

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATED FUZZ TESTING OF APPLICATIONS USING D-BUS COMMUNICATION SYSTEM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

MATÚŠ MARHEFKA

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

AUTOMATIZOVANÉ FUZZ TESTOVÁNÍ APLIKACÍ
KOMUNIKUJÍCÍCH PŘES SYSTÉM D-BUS
AUTOMATED FUZZ TESTING OF APPLICATIONS

USING D-BUS COMMUNICATION SYSTEM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

VEDOUCÍ PRÁCE

SUPERVISOR

MATÚŠ MARHEFKA

Ing. PETR MÜLLER

BRNO 2013

Abstrakt

Tato práce se zabývá použitím metody fuzzingu na testování aplikací komunikujících přes systém D-Bus. První část je zaměřena na seznámení s pojmem fuzzing a na využití této metody při testování aplikací. Následuje popis systému D-Bus a jeho architektury. V druhé části se práce zabývá vhodným způsobem, jakým by se daly pomocí metody fuzzingu testovat aplikace komunikující přes systém D-Bus. V rámci práce byl implementovaný nástroj na testování aplikací, které využívají tento systém na meziprocesovou komunikaci. Při implementaci tohoto nástroje byl kladen důraz na to, aby s ním bylo možné otestovat co největší spektrum aplikací. Testování proběhlo na třech vybraných aplikacích – *GNOME Shell*, *IMSettings* a *Evince*. Ve dvou ze tří zmíněných aplikacích (*GNOME Shell*, *IMSettings*) byly nalezeny chyby, které způsobily jejich pád. Implementovaný nástroj taktéž detekoval úniky paměti v aplikaci *IMSettings*.

Abstract

This thesis discusses use of the fuzzing for testing applications communicating through D-Bus system. The first part is focused on introducing the concept of the fuzzing and on use of this method when testing applications. Subsequently, there is a description of D-Bus system and its architecture. In the second part, the thesis deals with an appropriate way of using the fuzzing method for testing applications communicating through D-Bus system. A tool was implemented within this thesis for testing applications which use this system for interprocess communication. During implementation of the tool there was an effort to make it possible to test the greatest variety of applications. Testing took place on the three selected applications – *GNOME Shell*, *IMSettings* and *Evince*. In the two of these three applications (*GNOME Shell*, *IMSettings*) was found bugs which caused their crash. Implemented tool also detected memory leaks in *IMSettings* application.

Klíčová slova

D-Bus, fuzzer, fuzzing, testování, automatizace, generace náhodných dat, jazyk C.

Keywords

D-Bus, fuzzer, fuzzing, testing, automation, random data generation, C language.

Citace

Matuš Marhefka: Automated Fuzz Testing of Applications Using D-Bus Communication System, bakalářská práce, Brno, FIT VUT v Brně, 2013

Automated Fuzz Testing of Applications Using D-Bus Communication System

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pánů
Ing. Petra Müllera a Mgr. Miloše Malíka.

.....
Matůš Marhefka
May 9, 2013

Poděkování

Thank Ing. Petr Müller for his guidance, consultations and helpful comments which improved a quality of this work. I also would like to thank Mgr. Miloš Malík for his advices and technical assistance. Finally, big thank to my family and friends for their constant support.

© Matůš Marhefka, 2013.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1	Introduction	3
2	Fuzzing	5
2.1	Fuzzing as software testing technique	6
2.2	Fuzzer categories	6
2.3	Fuzzer subcategories	7
2.3.1	Pregenerated test cases	7
2.3.2	Random test cases	7
2.3.3	Manual protocol mutation testing	7
2.3.4	Mutation or brute force testing	7
2.3.5	Automatic protocol generation testing	8
2.4	Fuzzer types	8
2.4.1	Local fuzzers	8
2.4.2	File format fuzzers	9
2.4.3	Remote fuzzers	9
2.4.4	In-memory fuzzers	10
2.4.5	Fuzzer frameworks	10
3	D-BUS	11
3.1	Protocols	11
3.2	D-Bus architecture	12
3.2.1	Objects and object paths	13
3.2.2	Methods and signals	13
3.2.3	Interfaces	13
3.2.4	Proxies	13
3.2.5	Bus names	14
3.2.6	Addresses	14
3.2.7	Calling a method	15
3.2.8	Emitting a signal	15
3.3	GDBus – GLib D-Bus binding	16
3.3.1	Connection to a message bus	16
3.3.2	Proxy for accessing D-Bus interfaces	16
3.3.3	Object introspection	16
3.3.4	Method calls	19
3.3.5	Using <code>GError</code>	19

4	Designing dfuzzer, a tool for fuzz testing processes using D-Bus system	20
4.1	dfuzzer architecture	21
4.2	dfuzzer implementation	22
5	Fuzz testing results	26
5.1	Fuzz testing <i>Evince</i>	26
5.2	Fuzz testing <i>GNOME Shell</i>	26
5.3	Fuzz testing <i>IMSettings</i>	28
6	Conclusion	30

Chapter 1

Introduction

Testing is used to find bugs (errors or other defects) in software. By finding bugs, testing also provides quality assurance of software. Software testing can be stated as the process of validating and verifying that a computer software:

- meets the design and development requirements,
- works as expected,
- is secure,
- and that bugs are not present.

Fuzzing is a type of testing where a goal is to find bugs in software, preferably ones that have security implications. Fuzzing is automated, brute force technique of testing and it exploits the fact that many bugs in software are caused by handling inputs from a user without applying validation routines on that inputs. The goal of fuzz testing (fuzzing) is to crash an application or a protocol and analyze the results. Fuzzing is presented in the next chapter.

The goal of this work is to test applications using D-Bus communication system and to implement a tool for automation of this task. The targets of testing done in this work are applications connected to a session bus daemon and using it for routing messages.

In the third chapter the D-Bus message bus system is described. The chapter includes description of protocols in general, D-Bus architecture, message bus daemons, addressing on D-Bus, calling methods and emitting signals. Also the GDBus D-Bus binding is described which was used in an implementation of the fuzzing tool.

The fourth chapter is devoted to description of designing a tool for fuzzing applications using D-Bus communication system. The chapter discusses methods which can be used for testing D-Bus system and its applications. Subsequently, the architecture of the tool for fuzzing and its implementation are described.

Results of the fuzz testing performed with the implemented tool are discussed in the fifth chapter. The chapter includes tables where methods and results of their calls are stated for every tested application. Also the run times of the tests are stated and for the tests where memory leaks were detected, the tables also contains a normal process memory usage against an abnormal one.

The conclusion summarizes what was the purpose of this work, how a fuzzing was used to test applications using D-Bus communication system, evaluation of results and achievements and suggestions for improvements.

Chapter 2

Fuzzing

The term fuzzing was first used by professor Barton Miller who used fuzzing to test robustness of UNIX applications in 1989 [2]. Fuzzing is defined in “Fuzzing: Brute Force Vulnerability Discovery” [6, p. 22] as “*a method for discovering faults in software by providing unexpected input and monitoring for exceptions. It is typically an automated or semiautomated process that involves repeatedly manipulating and supplying data to target software for processing*”. Another definition (from “Fuzzing for Software Security Testing and Quality Assurance” [2, p. 1]) says it is “*a highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities. The name comes from modem applications tendency to fail due to random input caused by line noise on ‘fuzzy’ telephone lines*”. Some other terms used to describe tests similar to fuzzing include [2, p. 24]:

- Negative testing
- Protocol mutation
- Robustness testing
- Syntax testing
- Fault injection
- Rainy-day testing
- Dirty testing

The aim of fuzzing is to crash a program or a protocol and analyze the crash results. This is why many vendors want a crash data. It is very important because a crash tells so much about a program. Fuzzing is close to the boundary value analysis, where you create test values that infringe the boundary of known good and bad values. Fuzzing is very effective because many exploitable vulnerabilities are caused by applications accepting user input and processing that data without applying validation routines [6].

