the breakpoint.

Perhaps the most fundamental concept for working with GDB is that of the *frame*. This is short for *stack frame*, a term from the compiler field. A stack frame is the collection of information needed for each separate function invocation. It contains the functions parameters and local variables, as well as *linkage* information indicating where return values should be
placed and the location the function should return to. GDB assigns numbers to frames, starting at 0 and going up. Frame 0 is the innermost frame, i.e., the function most recently called.

GDB uses the readline library, as does the Bash shell, to provide command history, command completion, and interactive editing of the command line. Both Emacs and vi style editing commands are available.

Finally, GDB has many features of a programming language. You can define your own variables and apply common programming language operators to them. You can also define your own commands. Additionally, you can define special hook commands, user-defined commands that GDB executes before or after running a built-in command. You can also create while loops and test conditions with if ... else ... end.

GDB is typically used to debug programs on the same machine (host) on which its running. GDB can also be configured for cross-debugging, i.e., controlling a remote debuggee with a possibly different machine architecture (the target). Remote targets are usually connected to the host via a serial port or a network connection.

3.2 Command-line syntax

GDB is invoked as follows:

- `gdb [options] [executable [corefile-or-PID]]`
- `gdb [options] --args executable [program args ...]`

GDB has both traditional short options and GNU-style long options. Long options may start with either one or two hyphens. In Table II there is an overview of the command-line options.
3.3 GDB commands

In Table III a list of the most frequently used GDB commands is reported. Although the list is self-sufficient for many users, for a complete list of commands it is possible to read (12), (13) and (14).

In Listing 3.1 I reported the screenshot from an example GDB debugging session of a remote target (see (15)). Further information about remote target debugging operations and the RSP can be found in 3.4.

Listing 3.1. A GDB remote debugging session screenshot

```bash
localhost$ sh–hitachi–hms–gdb a.out
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
(gdb) target remote /dev/ttyS0
(gdb) load
Loading section .text, size 0x1280 vma 0x1000
Loading section .data, size 0x760 vma 0x2280
Loading section .stack, size 0x10 vma 0x3000
Start address 0x1000
Transfer rate: 53120 bits in <1 sec.
(gdb) b main
Breakpoint 1 at 0x8048476: file test.c, line 5.
```
(gdb) continue

Breakpoint 1, main () at test.c:5

5 for ( i = 0; i < 10; i++ ) {

(gdb) display j

1: j = 1074136126

(gdb) step

6 j = i * 2 + 1;

1: j = 1074136126

(gdb) step

5 for ( i = 0; i < 10; i++ ) {

1: j = 1

(gdb) quit

3.4 **GDB Remote Serial Protocol (RSP)**

*[The content of this section has been adapted from (16); all the pictures are from (17)].*

The GDB Remote Serial Protocol (RSP) is a simple, ASCII message-based protocol suitable for use on serial lines, local area networks, or just about any other communications medium that can support at least half-duplex data exchange.

RSP packets begin with a dollar sign ($), followed by one or more ASCII bytes that make up the message being sent, and end with a pound sign (#) and two ASCII hex characters representing the messages checksum. For example, the following is a complete RSP packet (see
The receiver of the packet responds immediately with either a “+” or a “-” to indicate that the message was received either intact or in error, respectively. A typical transaction involves GDB issuing a command to a debugging target, which then responds with data, a simple acknowledgement, or a target-specific error code. If the latter is returned, GDB will report the code to the user and halt whatever activity is currently in progress.

The `console output` message, which debugging targets use to print text on the GDB console, is the lone exception to the typical command-response sequence. Except when another command is already in progress, this message can be sent from the debugging stub to GDB at any time¹.

¹`console output` is not implemented in EDB and is however not mandatory.
3.4.1 Register- and memory- related commands

Read registers ("g") Example: $g#67
The debugger will issue this command whenever it needs to know everything about the debugging targets current register state. An example target response would be:
+ $123456789abcdef0...#xx
(Related 0 is 0x12345678, register 1 is 0x9abcdef0, and so on). The response is an ordered stream of bytes representing register data ordered per the definition in the targets macro file, gdb/config/<arch>/tm-<arch>.h (for example, gdb/config/sh/tm-sh.h for the Hitachi SH).

Write registers ("G") Example: $G123456789abcdef0...#xx
(Set register 0 to 0x12345678, register 1 to 0x9abcdef0, and so on). This message is the complement to the read registers command. With this command, GDB supplies an ordered stream of bytes representing data to be stored in the target processors registers immediately before program execution resumes. An example target response:
+ $OK#9a

Write register N ("P") Example: $P10=0040149c#b3
(Set register 16 to the value 0x40149c). When it wants to set the value of only one or two registers, GDB sends this command instead of sending a complete register set to the debugging target. The register numbering is the same as that used in the read registers and write registers commands. An example target response:
+ $OK#9a
Read memory (“m”) Example: $m4015bc,2#5a
(Read two bytes, starting at address 0x4015bc). A read memory command is sent by GDB to determine the values of local and global variables, the value of an opcode about to be replaced by a breakpoint instruction, and any other kind of information the user requests. The debugger generally is aware of any endian issues present in the debugging target, so the target need only return the result as a simple stream of bytes; GDB will reformat them as appropriate. Debugging stubs on targets that are sensitive to data widths should optimize the implementation of the write memory and read memory commands as the target architecture dictates. An example target response:
+ $2f86#06

Write memory (“M”) Example: $M4015cc,2:c320#6d
(Write the value 0xc320 to address 0x4015cc.) This command is the complement to the read memory command. An example target response:
+ $0K#9a

3.4.2 Program control commands

Program control commands are messages that gdb uses to control the behavior of the application being debugged. As such, these commands are somewhat more complicated to implement than the more basic register- and memory-related commands I have already covered.

Get last signal (“?”) Example: $?#3f
This command is used to find out how the target reached its current state. The response is the same as the “Last signal” response documented below.
**Step** ("s") Example: $s#73

When it wants the target to execute exactly one assembly language instruction, GDB issues a `step` command to the debugging target. The debugger sends this command when the user types `stept` or `step` at the GDB console. An example target response follows the `continue` command description.

**Continue** ("c") Example: $c#63

A `continue` command is issued when GDB releases the application to run at full speed, as happens when the user enters a continue command at the GDB console. An example target response follows.