The division of fuzzers to categories and subcategories as also information gathered in this chapter are from books “Fuzzing: Brute Force Vulnerability Discovery” [6] and “Fuzzing for Software Security Testing and Quality Assurance” [2].

2.1 Fuzzing as software testing technique

Testing in general uses two different approaches. The “white box” testing is used to test internal structures of an application as testers have access to a source code of an application and so are able to test many conditions and paths through the code base and uncover potential errors. In the “black box” approach testers do not have a knowledge of an application internals. Testers are aware of what the application is supposed to do but are not aware of how it does it. Rather than internals the functionality of an application is tested and the same applies for fuzzing as it is essentially the functional testing technique focused on the software security. The functional testing can be viewed as “black box” with one or more external interfaces available for injecting test cases, but without any other information available on the internals of the tested system.

One of a fuzzing main goals is to crash a system. It is a testing approach that would almost never occur in a normal environment (the negative testing) and it is used to test software robustness. As stated in “Fuzzing for Software Security Testing and Quality Assurance” [2, p. 18] robustness of software is “*an ability to tolerate exceptional inputs and stressful environmental conditions. Software is not robust if it fails when facing such circumstances. Attackers can take advantage of robustness problems and compromise the system running the software. Most security vulnerabilities reported in the public are caused by robustness weaknesses*”.

Programs and frameworks that are used to create fuzz tests or perform fuzz testing are commonly called fuzzers. Fuzzers fit the best for finding errors that can cause a program to crash, such as buffer overflows, denial of service attacks, format bugs and SQL injections. Fuzz testing is less effective for finding a security threats that do not cause program crashes, such as spyware, some viruses, worms, trojans and keyloggers.

Fuzz testing is simple and can often reveal defects that are overlooked when software is designed, written and debugged. Fuzz testing is usually used to find the faults that a normal testing is not able to detect. As any other type of a testing, fuzz testing also cannot guarantee a complete security, quality and effectiveness of software.

2.2 Fuzzer categories

Fuzzers can be divided into two groups:

- **Mutation-based fuzzers** apply mutations on existing data samples to create test cases. This means that existing chunk of data is taken by a fuzzer and fields which are defined as modifiable by specification within this chunk are modified each time before sending that data to the tested software. Modifications can be random or driven by some protocol specifications.
- **Generation-based fuzzers** create test cases from scratch by modeling the target protocol of a certain format. Creating test cases includes use of mechanisms to generate valid data for a protocol or a valid computer program for a compiler fuzz testing. Mechanisms to generate such valid data can include finite state machines, formal grammars or formal languages.

2.3 Fuzzer subcategories

Generally we can divide fuzzers into these subcategories:

- pregenerated test cases
- random test cases
- manual protocol mutation testing
- mutation or brute force testing
- automatic protocol generation testing

2.3.1 Pregenerated test cases

This method was used in *PROTOS framework* [7]. The method includes studying a particular specification to understand all supported data structures and the acceptable value ranges. Packets or files are then generated to test boundary conditions or violate the specification. A disadvantage is that there is no random component, once the list of test cases is exhausted, fuzzing is complete.

2.3.2 Random test cases

This method simply generate pseudo-random data and uses it as the target input, waiting for the result. Example of this can be:

```
while [1]; do cat /dev/urandom | nc -vv localhost 22; done
```

This command reads random data from Linux urandom device and then transmits that data to localhost address on port 22 (ssh). The biggest disadvantage of this simple technique is tracking back how some random bytes caused an application crash. This includes capturing the input we sent to application and also debugging a corrupted stack.

2.3.3 Manual protocol mutation testing

In manual protocol mutation testing there is no automated fuzzer involved. The researcher is the fuzzer, simply entering inappropriate data in an attempt to crash an application or to find some undesirable behaviour. The success depends on the researcher knowledge and experience. This class of fuzzing is most often applied to web applications.

2.3.4 Mutation or brute force testing

A brute force in this class of fuzzing is referring to a fuzzer that starts with a valid sample of a protocol or data and continually mangles some amount of bytes within that data packet or file. It requires only a little research and an implementation of a brute force fuzzer is relatively straightforward. The main advantage is that the process is fully automated. Disadvantages are that it takes many samples of data to get decent coverage of protocol specifications or file definitions, so many CPU cycles will be wasted on data that cannot be interpreted. Examples of brute force file format fuzzers include *FileFuzz* [16] and *not-SPIKEfile* [17].

2.3.5 Automatic protocol generation testing

Automatic protocol generation testing is a more advanced method of brute force testing. In this approach, research is needed to first understand a protocol specification and a file definition. Then the grammar is created that describes how the protocol specification works. It includes identifying portions of data that are to remain static and others that represents fuzzable variables. The fuzzer then generates fuzz data for that variables and sends the resulting packet to the target. The success depends on the researcher’s ability to pinpoint those portions of the specification that are most likely to lead to faults in the target application. Examples of this type of fuzzers are *SPIKE* [1] and *SPIKEfile* [17]. Both of these tools take *SPIKE* script descriptions of their target protocol or file format and use a fuzzing engine of *SPIKE* framework to create mangled data. *SPIKE* is actually a fuzzer creation kit, providing an API that allows users to create their own fuzzers for network based protocols using the C programming language. *SPIKE* defines a number of primitives that it makes available to C coders which allows *SPIKE* to construct fuzzed messages called “SPIKES” that can be sent to a network service to hopefully induce errors.

2.4 Fuzzer types

Each type of target has its own class of fuzzer as different target software needs different approaches to fuzzing. The fuzzer types in this section are covering only basic types. There can be more different and more specific fuzzer types. Subsections in this section will describe each type of fuzzer as divided in “Fuzzing: Brute Force Vulnerability Discovery” [6].

2.4.1 Local fuzzers

Local fuzzers are used for fuzzing applications running on the same computer and operating system as local fuzzers do. They usually serve to fuzz test command line arguments, environment variables, and any other locally available interfaces. Interesting targets for local fuzzers are UNIX setuid applications which allow a normal user to temporarily gain elevated privileges. Any vulnerability in this type of applications will give a user escalated privileges and ability to execute arbitrary code. Example of such an application can be the `passwd` program:

```
ls -l --full-time /usr/bin/passwd
-rwsr-xr-x. 1 root root 27848 2012-12-04 18:27:43 +0100 /usr/bin/passwd
```

which has the setuid flag set, to allow a user to change a password¹. Setuid means Set User ID upon execution. If the setuid flag is set on a file, a user executing that file gets the permissions of the individual user or group that owns the file. You can set the setuid flag of a file in Linux by:

```
sudo chown root:root file
sudo chmod u+s file
```

This will give a user root privileges when executing the file. There are two distinct targets for fuzzing setuid applications – applications accepting command-line arguments and applications using environment variables.

¹in “-rwsr-xr-x” privileges string, the “s” letter means a file has the setuid flag set

Command-line argument fuzzers

Applications usually process command-line arguments as strings. The following example demonstrates command-line argument stack overflow:

```
#include <string.h>
int main(int argc, char **argv) {
    char buffer[5];
    strcpy(buffer, argv[1]);
}
```

If it would have `setuid` bit set, it could be misused to get elevated privileges. The easiest way to find such errors (even without source code available) is fuzzing. Useful tools for command-line fuzzing can be *clfuzz* [5] and *iFUZZ* [17] which can be used for format string and buffer overflow testing.

Environment variable fuzzers

Another local fuzzer type is a fuzzer testing environment variables. Example of an error where value from the environment variable `HOME` is used unsafely:

```
#include <string.h>
int main(void) {
    char buffer[5];
    strcpy(buffer, getenv("HOME"));
}
```

This is the similar error as for command-line arguments – the buffer overflow. Tools for fuzzing this type of bugs are *Sharefuzz* [1] and *iFUZZ* [17].

2.4.2 File format fuzzers

Many applications are working with file input and output. All applications which use configuration files need to parse them. These applications must properly handle file parsing even if files are malformed (maliciously or by some random damage). The file format fuzzers are used to test these applications which must properly handle file input. A file format fuzzer will dynamically create different malformed files that are then parsed using the target application. The examples of useful tools for file fuzzing are *notSPIKEfile* [17] and *SPIKEfile* [17].

2.4.3 Remote fuzzers

Remote fuzzers are used for testing software that listens on a network interface. As most attacks are done remotely and can provide an attacker with access to sensitive data, targeting these applications for testing is important. Targets for this type of fuzz testing include network protocols, web applications and web browsers.

Network protocol fuzzers

Network protocol fuzzers can be divided into two categories based on protocol complexity. Simple protocols often have simple authentication or no authentication at all. They

are usually based on ASCII text and do not contain checksum or length fields. Complex protocols, on the other hand are comprised of binary data and they might use encryption for authentication. Useful tools for network protocol fuzzing are *SPIKE* [1] or *Peach fuzzer* [3].

Web application fuzzers

As web applications became popular they are used to access back-end services as e-mail, internet banking, and many more. Even traditional desktop applications as word processing are available on the Web. Web application fuzzers must be capable of communicating via HTTP protocol and they are looking for vulnerabilities unique to specific Web applications. Example of web application fuzzer is *WebScarab* [8] which behaves as a proxy intercepting web browser web requests and web server replies.

2.4.4 In-memory fuzzers

In-memory fuzzers can be implemented by freezing and taking a snapshot of a process and rapidly injecting faulty data into one of its input routines. After each test case, the snapshot taken previously is restored and new data is injected. This is repeated until all of the test cases are exhausted.

2.4.5 Fuzzer frameworks

A fuzzing framework is a generic fuzzer or fuzzer library that simplifies data representation for many types of targets. Fuzzing framework usually includes a library to produce fuzz strings or values that produce problems in parsing routines. It should also include a script-like language for creating a specific fuzzer. The most important property of generic fuzzers is reusability. The main disadvantages are development time of a framework, its complexity and sometimes limitations (target of fuzzing is not suitable for framework). Fuzzing frameworks include *SPIKE* [1] and *Peach fuzzer* [3].

Chapter 3

D-BUS

D-Bus is a protocol and a message bus system providing applications a simple way to talk to one another. D-Bus is “*a system for inter-process communication (IPC) and makes it simple and reliable to code a ‘single instance’ application or daemon, and to launch applications and daemons on demand when their services are needed*” [4].

The low-level API for D-Bus is written in C but most of the documentation and code is written for a higher level binding (Python or GLib). D-Bus has both the system bus daemon (for events such as “new hardware device added” or “printer queue changed”) and the session bus daemon (for general inter-process communication needs among user applications). The message bus is built on top of a general one-to-one message passing framework, which can be used by any two applications to communicate directly (without going through the message bus daemon). The communicating applications are either on one computer, or they communicate through unencrypted TCP/IP socket suitable for use behind a firewall [4].

The source of information in this chapter was “D-Bus specification” [10], “D-Bus tutorial” [11] and “Red Hat Magazine” [9].

3.1 Protocols

Computers use protocols in all aspects of internal and external communication. Protocols represent a structure for data transfer and processing with defined syntax. Certain standards and rules understood by both sending and receiving entities must be agreed on to communicate data in meaningful way [6].

Some protocols are designed to be human readable and are represented in plain text form. Other protocols are represented in binary format.

Plain text protocols

These protocols are human readable, they use mostly the printable ASCII scheme. This includes numbers, lowercase and capital letters, symbols, carriage returns (`'\r'`), new lines (`'\n'`), tabs (`'\t'`) and spaces. Plain text protocols are less efficient than binary protocols as they are more memory consuming, but they are easy to debug and analyse [6]. Many plain text protocols are transported in serialised XML or JSON formats.

Binary protocols

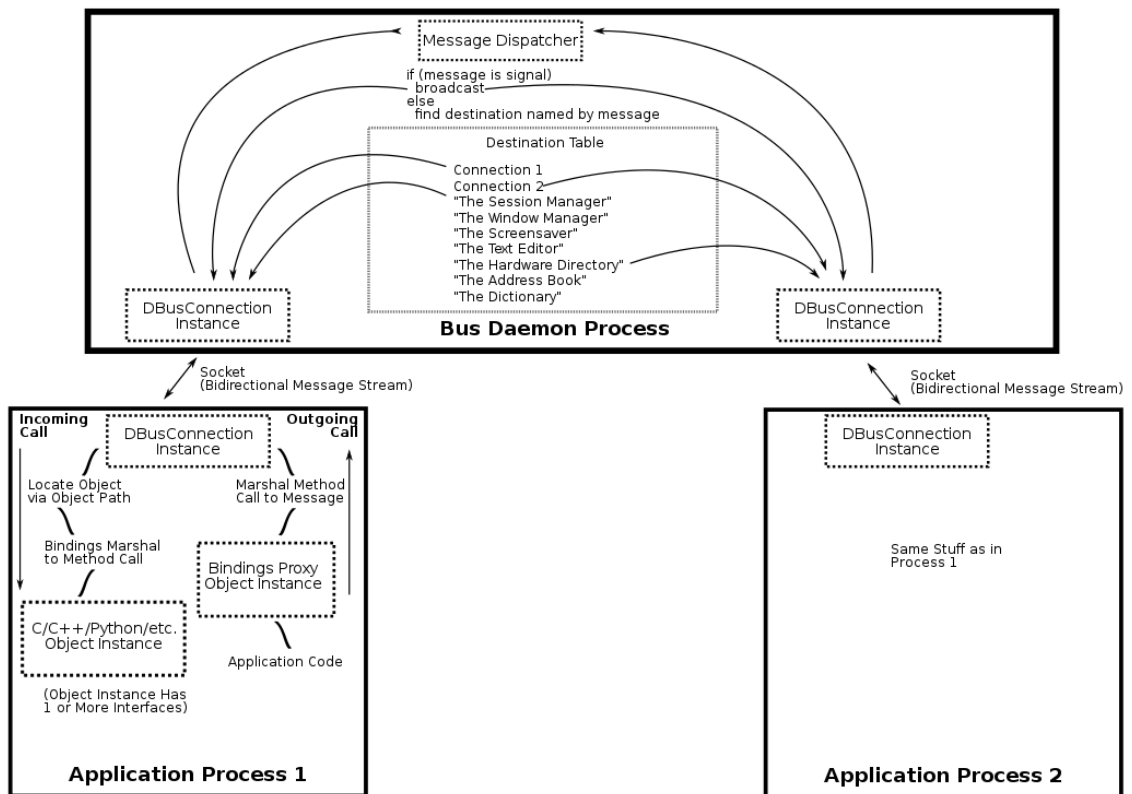
Binary protocols consist of a stream of raw bytes. Without an understanding of the protocol, the packets will not be meaningful [6]. They are intended or expected to be read by a machine rather than a human being, as opposed to a plain text. Binary protocols have advantage of terseness, which apply to a speed of transmission and interpretation. Binary protocols include TCP, UDP, RTP or SSH.

3.2 D-Bus architecture

D-Bus has several layers:

- A library, `libdbus`, that allows two applications to connect to each other and exchange messages.
- A message bus daemon executable, built on `libdbus`, that multiple applications can connect to. The daemon can route messages from one application to zero or more other applications.
- Wrapper libraries or bindings based on particular application frameworks.

Figure 3.1: D-Bus overview [4]



D-Bus contains the bus daemons which act as routers for messages. Processes can connect to the bus daemons to use their routing services as stated on D-Bus overview 3.1. There are two standard buses: the system bus and the session bus.

The system bus is a global daemon that any application running in any context can use as a transport router. It is a single point where applications can export services that anyone can use. Only one system bus daemon can be run at a time within an operating system.

The session bus is the bus local to the current user's session. It is used for communication between applications running within the same X Window System¹ session. For every login to X, a session bus daemon is started.