*Responses to the step and continue commands.* A debugging stub does not immediately respond to the `step` or `continue` commands, other than to send the "+" that signifies proper reception of the packet. Instead, the stub provides a response when the next breakpoint is reached, the requested instruction has been executed (in the case of the `step` command), an exception occurs, or the application exits.

There are two ways to respond to these commands: a brief “last signal” response, or a more useful “expedited response”.

**“Last signal” response** ("S") Example: $S05#b8

This is the minimum reply to the last signal, `step`, and `continue` commands. The “05” in the
response can be any one of the signal values used in the standard POSIX `signal()` function call. For example, “5” is a breakpoint exception, “10” is a bus error, and so forth.

**Expedited response (“T”)** Example: $T0510:1238;F:FFE0...#xx

This message combines the information in a “last signal” response (the “05” in the example message) with key register values that GDB may be immediately interested in. Designed to improve GDBs performance during code stepping, this message allows GDB to avoid a read registers request if the values it needs (the targets program counter and status register, generally) are included in this message. Registers are identified by the same numbering scheme used in the read registers and write registers commands; in the example provided, the value of register 16 (10 hex) is 0x1238, and register 15 (F hex) contains 0xffe0.

### 3.4.3 Other commands

**Console output (“O”) (optional)** Example: $O48656c6c6f2c20776f726c64210a#55

(Prints “Hello, world!\n” on the GDB console). This command allows a debugging stub to send a text message to the GDB console. The text to be displayed is sent in its hexadecimal byte equivalent (‘H’ == 0x48) and GDB will queue successive messages until it sees a newline (‘\n’ == 0x0a) character. This message always originates in the debugging target; GDB never sends a console output message to the debugging target.

**Empty response (“”)**

If a debugging stub encounters a command it does not support or understand, it should return

---

1 As Ethos does not—for design choice—implement signals, a $S05#b8 response packet is always returned by convention by EDB to indicate the presence of a breakpoint or other error condition.
an empty response. This allows GDB to select an alternate command if one is available.

Example: `<an unrecognized command>`

Target response: +#$00

**Error response ("E")**

When a debugging stub encounters an error during command processing, it should send an error response back to GDB. Bus errors and/or illegal addresses during memory operations are examples of commands which may generate an error response from a debugging stub.

Example: `<a command that produces an error>`

Target reponse: +$E01#xx

There are not any predefined error codes in GDB; when GDB receives an error message, it prints it on the console and halts the operation in progress.

### 3.5 Debugging session initiation

In 3.5 an example message exchange sequence used by GDB to initiate the debugging session with the remote target (`target remote` command) is reported.

For all the packets listed a default or dummy response is generated either because that functionality is not supported at all by Ethos or because EDB still does not implement it. For example, the `Hg-1` packet is used to tune settings related to multi-threaded programs debugging, but since Ethos does not support multi-threading at all a dummy reply will be generated (OK).

The only parameter that is utilized is the `PacketSize` parameter contained in the response to the `qSupported` packet that specifies the maximum supported length of a packet. This parameter is important, for example, when exchanging the content of an N bytes memory
location. If the length of the resulting packet is too long, it will need to be split into a sequence of multiple packets.\footnote{However, the size selected is big enough such that this latter case never holds.}

### 3.6 The relation between GDB and EDB

EDB is not properly a port of GDB. The kernel side of EDB (functions such as register retrieval or memory access or breakpoint insertion) have similar semantics and took inspiration from the functions in the i386 GDB stub (see (12) here https://sourceware.org/gdb/current/onlinedocs/gdb/Remote-Stub.html#Remote-Stub) but are rather tailored to be used on an OS (Ethos) than an embedded system.

Moreover, GDB does not interact directly with the kernel/embedded system as in the case of the GDB stub but rather with an intermediate software layer that:

- Shifts most of the “dirty” work from kernel space to user space to maintain small, clean and fully tested kernel routines
- Provides for authentication and authorization properties required to perform debugging (is debugging of process $P$ allowed for user $U$?)

The RSP is intended to be used mainly in \textit{embedded systems} where the debugging target, for computational or memory constraints, cannot execute GDB locally. The RSP allows the developer to connect such systems remotely to a computer (via a \textit{serial port} or \textit{over TCP/IP})
and debug applications remotely. To enable remote debugging the remote target has to support
at least the subset of packages listed in the stubs.

I decided to add support for the RSP in Ethos to:

1. Keep the kernel side of the application small and easy to read and test
2. Reduce complexity with respect to a complete application port
3. Overcome Ethos limitations (like its rudimental shell)

It is worth mentioning that packet parsing for the RSP is implemented at the Dom0 Linux level
(see Chapter 5). This means that formally it is the remote GDB proxy that supports the RSP,
not Ethos.

Ethos, instead, supports debugging requests sent via netStackGo to the process proxy
(see Chapter 5) using a RPC interface (namely, it receives and handles a GdbProxyCall data
structure that defines type of operation and parameters and replies with a GdbProxyReply
data structure). The reason for this is that we want to keep “dirty” operations like string
package manipulation completely outside of Ethos that instead supports a clean, well defined
and security features-rich interface.

In conclusion, EDB is based on GDB since the kernel side of EDB is an ex-novo implementa-
tion of basic debugging facilities (breakpoint insertion, register manipulation etc.) but all
the rest is performed by GDB and a software layer between GDB and Ethos. What “all the
rest” means is that to debug a program running on Ethos we launch GDB from a remote proxy
(e.g. Linux running on Dom0) and GDB will possibly parse—locally—the executable elf file
of the debuggee. After that, we debug the program running on Ethos (the debuggee) using the RSP. Finally, a software interface between GDB and the Ethos kernel will take care of serving debugging requests from GDB and implementing the communication channel between GDB and Ethos (using netStackGo, Chapter 4).
TABLE II