D-BUS protocol is binary, which means D-BUS messages incur low overhead when marshaling and demarshaling data. Messages consist of two sections, the header and the body. The header contains routing information and the type signature for the data. The body contains the data being sent in binary format. Each piece of data has a type code associated with it and is packed into the body accordingly. Some common types include bytes, 32 and 64 bit integers, doubles, unix file descriptors and strings. Common data types can be used to build more complex data types such as arrays, dictionaries or structures.

3.2.1 Objects and object paths

Messages are sent to objects. A D-Bus objects are conceptually similar to those found in object oriented programming languages with the exception that they are pointed to by object paths (not by memory addresses). Object paths are in forms of strings similar to Unix file system paths. D-Bus exports the `/org/freedesktop/DBus` object. Applications can register as many objects as they wish.

3.2.2 Methods and signals

Each object has members – methods and signals. Methods are operations that can be invoked on an object, with optional input (arguments) and output (return values). Signals are broadcasts from the object to any application on the same bus, which registered that it is interested in signals emitted by this object. Signals may contain a data payload. Methods and signals are referred to by name.

3.2.3 Interfaces

Each object supports one or more interfaces. An interface is a named group of methods and signals (as in GLib or Qt). Interfaces allow the same method name to be used more than once, so the interface is specifying which of those methods is actually invoked. Interfaces define the type of an object instance. D-Bus identifies interfaces with a simple namespaced string. Most bindings to other programming languages have mapping of those interface names directly to the appropriate programming language constructs (C++ virtual classes for example). D-Bus imports the `org.freedesktop.DBus` interface.

3.2.4 Proxies

A proxy object represents a remote object in another process. In the low-level D-Bus C API it is necessary to manually create a method call message, send it, then manually receive and process the method reply message. In higher-level bindings proxies are used to do that instead. So when a method is invoked on a proxy object, the high-level binding converts it

¹X Window is a system and a network protocol that provides a basis for graphical user interfaces

into a D-Bus low-level method call message, waits for the reply message, unpacks the return value, and returns it from the native D-Bus method.

3.2.5 Bus names

When each application connects to the bus daemon, the daemon immediately assigns it a name, called the unique connection name. This unique name begins with a colon character. Bus daemon ensures that these unique names are never reused during the lifetime of the bus daemon, so given unique name will always refer to the same application. An example of a unique name might be `:1.396`. The numbers after the colon have no meaning besides that they must be unique within the bus daemon. Application starts to own a name on the bus daemon as soon as the name is mapped to a particular application connection.

Applications may also own additional well-known names (also called services). A D-Bus application may register one or more well-known names (services) that it will then own until it releases them. Example could be the name called `org.freedesktop.dfuzzer`.

If application wants to own this well-known name (service), it should have an object at the path `/org/freedesktop/dfuzzerObject`. This object must be supporting the interface `org.freedesktop.dfuzzerInterface`. Other applications connected to the same bus daemon could then send messages to this bus name (service), object and interface to execute method calls.

In general, the unique names can be thought of as IP addresses and the well-known names as domain names. So `org.freedesktop.dfuzzer` might map to something like `:1.396` just as `google.com` maps to IP address `173.194.35.70`.

Using well-known names (services) brings one big advantage. When an application crashes or exits, the operating system kernel has a responsibility to close its connection to the message bus. As soon as the message bus daemon registers that an application disconnected, it sends out notification messages to all remaining applications that the disconnected application names have lost their owner. This can be used by other applications to monitor the lifetime of an application in which they are interested or an application they communicate with.

3.2.6 Addresses

Applications which use D-Bus are either servers or clients. A typical server would be the bus daemon which listens for incoming connections. Client on the other hand initiates a connection to the server (typically the bus daemon) and when the connection is established, communication is a symmetric flow of messages. Using client-server architecture only matters when setting up connections.

A D-Bus address specifies where a server will listen to connections, and where a client will connect. An example could be the address `unix:path=/tmp/abcdef` which specifies that the server will listen on a UNIX domain socket at the path `/tmp/abcdef` and the client will connect to that socket. An address of a server can also specify TCP/IP sockets or any other transport mechanism.

The address of the session bus daemon can be determined by reading an environment variable – in D-Bus protocol, this is done by `libdbus` automatically. `libdbus` also discovers the system bus daemon by checking a well-known UNIX domain socket path (an environment variable can be also used for the system bus daemon discovery).

In an unusual case when not using a bus daemon, there is a need to define client-server architecture – who will be the server and who will be the client. Also a mechanism for them to agree on the server address must be specified.

3.2.7 Calling a method

To make a particular method call on a particular object instance, a number of nested components have to be named:

`Address -> [Bus Name] -> Path -> Interface -> Method`

The bus name is optional (in brackets), as you only use it to route the method call to the right application when using the bus daemon. Otherwise a direct connection is used, so bus names are not used as there is no bus daemon.

The interface is used primarily for historical reasons, but it should be used anyway. If the interface is omitted, a method name is ambiguous and it is undefined which method will be invoked. When calling a method, two messages are routed through the bus daemon:

- a method call message sent from process A to process B
- a matching method reply message sent from process B to process A

The caller includes a different serial number in each call message, and the callee includes this number to allow the caller to match replies to calls.

D-Bus methods may accept any number of arguments and may return any number of values, including none. When method calls specify no return value, a “method return” message is still sent to the calling application. This allows applications using the remote API to know that the remote method invocation has completed even if no useful result is returned.

The only way to suppress the generation of the reply message in an acknowledgement to a D-Bus method call is if the “no reply expected” flag is sent as part of the method invocation. It is an optional D-Bus implementation feature.

3.2.8 Emitting a signal

A signal in D-Bus consists of a single message, sent by one process to any number of other processes and is entirely asynchronous. Signals may be emitted by D-Bus objects at any time. A signal is an unidirectional broadcast and it may contain only arguments (a data payload), as it is a broadcast, so it never has a return value.

The emitter (sender) of signals sends them to the bus daemon. Recipients of the signals register within the bus daemon to receive signals based on match rules – these rules typically include the sender and the signal name. The bus daemon then sends signals only to those recipients who have registered them.

3.3 GDBus – GLib D-Bus binding

Bindings are used to wrap low-level D-Bus C API calls to the higher level libraries or language constructs. Whenever the low-level D-Bus API is changed, only internals of bindings must be modified. This protects all of the applications using these bindings against need to adapt their code whenever D-Bus protocol is changed. That was the main reason why GDBus binding was used for implementation of the tool for fuzz testing applications using D-Bus system. Binding to GLib was chosen because many applications use it already and it is also the main higher level binding for GNOME applications communicating through D-Bus system.

The source of information in this section was “GNOME Developer Center” and its documentations [13, 12].

3.3.1 Connection to a message bus

The `GDBusConnection` type is used for D-Bus connections to remote peers such as a message buses. It is a low-level API which provides methods for connection to the specified message bus and also setting up connection properties. When a connection is established, the `GDBusConnection` type holds information about it.

3.3.2 Proxy for accessing D-Bus interfaces

For creating a proxy, the `GDBusConnection` type, a bus name (well-known or unique), an object path and a D-Bus interface name are required. If a proxy is created for a well-known name and no name owner currently exists, the message bus will be requested to launch a name owner for the name. This means that if an application requesting some well-known name (service) creates a proxy for that well-known name (and no name owner currently exists), a name owner will be launched by message bus. On successful proxy creation, it will be stored in `GDBusProxy` base class, which could be then used to access a D-Bus interface on a remote object for calling methods or emitting signals.

3.3.3 Object introspection

Object introspection can be used to obtain information about object interfaces, interface methods and individual method parameters. The introspection file can be obtained by calling the `org.freedesktop.DBus.Introspectable.Introspect` method on an initialized proxy base class. This method name contains dots to let D-Bus know that a name is split into interface and method name parts. This allows using proxy for invoking methods on other interfaces. Call of the `org.freedesktop.DBus.Introspectable.Introspect` method returns introspection of object in serialized form in a `GVariant` type. `GVariant` value is then deserialized into classical string, which is parsed and output of parsing is used

to fill a `GDBusNodeInfo` structure. This structure can be then used to find desired interfaces, methods and arguments either by specialized `GDBus` functions or by working with pointers through a `GDBusNodeInfo` structure. Example of introspection of the `/org/gnome/Shell` object by calling tool for working with D-Bus objects, `gdbus` (some omitted interfaces are highlighted by dots):

```
$ gdbus introspect --session -d org.gnome.Shell -o /org/gnome/Shell --xml

<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
    "http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<!-- GDBus 2.34.2 -->
<node>
.
.
.
<interface name="org.freedesktop.DBus.Introspectable">
  <method name="Introspect">
    <arg type="s" name="xml_data" direction="out"/>
  </method>
</interface>
.
.
.
<interface name="org.gnome.Shell">
  <method name="Eval">
    <arg type="s" name="script" direction="in">
    </arg>
    <arg type="b" name="success" direction="out">
    </arg>
    <arg type="s" name="result" direction="out">
    </arg>
  </method>
  <method name="ScreenshotArea">
    <arg type="i" name="x" direction="in">
    </arg>
    <arg type="i" name="y" direction="in">
    </arg>
    <arg type="i" name="width" direction="in">
    </arg>
    <arg type="i" name="height" direction="in">
    </arg>
    <arg type="b" name="flash" direction="in">
    </arg>
    <arg type="s" name="filename" direction="in">
    </arg>
    <arg type="b" name="success" direction="out">
    </arg>
  </method>
  <method name="ScreenshotWindow">
```

```

    <arg type="b" name="include_frame" direction="in">
    </arg>
    <arg type="b" name="include_cursor" direction="in">
    </arg>
    <arg type="b" name="flash" direction="in">
    </arg>
    <arg type="s" name="filename" direction="in">
    </arg>
    <arg type="b" name="success" direction="out">
    </arg>
</method>
<method name="Screenshot">
    <arg type="b" name="include_cursor" direction="in">
    </arg>
    <arg type="b" name="flash" direction="in">
    </arg>
    <arg type="s" name="filename" direction="in">
    </arg>
    <arg type="b" name="success" direction="out">
    </arg>
</method>
<method name="FlashArea">
    <arg type="i" name="x" direction="in">
    </arg>
    <arg type="i" name="y" direction="in">
    </arg>
    <arg type="i" name="width" direction="in">
    </arg>
    <arg type="i" name="height" direction="in">
    </arg>
</method>
<property type="b" name="OverviewActive" access="readwrite">
</property>
<property type="s" name="ShellVersion" access="read">
</property>
</interface>
.
.
.
</node>

```

As seen from object introspection, the `/org/gnome/Shell` object has more interfaces (object must have at least one interface). Interfaces has methods and signals. Each argument of a method is declared with name, its direction and signature string (signature encodings are in table 3.1).

3.3.4 Method calls

To call a method, its name must be specified including arguments serialized in a **GVariant** type. Each argument of a method has its signature encoding as stated in table 3.1.

Table 3.1: Signature encoding

Character	Data type
y	8-bit unsigned integer
b	boolean value
n	16-bit signed integer
q	16-bit unsigned integer
i	32-bit signed integer
u	32-bit unsigned integer
x	64-bit signed integer
t	64-bit unsigned integer
d	double-precision floating point number
s	UTF-8 string (no embedded null characters)
o	D-Bus Object Path string
g	D-Bus Signature string
a	Array
(Structure start
)	Structure end
v	Variant type (GVariant type)
{	Dictionary begin
}	Dictionary end
h	Unix file descriptor

The signatures of arguments are then joined to form the signature string, which serves as format string (similar to the format strings in `printf()` function) to create a **GVariant** type containing all argument values. For example, a method accepting a string and a signed 32-bit integer and returning no values would use “(si)” for the argument signature and empty string for the return value.

3.3.5 Using GError

GLib provides a standard method of reporting errors from a called function to the calling code. Reporting errors is solved by exceptions in many other languages. Almost every function takes a return location of **GError** object as its last parameter. Caller of the function can then (after a function return) test a **GError** value if it is not `NULL`, report error message to a user and recover from the error or end a program.

Chapter 4

Designing dfuzzer, a tool for fuzz testing processes using D-Bus system

There are many ways of fuzz testing D-Bus system. In general there are two main possibilities:

- fuzz testing of the D-Bus daemons and the D-Bus protocol
- fuzz testing of D-Bus daemons clients

Fuzz testing of the D-Bus daemons and the D-Bus protocol

When fuzz testing the D-Bus daemons and its protocol there is need to use the native C D-Bus API. Bindings cannot be used, because they have strict checks of data being sent through their proxy objects. The principle of this fuzz testing would be communication of two applications through a D-Bus daemon. They would send malformed D-Bus protocol packets (messages) to each other, by mangling portions of data within these packets. Fuzzer would have to simulate a communication between bus daemon clients and it would also have to be watching the bus daemon status – its memory usage and its state.

Fuzz testing of D-Bus daemon clients

This work was considering three options of fuzz testing D-Bus clients:

- Connect to the session bus daemon and test processes connected to it by sending them messages and monitoring results
- Connect to the system bus daemon and test processes connected to it by sending them messages and monitoring results
- Simulate the session bus daemon, let processes connect and test them

The first two options – using the session/system bus daemon for routing messages between processes has many advantages. The biggest one is that all processes connected to a bus and using well-known names can be fuzz tested. This is enabled by an object introspection. The object introspection tells what interfaces does the object contain, including

interface methods and their arguments. Other advantages may be routing of messages by a bus daemon or using bindings instead of the native D-Bus C API, which is related to produce less code. Disadvantage of using the session/system bus daemon with bindings are strict data checks, which do not allow to route undesirable messages as for example invalid UTF-8 strings or object paths. Although it may seem to be a limitation for a fuzzer, other processes using bindings have this limitation too, so it would not be appropriate to test messages the processes can never send or receive through a bus daemon.

The third option – simulating session bus daemon would require using the native C D-Bus API with `libdbus` library. This includes to code a bus daemon simulator which can handle D-Bus connections and use message dispatcher with destination table to correctly route messages through D-Bus connections between processes. Fuzz testing would then include connection of tested processes to the bus daemon simulator and letting them communicate with each other. The bus daemon simulator would be modifying this communication messages and watching processes reactions to these modified messages.

The first option was chosen for the fuzzer as there are more applications on the session bus than on the system bus daemon and there is no need to implement the tool in the native D-Bus C API. The system bus daemon also uses the same mechanism for connection of applications as the session bus, and so extending the fuzzer to work also for the system bus daemon should be straightforward.

4.1 dfuzzer architecture

dfuzzer is a tool implemented within this work for testing D-Bus applications. Targeted applications for dfuzzer are session bus daemon clients. It is a command-line tool which takes options as arguments for setting up fuzz testing of a chosen application. dfuzzer is a generation-based fuzzer (2.2) and it randomly generates data for method arguments according to signatures of arguments (3.1). It is a local fuzzer (2.4) although it does not interact with tested applications processes directly, but through a system daemon – the session bus.

Division into modules

dfuzzer is using a modular architecture, and so individual problems are divided into individual modules. These modules include *random module*, *introspection module* and *fuzz module*.

The *random module* is responsible for pseudo-random data generation. It can generate data for every primitive method argument signature (3.1), except complex data types as structures, arrays of types and dictionaries which were not implemented. It also saves generated data sizes to be able to give a condition to end fuzz testing of a method.

The *introspection module* performs an object introspection. It requests introspection file of a chosen object from the session bus daemon. Returned XML file with object interfaces, methods and their arguments is then parsed into GDBus structures and desired interface is found. The *introspection module* also includes iterators for these GDBus structures to iterate through interface methods and their arguments.