<table>
<thead>
<tr>
<th>GDB COMMAND-LINE OPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-line option</td>
</tr>
<tr>
<td>--args</td>
</tr>
<tr>
<td>--async, --noasync</td>
</tr>
<tr>
<td>-b baudrate, --baud baudrate</td>
</tr>
<tr>
<td>--batch</td>
</tr>
<tr>
<td>--cd dir</td>
</tr>
<tr>
<td>-c file, --core file</td>
</tr>
<tr>
<td>-d dir, --directory dir</td>
</tr>
<tr>
<td>-e file, --exec file</td>
</tr>
<tr>
<td>-f, --fullname</td>
</tr>
<tr>
<td>--help</td>
</tr>
<tr>
<td>--interpreter interp</td>
</tr>
<tr>
<td>-n, --nx</td>
</tr>
<tr>
<td>-nw, --nowindows</td>
</tr>
<tr>
<td>-p pidnum, -c pidnum, --pid pidnum</td>
</tr>
<tr>
<td>-q, --quiet, --silent</td>
</tr>
<tr>
<td>-r, --readnow</td>
</tr>
<tr>
<td>-s file, --symbols file</td>
</tr>
<tr>
<td>--se file</td>
</tr>
<tr>
<td>--statistics</td>
</tr>
<tr>
<td>-t device, --tty device</td>
</tr>
<tr>
<td>--tui</td>
</tr>
<tr>
<td>-x file, --command file</td>
</tr>
<tr>
<td>--version</td>
</tr>
<tr>
<td>-w, --windows</td>
</tr>
<tr>
<td>--write</td>
</tr>
</tbody>
</table>
TABLE III

GDB MOST FREQUENTLY USED COMMANDS

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>backtrace</td>
<td>Show call trace</td>
<td>ba</td>
</tr>
<tr>
<td>break</td>
<td>Set breakpoint at routine entry or at line number</td>
<td>b main</td>
</tr>
<tr>
<td>continue</td>
<td>Continue from breakpoint</td>
<td>cont</td>
</tr>
<tr>
<td>delete</td>
<td>Remove breakpoint</td>
<td>d 3</td>
</tr>
<tr>
<td>finish</td>
<td>Step until end of routine</td>
<td>fin</td>
</tr>
<tr>
<td>info breakpoints</td>
<td>List current breakpoints</td>
<td>i br</td>
</tr>
<tr>
<td>next</td>
<td>Step to next statement and over routine calls</td>
<td>ne</td>
</tr>
<tr>
<td>print</td>
<td>Print expression</td>
<td>print 1.0/3.0</td>
</tr>
<tr>
<td>run</td>
<td>(Re)run program, optionally with arguments</td>
<td>ru</td>
</tr>
<tr>
<td>step</td>
<td>Step to next statement and into routines</td>
<td>s</td>
</tr>
<tr>
<td>x</td>
<td>Examine memory</td>
<td>x/s *environ</td>
</tr>
<tr>
<td>until</td>
<td>Continue execution until reaching a source line</td>
<td>until 2367</td>
</tr>
</tbody>
</table>
Figure 3. Debugging session initiation

GDB (RSP Client)          Target (RSP Server)

qSupported
PacketSize=119
?
S05
Hc-1
OK
qC
(empty)
qOffsets
Text=0;Data=0;Bss=0;
Hg-1
OK

Report packet size supported
Report we stopped due to signal 5 (TRAP exception)
Future step/continue operations on all threads
Current thread is -1
Report offsets to be used when loading code
All other future operations should apply to all threads
Report all general register values
Offer to provide symbol values (none required)
CHAPTER 4

NETSTACKGO

4.1 Introduction to netStackGo

netStackGo is a port of Ethoss native networking protocol, MinimaLT (see (1)) to Linux. MinimaLT provides encrypted, authenticated, and authorized networking. It interfaces both to sayI (see (11)) and to Linux authorization hooks.

netStackGo enables MinimaLT to be used between Linux Go applications and either Linux or Ethos Go applications. Typical netStackGo usage is shown in 4.1. netStackGo consists of two Linux components:

- The netStackGo daemon, nsgd (light yellow)
- the netStackGo package (in Go) (light green), enabling applications to interface with the nsgd

As can be seen from the figure, a single nsgd can provide tunnels to multiple hosts. nsgd concurrently supports an arbitrary number of applications and can connect to an arbitrary number of remote hosts.

netStackGo is used in EDB to allow the remote GDB proxy and the Ethos process proxy to communicate (see chapter 5).
4.2 An example ping application

In this section, two examples of ping programs are introduced. Before talking about the examples, I will introduce the asynchronous operation mechanism and events. EDB communicates with the remote GDB proxy in a similar way as the ping example provided. In particular, two new Ethos data types are defined for communication purposes, GdbProxyCall and GdbProxyReply. Finally, all networking operations executed in EDB are synchronous and follow a determined order—i.e. the remote GDB proxy is the only side of the communication channel who initiates the communication with a request and the Ethos process proxy only waits for requests, and replies.
4.2.1 Asynchronous operation and events

Operations provided by netStackGo are asynchronous. This means one can issue an operation, do something else while waiting it to complete, then check the status of it and retrieve any result. An event is immediately returned upon issuing an asynchronous operation, and the BlockAndRetire system call is used to get the results. Three objects are returned—data, connection and error—but not all of them are applicable for all operations.

- data, connection, error := event.BlockAndRetire()

4.2.2 Ping client

The first example (see Listing 4.1 is a ping client that uses the synchronous programming interfaces of netStackGo. The sequence of instructions is summarized below.

1. Ipc to create a connection, specifying the remote host name, service name ("ping"), and name of service instance (again, "ping").

2. BlockAndRetire to get the result of Ipc—i.e. the new connection and an error, if any.

3. Create an encoder and decoder, which can encode and decode string types (GdbProxyCall or GdbProxyReply types respectively in the case of EDB). They will ultimately invoke the Read or Write system calls to complete their job.

4. Encode a string containing the message to be sent to the remote host, and call Flush to flush the buffer.
5. Decode a string received from the remote host, and print it on screen (in the case of EDB I will decode a GdbProxyCall type and execute the relative debugging operation issuing a RPC to the kernel.

6. Close the connection.

4.2.3 Ping server

The second example (see Listing 4.2) is a ping server that uses the asynchronous programming interfaces of netStackGo. For simplicity, it only serves one ping.