The *fuzz module* is given a name of a method and its arguments signatures. Subsequently, individual arguments are created from the signatures by the *random module* for data generation. After arguments were created, a method is called. When a method call returns NULL, it means that a tested application does not respond or it disconnected from the bus daemon. The *fuzz module* is also parsing a `/proc/pid/status`¹ file to monitor a tested application process status to be able to confirm that a tested application really disconnected from the bus daemon. Process status file is also used for monitoring an application process memory usage.

4.2 dfuzzer implementation

The tool for fuzzing session bus clients was implemented in the C programming language using GDBus binding to the native D-Bus C API. The fuzzer was implemented and tested in the GNU/Linux operating system.

When a user logs in to X Window System, the session bus daemon is started. Application processes can then connect to the session bus daemon to use it for an inter-process communication. When a process is connected to the session bus, it can either provide a service (specified by well-known name) or only use the session bus daemon for communication with other processes. Usually a process with a well-known name acts as a server and a process with a unique name is initiating a connection and so acts as a client.

dfuzzer is fuzz testing only processes with well-known names on the session bus daemon. This means it needs to own only a unique name within the session bus. dfuzzer is a command-line tool which takes the following required options as its arguments:

- a process bus name
- a process object path
- a process interface

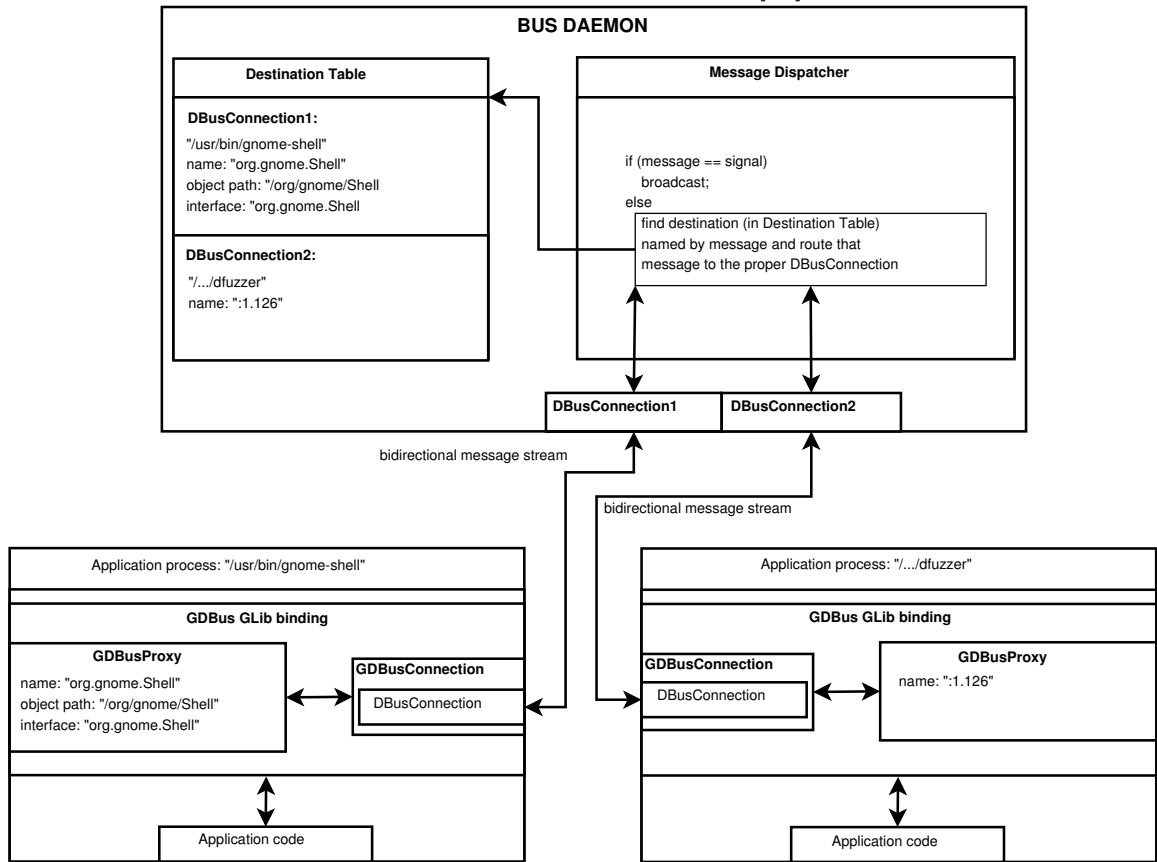
These options are required as dfuzzer must know which interface of which object and which bus name it should communicate with through the session bus daemon. Other options which are not required include:

- a name of a file for logging (default created log file is *log.log*)
- memory limit for a tested process – if this limit is exceeded, dfuzzer logs a warning message into a log file (default memory limit was chosen as three times process initial memory size)
- maximum buffer size for generated strings (default maximum size is 50000 bytes)
- flag to launch process after crash – if tested process crashes during fuzzing and this option is set, crashed process will be launched again and testing will continue

¹pid is process identification number

When dfuzzer is launched it creates a connection to the session bus daemon by calling the method `g_bus_get_sync()` which returns a `GDBusConnection` object containing all information about the connection including unique name assigned to dfuzzer from the session bus daemon. Then a specified process bus name, an object path and an interface are used in call of the function `g_dbus_proxy_new_sync()` to create a `GDBusProxy` object which will be used for communication with a tested process. Communication of dfuzzer with a tested process is shown on the figure 4.1. After connecting to the session bus daemon, dfuzzer requests a tested process identifier for a specified process bus name by calling the method `org.freedesktop.DBus.GetConnectionUnixProcessID`. A process identifier is used for monitoring a tested process status.

Figure 4.1: dfuzzer communication with *GNOME Shell* [14] through the bus daemon



Before fuzz testing of a specified process, object introspection must be done first. The interface `org.freedesktop.DBus.Introspectable` contains the `Introspect` method which can be called using a `GDBusProxy` object to get an XML file containing all of the interfaces and the methods of a process specified by a `GDBusProxy` object. Returned XML file is in string format and so it is parsed into a `GDBusNodeInfo` structure which can be used to find a particular interface. Interfaces are stored in a list of `GDBusInterfaceInfo` structures which contain lists of methods (`GDBusMethodInfo` structure) and to each method there is a list of its arguments (`GDBusArgInfo` structure). The *introspection module* which is part of dfuzzer encapsulates all object introspection details and provides an interface for easy iteration through methods and their arguments.

Every method argument signature is specified by its direction which is either “out” (returned argument signature) or “in” (accepted argument signature). When an object introspection of a specified interface is done, dfuzzer iterates through interface methods. For every method, its accepted arguments signatures (with direction “in”) and a method name are passed to the *fuzz module*. Fuzz testing of a method passed to the *fuzz module* takes place in the cycle by calling this method many times, but with different arguments. Before each call of a method, its arguments must be generated first. Arguments are generated by calling functions from the *random module* according to argument signature. Generated arguments are stored in `GVariant` types. A `GVariant` is used instead of the C `union` type as any data described by a signature (3.1) can be stored inside it. A `GVariant` is created by calling the function `g_variant_new()` which can be think of as an analogue to the `sprintf()` function. The `g_variant_new()` function takes a format string containing signatures (3.1) as its first argument and remaining arguments of the function are sources of data according to a format string.

Before calling a method, a `GVariant` type containing structure with all arguments must be created. A function call to create this `GVariant` must be constructed dynamically, during runtime of dfuzzer as every method has different number and type of arguments. The `libffi` library is used for this dynamic function call construction. After a `GVariant` containing structure with all arguments is created, a method can be called through a `GDBusProxy` object using the `g_dbus_proxy_call_sync()` function. A method is called synchronously because when error occurs, it returns a `NULL` and so dfuzzer knows that well-known name is no longer on the bus daemon or no response returned after timeout (even if a method has no return value, a “method return” message is still sent to the caller). When the `g_dbus_proxy_call_sync()` function returns `NULL` on a method call, dfuzzer looks into a tested process status file `/proc/pid/status` to be sure that a tested process really crashed. If not, it means that a tested process spent a long time processing the data received from dfuzzer and so it did not responded to the method call from dfuzzer in given timeout. When dfuzzer finds out that a tested process did not crash, testing is continuing.