1. Advertise the service “ping”, with same service instance name.

2. Import a new connection.

3. BlockAndRetire to get the result of Import—i.e. the new connection and an error if any.

4. IpcRead to read from the connection asynchronously.

5. BlockAndRetire to get the result of IpcRead.

6. IpcWrite to write on the connection asynchronously.

7. BlockAndRetire to get the result of IpcWrite.

8. Close the connection.
Listing 4.1. Ping client (synchronous)

generic main

import (  
  "ethos/nsg"  
  "fmt"
)

cfunc main() {  
event, err := nsg.Ipc ("hostname", "ping", "ping")  
if err != nil {  
  panic (err)
}

  , c, err := event.BlockAndRetire ()  
if err != nil {  
  panic (err)
}

en := nsg.NewEncoder (c)  
de := nsg.NewDecoder (c)

err = en.String ("Hello, from netStackGo!")  
if err != nil {  
  panic ("Error writing")
}

en.Flush ()  
fmt.Println ("Sent: Hello, from netStackGo!")

str, err := de.String ()  
if err != nil {  
  panic ("Error reading")
}

fmt.Println ("Received: ", *str)

c.Close ()
}
Listing 4.2. Ping server (asynchronous)

```go
package main

import (
    "ethos/nsg"
)

func main() {
    ad, err := nsg.Advertise("ping", "ping")
    if err != nil {
        panic(err)
    }

    event, err := ad.Import()
    if err != nil {
        panic(err)
    }

    , c, err := event.BlockAndRetire()
    if err != nil {
        panic(err)
    }

    event, err = c.IpcRead()
    if err != nil {
        panic(err)
    }

    data, _, err := event.BlockAndRetire()
    if err != nil {
        panic(err)
    }

    event, err = c.IpcWrite(data)
    if err != nil {
        panic(err)
    }

    , _, err = event.BlockAndRetire()
    if err != nil {
        panic(err)
    }

    c.Close()
}
```
In 5 there is a scheme of the design of EDB. The main reasons that led me to such a design (and that I will explain better throughout the chapter) are:

1. GDB should run “remotely” on another system (i.e. on Linux running on Dom0) and debug processes running on Ethos connecting to it and exploiting the RSP.

2. The user must be authenticated and the debug requests authorized properly, using Ethos’s built-in mechanisms for authorization and authentication.

3. The user must have access to its list of processes (ps command) and to that process’ debug portal file descriptor (FD)—see Section 2.4.

We want to run GDB remotely—as explained before—mainly to reduce the effort involved in porting GDB. However, there is also a second reason: we want to keep “dirty” operations like string packages manipulation as out as possible from Ethos that is instead intended to provide a well defined, robust and easy to understand interface.

After reading Chapter 2 and Chapter 4, it should be clear that RPCs and netStackGo embed Ethos’ authorization and authentication primitives and thus the second point is automatically satisfied—this is a demonstration of the flexibility and ease of use of Ethos interface, especially with respect to security features.
Figure 5. The design of EDB.
The `ps` command has been implemented specifically to print processes’ information in order for the user to not only retrieve information on its running processes, but also to pick any process available for debugging—identified by means of its PID—and request an attach to be performed to it.

The `gdbProxy` that will ultimately request the debugging operations to the kernel will also need to know the process’ debug portal FD. As explained in Section 2.4, this FD is only available to the parent process and I shall explain how I managed to solve such an issue.

5.1 An in-depth analysis of the design

The fulcrum of EDB is the “processProxy”. This per user virtual process:

- Maintains a list of debug portal and terminate portal FDs, together with further information on the user’s active processes—this is from where the `gdbProxy` will collect the debug portal FDs.
- Advertises two services:
  - `/services/debug/username`
  - `/services/processStatus/username`
- Forks the so called `gdbProxy`.

The `processProxy` receives and stores the debug portal and terminate portal file descriptors for each and every process belonging to user $U$ via the `FdReceive()` system call $^1$. After that,

$^1$The file descriptors are sent by a wrapper of `Fork()`, depicted in figure 5, via the `FdSend()` system call.
thanks to an ad-hoc implemented `GetProcessStatus()` system call, the `processProxy` can retrieve the PID of the process and eventually store it as well.

As I already said, the `processProxy` advertises two services:

1. `/services/debug/username`
2. `/services/processStatus/username`

The latter is used by the `ps Ethos` command, created for the purpose, to request an on-screen printout of the list of available processes\(^1\).

The former is used by the `remoteGdbProxy` to initiate a debugging session (again, this service is *per user*). The `remoteGdbProxy` will first issue an `Attach()` RPC to one of the PIDs previously listed with the `ps` command. After that, provided that the call is successful\(^2\), further debugging operations can be requested using the `Debug()` RPC. All parsing operations and interaction with GDB (over TCP/IP) will be performed by the `remoteGdbProxy`.

At this point I assume the reader will still have doubts about how EDB works, especially with regards to the role of the `gdbProxy`. In section 5.2 there is an example debugging session, also explaining step by step how to run GDB and debug a program running on Ethos with the `target remote` command.

\(^1\)It is worth to mention again that `processProxy` is a per user virtual process so when user \(U\) issues `ps` only his processes will be shown—this strengthen authorization requirements.

\(^2\)The process associated with the PID could have terminated or been attached to and be no longer available for debugging by the time we list it with `ps` and then issue the `Attach()` . For this reason, the `Attach()` *can still fail*. 
5.2 An EDB case scenario

Let us suppose we want to debug two processes belonging to user “Jon” \(^1\).

Firstly, we run Ethos and login with the “Jon” user credentials. “Jon” can now run processes (both user applications—e.g. a browser—and system services). As processes are created using the `Fork()` system call every time a process is run the `fork_wrapper()` function is issued. This function will first fork the process and eventually send via `FdSend()` its terminate and debug portals file descriptors to the `processProxy` virtual process belonging to “Jon”.

Secondly, we run the `ps` command and list the available processes. We pick up the PID of the process we want to debug. The PID is used just as an identifier for the process and it could as well be replaced by an increasing number assigned by `ps` or the name of the process\(^2\).

Thirdly, we run a `remoteGdbProxy` instance using the aforementioned PID as command line parameter. The `remoteGdbProxy` will communicate using `netStackGo` with “Jon”’s `processProxy` over the service `/services/debug/Jon`. (First of all, an `Attach()` will be issued and lastly, if the `Attach()` was successful, a new debugging session (using the `Debug()` RPC) will be initiated).

Finally, let us analyze how the `processProxy` manages the debug service. As soon as the `remoteGdbProxy` issues the `Attach()`, `processProxy` actually forks into `gdbProxy`. The imported file descriptor (associated with the `netStackGo` channel that connects to the `remoteGdbProxy`) \(^1\)

---

\(^1\) If it was two processes belonging to different users or multiple processes, it would have made no much difference and the steps are similar.

\(^2\) I have chosen to use the PID as the identifier because it is thus easier to unambiguously identify which process are we selecting.
is closed in processProxy and is used only in the gdbProxy—by inheritance—to manage that particular debugging session. From that moment on, processProxy will go back and wait for another instance of the remoteGdbProxy to perform an Attach() while the gdbProxy will manage the debugging session. This means that we will have $N$ gdbProxy instances, where $N$ is the number of processes being debugged on the system—for all users. $N_{Jon}$ of such instances will communicate with the remoteGdbProxy using the /services/debug/Jon service, where $N_{Jon}$ is the number of processes being debugged for user “Jon”.

When we request another connection on /services/debug/Jon from the remoteGdbProxy, the processProxy will import a new IPC file descriptor for it. A second gdbProxy is created and a new process is attached to and so on and so forth...
CHAPTER 6

THE IMPLEMENTATION

6.1 User-space to kernel RPC

The first step towards implementing EDB consisted in enabling the user to issue a RPC from user-space directly into the kernel. Before this work, RPC have been successfully used to communicate between user-space processes and inside of the kernel space, but establishing a communication channel from user-space to kernel was not yet attempted.

To understand how this is accomplished it is fundamental to understand the Ethos types notation and the concepts of types definition, encoding and decoding of a type. With a RPC we can issue a procedure to be executed remotely (in this case, from user-space to kernel, on the same machine) passing our parameter to it. When replying, the other end of the communication—in our case the kernel—will issue a reply RPC the other way around—from kernel to user-space—passing the result as parameter 1.

The RPC works by encoding the data in a data structure that encapsulates the values and their types in a predefined format. This encoded chunk of memory is ultimately written using the write() system call on a file descriptor (in our case it is a debugging portal file descriptor, but it could be an IPC file descriptor as well). The write() system call implementation will

1RPC return a null value for how they are implemented. To make it possible to return a value a reply RPC is defined that uses as parameter the data to return.
have a switch case that distinguishes the various types of file descriptor (IPC, file, debug portal, terminate portal...) since *for each type a different function* will take care of the internals of the write \(^1\). In case a debug portal file descriptor is used, the write logic will be taken care of by the \texttt{gdbWrite()} function.

The \texttt{gdbWrite()} function decodes the data written using the provided tools, parses the \texttt{GdbProxyCall} data structure, executes the specified debugging function and saves the return value (if any) in a \texttt{GdbProxyReply} variable in the *process data structure*.

In Listing 6.1 the source code for the \texttt{gdbWrite()} is reported (the \texttt{gdbRead()} instead is called when a reply RPC is requested and basically reads the reply from the debuggee process data structure, encodes it and sends it to user-space via the returned \texttt{Event}—see Listing 6.2).

```
Listing 6.1. gdbWrite()

// Manages writes to debug portal file descriptors
// (debug operation request from user)
// Parameters: packet written (contents) and write event created in the
// Write system call
Status

gdbWrite(String *contents, Event *writeEvent)
{
```

\(^1\)Similarly, in Linux we have that a file descriptor can represent a file, as well as a device.
EtnRpcHost *DHost;
EtnBufferDecoder *DDecoder;
EtnLength memSize = sizeof(GdbProxyCall) + sizeof(uint64_t) * 2;

DDecoder = etnBufferDecoderNew(contents->ptr, memSize);
DHost = etnRpcHostNew(EtnToValue(&EdbRpcServer, 0), NULL,
                        (EtnDecoder *)
                        DDecoder);

etnRpcHandle(DHost);

eventComplete(writeEvent);
return current->edbStatus;
}

Listing 6.2. gdbRead()

// Just return the encoded "current->edbReply"
Status
gdbRead(Event *readEvent)
{
    EtnRpcHost *DHost;
    EtnBufferEncoder *DEncoder;
Status status;

EtnLength memSize = sizeof(GdbProxyReply) + sizeof(uint64_t)*2, len;

uint8_t *mem = (uint8_t *)malloc(charXtype, memSize);

DPEncoder = etnBufferEncoderNew(mem, memSize);

DPHost = etnRpcHostNew(EtnToValue(&EdbRpcServer, 0),
                        (EtnEncoder *)
                        DPEncoder, NULL);

status = edbRpcDebugReplyCall(DPHost, 0,
                               current->
                               edbReply
                               , &len
                               );

ASSERT(status==StatusOk);

readEvent->eventReturn.ref = refAllocateInitialize(charXtype,
                                                   memSize, mem);

if (readEvent->eventReturn.ref == NULL) {
    status = StatusNoMemory;
}
In Listing 6.3 the instructions used in user-space to request a debugging operation to the kernel is reported. It is important to note that we build the `Reader` and `Writer` objects passing the debug portal file descriptor.

Listing 6.3. RPC call and reply

```go
func edbRpcDebug(e *Encoder, cid uint64, p *GdbProxyCall) {
    if !sessionState.Attached {
        log.Printf("gdbProxy %d: attempting to execute a debug operation before the attach!\n", syscall.GetPid())
        sessionState.Terminate = true
        return
    }

    reader := ethos.NewReader(sessionState.DebugPortalFd)
    writer := ethos.NewWriter(sessionState.DebugPortalFd)
    kernelEnc := NewEncoder(writer)
}
```
The `write()` system call, with the switch statement that manages debug portal file descriptors and calls the appropriate managing function (`gdbWrite()`) is reported in Listing 6.4. The `read()` system call is similar, except that it will call the `gdbRead()` function.