In the case a tested process crashed or exceeded a specified memory limit, a log is added to a log file describing a specified event. A log file entry is a name of a method with its arguments. If some event occurred during testing a method, a log is created within a method entry. Log headers are written in brackets with a method name as prefix and have a serial number (`[method_name LOG 1]`, `[method_name LOG 2]`, etc.). After a log header there is a message describing an event which occurred followed by a process memory size. The last items in log are the inputs on which an event occurred and so the arguments signatures with the corresponding values. An example of the truncated log file from fuzz testing the *GNOME Shell 3.6.3.1* [14]:

```

=====
fuzzing method Eval(s):

end of fuzzing of method 'Eval'
=====

fuzzing method ScreenshotArea(i, i, i, i, b, s):
[ScreenshotArea LOG 1]
  process disconnected from D-Bus
  last known process memory size: [289028 kB]
  on input:
  --i-- '2147483647'
  --i-- '2147483647'
  --i-- '2147483647'
  --i-- '-2147483648'
  --b-- 'true'
  --s [length: 67 B]-- '}>;Vh1C)␣H-C}zF>\!550d-%49!Nax;_4S3|@W$>|1aw)%4e#
Iz4/%9@;7l|0%]BXm?'

end of fuzzing of method 'ScreenshotArea'
=====

fuzzing method ScreenshotWindow(b, b, b, s):

end of fuzzing of method 'ScreenshotWindow'
=====
...

```

The *random module* provides functions for generation of all basic data types (stated in table 3.1) except arrays, structures and dictionaries which were not implemented. When generating numbers, the `rand()` and `random()` functions are used. Besides pseudo-random numbers, the *random module* also generates a specific boundary values of number types (also negative for signed number types). As dfuzzer uses GDBus binding, the functions in the *random module* generate only valid UTF-8 strings. Pseudo-random characters which are used to fill strings are counted to fit the printable range:

$$\text{rand()} \% (127 - 32) + 32$$

The *random module* also provides the NULL terminated array of strings, which will be sent to a tested process if it has any string parameters. Tester can include any valid UTF-8 strings inside this array including shell commands, bus names, object paths, interfaces or format strings.

To test dfuzzer functionality during an implementation, a test server was implemented which exports a well-known name on the session bus daemon with one own interface and one method. Source codes of dfuzzer including the test server which was used for its testing can be found on <https://github.com/matusmarhefka/dfuzzer>.

Chapter 5

Fuzz testing results

dfuzzer was used to fuzz test three chosen applications which use D-Bus system for inter-process communication:

- *GNOME Shell* [14] – the core user interface of the GNOME desktop environment providing basic functionality like switching between windows and launching applications
- *IMSettings* [18] – a framework that delivers Input Method settings and applies the changes immediately, so it will takes an effect without restarting applications and the desktop
- *Evince* [15] – a document viewer for multiple document formats

Each application was fuzz tested twice. The results of testing can be found in log files on the CD.

5.1 Fuzz testing *Evince*

dfuzzer was fuzz testing methods of:

```
bus name:  org.gnome.evince.Daemon
object path: /org/gnome/evince/Daemon
interface: org.gnome.evince.Daemon
```

The first and the second fuzz tests results of the tool did not find any non-standard behaviour of the *Evince* document viewer.

5.2 Fuzz testing *GNOME Shell*

Fuzz tests results of the *GNOME Shell* are presented in the tables 5.1 and 5.2. dfuzzer was fuzz testing methods of:

```
bus name:  org.gnome.Shell
object path: /org/gnome/Shell
interface: org.gnome.Shell
```

The first fuzz tests of the *GNOME Shell* took approximately 4 minutes and 40 seconds to complete. The second fuzz tests were not finished completely because the *GNOME Shell* could not even reload from the crash.

Table 5.1: Results of first fuzz tests of the *GNOME Shell*

Method name	Event	Memory size [kB]
Eval(s)	–	standard
ScreenshotArea(i, i, i, i, b, s)	process disconnected from D-Bus	146160
ScreenshotWindow(b, b, b, s)	–	standard
Screenshot(b, b, s)	–	standard
FlashArea(i, i, i, i)	–	standard

The first fuzz tests results (in table 5.1) show that a method `ScreenshotArea` caused a crash of the *GNOME Shell*. The input arguments were:

`ScreenshotArea(i, i, i, i, b, s):`

```
--i-- '2147483647'
--i-- '2147483647'
--i-- '2147483647'
--i-- '-2147483648'
--b-- 'false'
--s [length: 114 B]-- '\fR3~,jc5?\_FzmxB%Na0?HmnW)3e+LBq~Rn,=R>]/z
!(iyJnE);dQ*P'0c1;0,*-_x6|HqwULX]60%"t/;V'E(dRkQmy^w';n:87kmi8CDLsJNeVi'
```

Other methods did not cause any non-standard behaviour.

Table 5.2: Results of second fuzz tests of the *GNOME Shell*

Method name	Log message	Memory size [kB]
Eval(s)	–	standard
ScreenshotArea(i, i, i, i, b, s)	process disconnected from D-Bus	287300
ScreenshotWindow(b, b, b, s)	process disconnected from D-Bus	58040

The second fuzz tests (in table 5.2) crashed the *GNOME Shell* again with the following input arguments of the method `ScreenshotArea`:

`ScreenshotArea(i, i, i, i, b, s):`

```
--i-- '-2147483648'
--i-- '2147483647'
--i-- '2147483647'
--i-- '2147483647'
--b-- 'true'
--s [length: 164 B]-- 'Pqbnvwe2FJY4UQ_E2Ei!0 g"\0i"TTk%'Kp}CV-j}c~
S2\vd" 'E0]-N::8:nm&ptQaoq5|Y4'Vdo0[4/9V[v'+2zCLi10Z#}9roN1G_JzE:++!N:
;%v/b">]BgDQ7{T55E#d2GD]BfwMhDd!F[0*Z0d}7*L0&!a<Ca}V)'
```

Compared to the first fuzz tests, one more serious crash of the *GNOME Shell* on a call of the method `ScreenshotWindow` was found. The crash happened immediately after the *GNOME Shell* restart which was caused by the previous call of the `ScreenshotArea` method. The method `ScreenshotWindow` was called with these input arguments:

```
ScreenshotWindow(b, b, b, s):
  --b-- 'false'
  --b-- 'true'
  --b-- 'false'
  --s [length: 77 B]-- 'G1C!HhBR}Id{mEU~C(8\tR)= '~. |6%'Z6" [^j].eGr_5
851X;I60x<N}8YzN^Y)t[aPC_~*'Of:d{'
```

and caused a crash of the *GNOME Shell* from which it could not reload and only offered a logout option.