Listing 6.4. write() system call

```c
Status
syscallWrite(arch_interrupt_regs_t *regs)
{
    Event *event = NULL;
    Fd fd = 0;
    RdId fdId;
    void *ptr;
    Status status = StatusOk;
    RetirePair retirePair;
    String *contents = NULL;
    Ref *timeString = NULL;
```
EventId eventId = 0;

FdType fdType;

TimePrintString string;

getArgs(retirePair);

fd = retirePair.fd;

getRefBuffer(contents, retirePair.memStruct);

status = fdFind(fd, &fdType, &fdId);

if (status != StatusOk)
{
    printk("syscallWrite[%llu]: invalid %u
", current->processId, fdType);
    goto done;
}

status = eventCreateUserspace(
    EventClassContinue, &event);

if (status != StatusOk)
{
    goto done;
}
switch (fdType)
{
    case FdTerminal:
        status = terminalWrite(fdId, contents, event);
        break;
    case FdDirectory:
        // Request the read.
        timeString = stringAllocateInitialize(timeOfDayString(string));
        status = fileWriteVar(fdId, timeString, contents, event);
        refUnhook(&timeString);
        break;
    case FdIpcImporter:
    case FdIpcInitiator:
        status = ipcWrite(fdId, fdType, contents, event);
        break;
}
case FdDebug:
    status = gdbWrite(contents, event);
    break;

default:
    xprint("processId=$[handle] invalid fdType=%[uint]\n",
            current->processId, fdType);
    status = StatusInvalidFileType;
    break;
}
if (status != StatusOk)
{
    goto done;
}

// put it into user space
eventId = event->eventId;
putReturn(eventId);

done:
    if (event && (StatusOk!=status))
The kernel process data structure needed some modifications to enable RPC communication. In particular, after executing the debug operation, we need to save the result in the kernel memory until the user requests the data and a reply RPC is issued. This information (GdbProxyReply type) is saved in the process control block. In Listing 6.5 the process data structure is reported.

Listing 6.5. The process control block

```
// Definition of process context
```
typedef struct Process_S {
    // contains information for kernel execution environment
    KernelExec kernelExec;
    // Architecture specific state (registers, etc)
    arch_process_t specific;
    // Floating point store
    FloatingPointStore fpuStore;
    // ref to Userspace, kernel exit status
    // this is ref is copied over to file descriptors that
    // reference it
    Ref *exitStatus;
    // List of all events which are waiting on
    // this process to exit
    ListHead exitEventWaiting;
    // Events which are not yet completed.
    ListHead pendingEvents;
    // Completed Events.
    ListHead completedEvents;
    // number of pendingEvents plus completedEvents
    msize_t eventCount;
}
// Used for keeping track of process within the ready, waiting and terminated lists.
ListHead processList;

// Address Space Context.
ProcessMemory *processMemory;

// Resource list.
ListHead resourceList;

// Fd table
FdTable fdTable[FdTableSize];

// Next free slot in fd table
Fd fdNextFree;

// Deschedule time. Used by the scheduler to
// figure out whether
// a process had run over its allotted quantum.
// During scheduling,
// reset to current time + quantum length.
// The quantum comprises
// both the time spent in the kernel and in
// the program code.
s_time_t descheduleTime;

// User that owns process.
AuthUser *user;

// Process ID.
ProcessId   processId;

// Process group ID
ProcessId   processGroups[MaxProcessGroup];

// Pointer to registers on the stack – used by edb
// (see also function scheduleEnter() in schedule.c)
arch_interrupt_regs_t  *registers;

// Status to return – used by edb (because of RPC)
Status edbStatus;

// Pid of the process being debugged (edb)
ProcessId  debugPid;

// EventId used for the attach procedure
EventId  gdbProxyEventId;

EventId   trapInt3EventId;

// Reply (edb)
GdbProxyReply edbReply;

} Process;

The GdbProxyCall and GdbProxyReply are defined together with the RPC interface specification in the EdbTypes.t definition file in Listing 6.6.
Listing 6.6. The \texttt{EdbTypes.t} types definition file (for Intel\textregistered\, x86 64 bit)

\begin{verbatim}
GRegisters struct {
  R15  uint64
  R14  uint64
  R13  uint64
  R12  uint64
  Rbp  uint64
  Rbx  uint64
  R11  uint64
  R10  uint64
  R9   uint64
  R8   uint64
  Rax  uint64
  Rcx  uint64
  Rdx  uint64
  Rsi  uint64
  Rdi  uint64
  Pad  uint64
  Error_code64 uint64
  Rip   uint64
  Cs   uint64
}
\end{verbatim}
Eflags64 uint64
Rsp uint64
Ss uint64

GdbProxyCall struct {
    Id uint8
    Pid uint64
    Addr *uint8
    Size uint32
}

GdbProxyReply struct {
    Id uint8
    AttachState uint64
    PacketSize uint64
    GReg GRegisters
    Memory [512]uint8
    MemorySize uint16
}
EdbRpc interface {
  Debug(cid uint64, p GdbProxyCall) (rcid uint64, r GdbProxyReply)
  Attach(cid uint64, pid uint64) (rcid uint64, r uint64)
}

6.2 “Attach” and breakpoint insertion

When a breakpoint is hit (INT 3 command on the Intel® x86 architecture) a management function is called (trap handler). In Ethos, breakpoints are based on events.

In particular, when we want to attach to a running process, we insert a breakpoint in the location specified by its instruction pointer (IP) so that the next instruction will be INT 3 ¹.

Before resuming execution of the aforementioned process, an event is created and waited upon. The event ID is saved in the debuggee process data structure (gdbProxyEventId, see Listing 6.5).

When the debuggee process is resumed, the trap is encountered and trap handler is executed. The trap handler basically creates another event and saves it in the process data structure (trapInt3EventId, see Listing 6.5), completes the gdbProxyEventId event and waits on trapInt3EventId.

Finally, the debugging process will resume (because its waiting event was completed by the trap handler), it will restore the breakpoint (the 0xCC byte) to its original value, and will notify the user of the successful attach.

¹In the Intel® x86 architecture, we substitute the first byte of the instruction with the byte 0xCC.
After attaching, debuggee process execution can be resumed (for example, with the `continue` command) completing the event that the debuggee process is waiting on (the event associated to the `trapInt3EventId` event ID).

Breakpoint insertion works in the same exact way, but in that case the INT 3 instruction (namely, the 0xCC byte) is inserted at an address in the debuggee process memory space specified by GDB.

In Listing 6.7 there is the source code of the trap handler and in Listing 6.8 there is the function invoked by `gdbWrite()` to perform the debuggee process attach routine (of course, when we modify the debuggee process memory, since the debuggee process has a different page table, we need to perform a page table switch using Ethos provided low-level functions as it is evident from the source code).