5.3 Fuzz testing *IMSettings*

Fuzz tests results of the *IMSettings* are presented in the tables 5.3 and 5.4. dfuzzer was fuzz testing methods of:

```
bus name: com.redhat.imsettings
object path: /com/redhat/imsettings
interface: com.redhat.imsettings
```

Table 5.3: Results of first fuzz tests of the *IMSettings*

Method name	Log message	Initial mem. [kB]	Current mem. [kB]
<code>GetInfoVariants(s)</code>	memory size exceeded set limit	4000	36044
<code>GetInfoVariant(s, s)</code>	process disconnected from D-Bus	4000	83604
<code>GetUserIM(s)</code>	memory size exceeded set limit	3604	32580
<code>GetSystemIM(s)</code>	memory size exceeded set limit	3604	97408
<code>IsSystemDefault(s, s)</code>	memory size exceeded set limit	3604	144064
<code>IsUserDefault(s, s)</code>	memory size exceeded set limit	3604	178972
<code>IsXIM(s, s)</code>	process disconnected from D-Bus	3604	214936
<code>SwitchIM(s, s, b)</code>	process disconnected from D-Bus	4000	4060
<code>LoadModule(s)</code>	–	standard	standard
<code>UnloadModule(s)</code>	–	standard	standard

The first fuzz tests results indicate that the *IMSettings* daemon is leaking memory as can be seen from the table 5.3. dfuzzer also registered three crashes of the *IMSettings* daemon. After few calls of the method `GetInfoVariants` the *IMSettings* daemon memory size reached the 36044 kB (the initial memory size was 4000 kB). The first crash occurred on a call of the method `GetInfoVariant` with the size of memory 83604 kB. The fuzz tests of the next methods to the `IsUserDefault` method confirm memory leaks in the *IMSettings* daemon (memory size raised from the 3604 kB to the 178972 kB). The last two crashes of the *IMSettings* daemon are caused by the `IsXIM` and the `SwitchIM` methods. The inputs

which are responsible for the memory leaks and the crashes of the *IMSettings* daemon were long strings, usually with size of several hundreds of bytes. These inputs can be found in log files on CD.

The second fuzz tests (stated in table 5.4) confirmed the results of the first tests. Each of the two fuzz tests took approximately 8 minutes to complete.

Table 5.4: Results of second fuzz tests of the *IMSettings*

Method name	Log message	Initial mem. [kB]	Current mem. [kB]
GetInfoVariants(s)	memory size exceeded set limit	3652	32924
GetInfoVariant(s, s)	process disconnected from D-Bus	3652	82832
GetUserIM(s)	memory size exceeded set limit	3600	32500
GetSystemIM(s)	memory size exceeded set limit	3600	97388
IsSystemDefault(s, s)	memory size exceeded set limit	3600	145408
IsUserDefault(s, s)	memory size exceeded set limit	3600	177340
IsXIM(s, s)	process disconnected from D-Bus	3600	212400
SwitchIM(s, s, b)	process disconnected from D-Bus	3645	3852
LoadModule(s)	–	standard	standard
UnloadModule(s)	–	standard	standard

Chapter 6

Conclusion

The goal of this work was to use the fuzzing for testing applications using D-Bus communication system and an automation of this task. The work describes existing fuzzers for individual testing targets and also D-Bus communication system including its architecture. Subsequently, the tool architecture and implementation are discussed followed by the tests results produced by the implemented tool.

The implemented tool (dfuzzer) was used to test three chosen applications – *GNOME Shell*, *IMSettings* and *Evince*. An object introspection was used to allow dfuzzer to “study” application object interfaces, so the tool can be used to test all the applications connected to the session bus daemon besides ones chosen for the testing in this work. The only limitation in this fuzz testing automation is that some functions containing more complex arguments (like structures, arrays of types and dictionaries) are impossible to test with dfuzzer, because the author did not manage to find the appropriate automated method for generation of such complex types during the tool implementation. The current implementation of dfuzzer skips the functions containing complex types (example of a complex type may be an array of structures containing an array of strings (“a(as)”). The absence of complex types generation should be eliminated in a further development of the project.

The testing of chosen applications revealed bugs which were reported to the developers of the applications. The most bugs have been found in the *IMSettings* daemon which suffered from the crashes and memory leaks. These bugs were reported on the *IMSettings Issues* page¹, but they have not been confirmed yet. The *GNOME Shell* tests also revealed bugs. One test case caused the *GNOME Shell* to crash, but it reloaded and testing continued. The second test case on the other hand crashed the *GNOME Shell* immediately after the first test case and caused that it was unable to reload, only logout option was offered. The approach of the GNOME developers to the reported bugs² was lax. One of them commented that the values given to the tested methods are not valid, so it is almost legit the *GNOME Shell* crashes in this case and that the tested methods are in a private interface used only by certain GNOME applications. These facts may be true, but still letting the application to crash is not a good programming practice as one of the developers also admitted. The decision whether the GNOME developers correct these bugs and make the *GNOME Shell* more robust is anyway on them. This work is not going to discuss or

¹<https://bitbucket.org/tagoh/imsettings/issue/1/imsettings-d-bus-interface>

²https://bugzilla.gnome.org/show_bug.cgi?id=699752

disprove the GNOME developers opinions, its goal was just to pinpoint such cases that force applications to behave incorrectly.

The implemented fuzzer is not bound to any specific application. It targets all the applications connected to the session bus daemon, and so its effectiveness might not be the same as for a fuzzer targeting a specific application. For the future, it would be good to add a generation of complex argument types, extend dfuzzer to work also for the system bus daemon applications testing and also include dfuzzer into some application test suites.

Bibliography

- [1] Dave Aitel. Immunity Free software [online]. URL: <http://www.immunitysec.com/resources-freesoftware.shtml>, 2002 [cit. 2013-04-08].
- [2] Ari Takanen and Jared DeMott and Charlie Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House Print on Demand, 2008. ISBN 1596932147.
- [3] Michael Eddington. Peach fuzzing platform [online]. URL: <http://peachfuzzer.com/>, 2013 [cit. 2013-04-08].
- [4] Red Hat and the community. Software/dbus [online]. URL: <http://www.freedesktop.org/wiki/Software/dbus>, 2012-08-24 [cit. 2013-03-22].
- [5] Pranay Kanwar. clfuzz [online]. URL: <http://packetstormsecurity.com/files/author/4866/>, 2006-04-12 [cit. 2013-04-08].
- [6] Michael Sutton and Adam Greene and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007. ISBN 0321446119.
- [7] University of Oulu. Protos - security testing of protocol implementations [online]. URL: <https://www.ee.oulu.fi/research/ouspg/Protos>, 2007 [cit. 2013-04-07].
- [8] OWASP. Owasp WebScarab Project [online]. URL: <https://www.owasp.org/index.php/Webscarab>, 2013 [cit. 2013-04-09].
- [9] John Palmieri. Get on D-Bus [online]. URL: <http://www.redhat.com/magazine/003jan05/features/dbus/>, 2005 [cit. 2013-03-23].
- [10] Havoc Pennington, Anders Carlsson, Alexander Larsson, Sven Herzberg, Simon McVittie, and David Zeuthen. D-Bus Specification [online]. URL: <http://dbus.freedesktop.org/doc/dbus-specification.html>, 2013-02-22 [cit. 2013-03-23].
- [11] Havoc Pennington, David Wheeler, John Palmieri, and Colin Walters. D-Bus Tutorial [online]. URL: <http://dbus.freedesktop.org/doc/dbus-tutorial.html>, [cit. 2013-03-22].
- [12] The GNOME Project. Highlevel D-Bus Support [online]. URL: <https://developer.gnome.org/gio/stable/gdbus-convenience.html>, 2012 [cit. 2013-04-02].

- [13] The GNOME Project. Lowlevel D-Bus Support [online]. URL: <https://developer.gnome.org/gio/stable/gdbus-lowlevel.html>, 2012 [cit. 2013-04-02].
- [14] The GNOME Project. GNOME Shell. URL: <https://live.gnome.org/GnomeShell>, 2013-03-13 [cit. 2013-05-04].
- [15] The GNOME Project. Evince. URL: <http://projects.gnome.org/evince/>, 2013-05-04 [cit. 2013-05-04].
- [16] Michael Sutton. iDefense security intelligence services [online]. URL: http://www.verisigninc.com/en_US/products-and-services/network-intelligence-availability/idefense/index.xhtml, 2006-06-15 [cit. 2013-04-07].
- [17] Michael Sutton, Adam Greene, and Pedram Amini. Fuzzing: Brute force vulnerability discovery – software from the book [online]. URL: <http://www.fuzzing.org/>, 2007 [cit. 2013-04-01].
- [18] Akira Tagoh. IMSettings. URL: <http://tagoh.bitbucket.org/imsettings/>, 2013-04-04 [cit. 2013-05-04].