**Listing 6.7. INT 3 trap handler routine**

```c
void
trapInt3(
    arch_interrupt_regs_t *regs
)
{
    ArchPrivilegeState state = archPrivilegeState(regs);
    if (state == PRIVILEGED_MODE)
    {
```
printk("Unhandled trap (Int3) in kernel\n");

dbgExit(regs);

}

else
{

Event *event = NULL, *gdbEvent = NULL;
Status status = StatusOk;

ASSERT(current);

printk("Process %llu caused an Int3 fault (trap 3) – handled\n",
current->processId);

status = eventCreateUserspace(EventClassContinue, &event);
if (status != StatusOk)
{

printk("Event creation failed (trap Int3).\n");
archDebugRegsDump2(regs);
processKill(current);
}
current->trapInt3EventId = event->eventId;
gdbEvent = eventFind(current->gdbProxyEventId);
ASSERT(gdbEvent);

status = eventComplete(gdbEvent);
if (status != StatusOk)
{
    printk("eventComplete returned an invalid status.\n");
    archDebugRegsDump2(regs);
    processKill(current);
}
printk("Trap Int3: eventcompleted\n");

status = eventWaitTreeCreateBlockAndDestroy(current->
    trapInt3EventId);
if (status != StatusOk)
{
    printk("eventWaitTreeCreateAndBlock returned an
    invalid status.\n");
    archDebugRegsDump2(regs);
processKill(current);
}

printk("Trap Int3: returned from
eventWaittreeCreateBlockAndDestroy\n");
}

Listing 6.8. EDB attach routine

// Performs attach procedure.

// pid: PID of process to be debugged (originally specified by the remote gdb proxy)

static Status edbAttach(ProcessId pid) {
    Status status;
    Process *p;
    // Address where to insert a breakpoint
    BreakpointOpcode *bkpointAddr;
    // Backup of breakpoint location (to be restored)
    BreakpointOpcode bkpointBackup;
    // User event created to wait for breakpoint to be reached
    Event *event;
status = processFind(pid, &p);

if (status != StatusOk) {
    // The pid could refer to a process which does not exist anymore...
    printk("edb, process not found\n");
    return status;
}

bkpointAddr = archedbGetInstructionPointer(p);

// Substitute next instruction with INT3
processMemorySwitch(p->processMemory);

bkpointBackup = *bkpointAddr;
*bkpointAddr = ARCHEDB_INT3_OPCODE;
processMemorySwitch(current->processMemory);

// Create an event
status = eventCreateUserspace(EventClassContinue, &event);
if (status != StatusOk) {

printk("Event creation failed during attach.\n");
return status;
}

// Save event ID into process data structure of the process being debugged
p->gdbProxyEventId = event->eventId;

// Wait for the process being debugged to reach the breakpoint and complete the event
status = eventWaitTreeCreateBlockAndDestroy(p->gdbProxyEventId);
ASSERT(status == StatusOk);

// Restore original operation and value of instruction pointer register
processMemorySwitch(p->processMemory);
*bkpointAddr = bkpointBackup;
archedbSetInstructionPointer(p, bkpointAddr);
processMemorySwitch(current->processMemory);

current->debugPid = pid;  // Save pid of process being debugged
current->edbReply.attachState = ARCHEDB_ATTACH_SUCCESSFULL;
return StatusOk;
}

6.3 Remote packets support

As of now, EDB supports the packets listed in Table IV. As Ethos is a single-threaded OS, the support of many packets managing multi-threaded applications debugging is unnecessary, thus simplifying the implementation.

Of course, the GDB commands that rely on these packets to work are all supported.

6.4 The process proxy

The process proxy provides three services:

1. Received the debug portal and terminate portal file descriptors.

2. Services requests for the ps command on /services/processStatus/user.

3. Services (debugging) requests from netStackGo on /services/gdbProxy/user.

All of this requests are satisfied as described in the previous chapters. The most remarkable implementation aspect of the process proxy is the use of Ethos’ event wait trees to wait on the events associated to each of the aforementioned services.

An event wait tree is a data structure that describes a set of events and properties associated with them. A user can block on the tree and execution will resume only when the tre is unblocked (which can happen when, for example, when a node’s children events all terminates, or only a
TABLE IV

EDB: SUPPORTED PACKETS

<table>
<thead>
<tr>
<th>Packet</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Indicate the reason the target halted.</td>
</tr>
<tr>
<td>B addr,mode</td>
<td>Set (mode is ‘s’) or clear (mode is ‘c’) a breakpoint at addr.</td>
</tr>
<tr>
<td>c [addr]</td>
<td>Continue. addr is the address from where to resume. If addr is omitted, resume at current address.</td>
</tr>
<tr>
<td>D</td>
<td>Detach GDB from the remote system. It is sent to the remote target before GDB disconnects via the <code>detach</code> command.</td>
</tr>
<tr>
<td>g</td>
<td>Read general registers.</td>
</tr>
<tr>
<td>G XX...</td>
<td>Write general registers.</td>
</tr>
<tr>
<td>m addr,length</td>
<td>Read length bytes of memory starting at address addr.</td>
</tr>
<tr>
<td>M addr,length:XX...</td>
<td>Write length bytes of memory starting at address addr. XX... is the data and each byte is transmitted as a two-digit hexadecimal number.</td>
</tr>
<tr>
<td>p n</td>
<td>Read value of register n.</td>
</tr>
<tr>
<td>P n...=r...</td>
<td>Write register n... with value r...</td>
</tr>
<tr>
<td>q name params...</td>
<td>General query packets. Only the minimum necessary query packets are supported (for example, GDB has to know the maximum size in bytes of a packet and a specific query packet is supported for that purpose).</td>
</tr>
<tr>
<td>s [addr]</td>
<td>Single step. addr is the address at which to resume. If addr is omitted, resume at same address.</td>
</tr>
</tbody>
</table>

subset terminates, or at least one terminates, and so on and so forth... combining nodes in a hierarchy).

The use I made of event wait trees in the process proxy is that of creating a tree that enables me to wait on the three events and awake when at least one completes. This is a typical example of Ethos events in action. Further details are in Listing 6.9
Listing 6.9. The process proxy

... 

// Advertise processStatus
procStatusListen, status := ethos.Advertise(procStatusServiceFd, "processStatus/"+syscall.GetUser())
if status != syscall.StatusOk {
    log.Fatalf("Error calling Advertise for procStatusServiceFd : %v
\n", status)
}

// Advertise gdbProxy

gdbProxyListen, status := ethos.Advertise(gdbProxyServiceFd, "gdbProxy/"+syscall.GetUser())
if status != syscall.StatusOk {
    log.Fatalf("Error calling Advertise for gdbProxyServiceFd : %v\n", status)
}

// Create event tree

eventTree := make([]syscall.EventTree, 4)
eventId, status := syscall.FdReceive()

if status != syscall.StatusOk {
    log.Fatalf("Error calling FdReceive : %v\n", status)
}

eventTree[0] = syscall.EventTree{EventId: eventId, YetToBeSatisfied:
    1, Parent: 3}

eventId, status = syscall.Import(procStatusListen)
if status != syscall.StatusOk {
    log.Fatalf("Error calling Import on procStatusListen : %v\n", status)
}

eventTree[1] = syscall.EventTree{EventId: eventId, YetToBeSatisfied:
    1, Parent: 3}

eeventId, status = syscall.Import(gdbProxyListen)
if status != syscall.StatusOk {
    log.Fatalf("Error calling Import on gdbProxyListen : %v\n", status)
}
eventTree[2] = syscall.EventTree{EventId: eventId, YetToBeSatisfied: 1, Parent: 3}

for {
    // Reset root
    eventTree[3] = syscall.EventTree{EventId: 0, YetToBeSatisfied: 1, Parent: 0}
    // Block on event tree
    status = syscall.Block(eventTree)
    if status != syscall.StatusOk {
        log.FatalError("Error blocking on event tree : %v\n", status)
    }

    if eventTree[0].YetToBeSatisfied == 0 { // Received a file descriptor
        ...
    } else if eventTree[1].YetToBeSatisfied == 0 { //
        procStatusListen
        ...
    } else if eventTree[2].YetToBeSatisfied == 0 { // gdbProxyListen
        ...
    }
6.5 The fork wrapper

The fork wrapper is a small piece of code that simply calls the (real) fork, called `fork_wrapped()` on behalf of the user and before returning sends (via `fdSend()` the terminate and debug portal file descriptors to the process proxy (see Listing6.10).

Listing 6.10. The fork wrapper

```c
fork(ulong level, Fd *terminateFd, Fd *debugFd) {

    Status status;
    ProcessId before, after; // To detect when we are executing as father
    Fd fdArray[2]; // File descriptors to fdSend() to user process proxy

    before = getPid();
    status = fork_wrapped(level, terminateFd, debugFd);
    after = getPid();
}
```
if (StatusOk == status && before == after) { // Father process
    fdArray[0] = *terminateFd;
    fdArray[1] = *debugFd;

    status = fdSend(fdArray, 2, (char *)getUser().ptr, "procProxy");
    return status;
}

return status;

6.6 The GDB proxy

The GDB proxy, which is the core of EDB, is actually reduced to a few lines of code. It is forked by the process proxy and inherits the imported IPC file descriptor to communicate with the remote GDB proxy.

It merely reads commands from the remote GDB proxy and forwards them to the kernel using a RPC, and viceversa (see Listing 6.11).

Listing 6.11. The GDB proxy

... reader := ethos.NewReader(gdbProxyImported)
writer := ethos.NewWriter(gdbProxyImported)

gdbProxyEnc := edbtypes.NewEncoder(writer)
gdbProxyDec := edbtypes.NewDecoder(reader)

// Init debugging session state
edbtypes.InitSession(gdbProxyImported)

// Core debugging session
for !edbtypes.Terminate() {
    gdbProxyDec.HandleEdbRpc(gdbProxyEnc)
}

syscall.Close(gdbProxyImported)

// GDB proxy
func edbRpcDebug(e *Encoder, cid uint64, p *GdbProxyCall) {
    if !sessionState.Attached {
        log.Printf("gdbProxy %d: attempting to execute a debug operation before the attach!\n",
            syscall.GetPid())
    }
}
6.7 The remote GDB proxy

The remote GDB proxy is a very rudimental parser of the RSP protocol.

The remote proxy is run on Linux before GDB. It services requests from GDB on a local port (one can even run GDB on a physically remote machine and connect to the remote GDB proxy via TCP/IP).

The remote proxy simply listen for requests from GDB on the socket, parses them, “translates” the request into a {\texttt{GdbProxyCall}} to send to Ethos via {\texttt{netStackGo}}, waits for a reply (a
GdbProxyReply structure) from Ethos, translates the reply into a RSP reply packet and sends the packet via the socket to GDB.

The remote proxy stops when a reply from Ethos indicates or implies a terminating condition and notifies GDB accordingly so that the user might be able to read an error condition (if any) and conclude the debugging session.
CHAPTER 7

CONCLUSIONS

GDB is a flexible, well designed and portable debugger. In this work we have demonstrated how simple it can be to “utilize” the features of GDB to simplify the development and implementation of debugging facilities on a novel operating system.

The RSP, originally intended for embedded systems debugging, has been adapted and exploited to reduce the complexity of implementation. I managed to focus on the design of user-space debugging functionalities for Ethos, delegating the more daunting tasks of executable format file parsing, debugging session state management and user interface implementation to GDB. This methodology, that I documented extensively in this thesis, can be of reference for other work on the subject.

The implementation of EDB required little to no modification of Ethos’ kernel data structures and a single kernel source file, \texttt{edb.c}, contains all of the most relevant newly implemented functionalities. Most importantly, authorization and authentication of debugging operations where achieved without even considering the problem, but just taking advantage of Ethos’ built in security features.

In conclusion, I implemented a user-space debugger for Ethos minimizing development effort and I managed to design the debugger in a way that it can take full advantage of Ethos’ embedded security features. This resulted in a debugger that despite “conventional” ones is \textit{secure}—authorization and authentication of debugging operations are always guaranteed.
This work also demonstrates the effectiveness, flexibility and simplicity of the Ethos operating system interface. In fact, by just using the already defined interfaces with embedded security features, I was able to seamlessly design a secure debugger without even considering the inner implementation details and interfaces of the authentication and authorization algorithms used inside of Ethos.


VITA

NAME:

Fernando Visca

EDUCATION:

MSc in Electrical and Computer Engineering, University of Illinois at Chicago, 2014
MSc in Computer Engineering, Politecnico di Torino (Torino, Italy), 2014
BSc in Computer Science, ”La Sapienza” Universit di Roma (Roma, Italy), 2011