

# RFID FUZZING ON THE COMMAND LAYER

## Abstract

RFID Fuzzing can prove to be a valuable tool in RFID security testing. By creating an automated tool that can test the robustness of an RFID system we can see if RFID hard- and middleware is capable of handling a large stream of malformed information and thus test how reliable the system is. This thesis focuses on command layer fuzzing, and the implementation of it inside a module of the well known RFID security program RFID Guardian. Being a work in progress, this thesis also gives pointers for further research in the RFID fuzzing field.

## 1 Introduction

Radio Frequency Identification (RFID) is becoming more and more known and popular each day, with big and widely used implementations like the OV chipkaart and passport watermarking. With this wide usage, security and integrity become a large issue. Recent hacks like the OV chipkaart cloning have shown that the RFID standards of today aren't as secure as they ought to be. While the security of the protocol itself is one thing on its own, the main flaws of RFID security lie in the middleware software which uses RFID hardware and assumes that the data coming from the reader and tags are correct, confidential and integrate. Sadly, we see the opposite in practice.

Fuzz testing or fuzzing is a software testing technique that provides random data ("fuzz") to the inputs of a program. RFID Fuzzing is a way of testing the security of the RFID system by using random data to try and break the middleware. This method is based on forging responses from tags instead of letting real tags respond in an ordinary way. With this method we can check whether the middleware is capable of handling incorrectly formatted responses, id's and payloads. If we would construct a program that iteratively tries all possible responses, both correct and incorrect, we would have created the perfect and flawless fuzzer. While this sounds pretty good, it isn't cost and time-effective to create such a program for it has to forge and test all possible responses which can take a very long time.

This is exactly why we have to construct a clever piece of code that will try to forge a request that seems valid to the reader, but contains incorrect information and payload. A lot of research on creating custom responses has already been done. The RFID Guardian Project<sup>1</sup> is a Vrije Universiteit project about providing security and privacy in RFID technology. Amongst many other functions, this software enables us to simulate RFID readers and tags through software. Andrew Richardson[1] started the fuzzing project inside the Guardian software in 2007 and provided me with the fuzzing structure and guidelines for the fuzzing tool.

The objective of this project will be to design and implement fuzzing code into the RFID Guardian software and make it usable for further research. Before explaining any written code, we have to get proper understanding of the RFID protocol. The ISO-15692[2] standard clearly describes how RFID requests and responses are formatted. After defining the terms, I will describe what procedures I followed while researching RFID fuzzing on the command layer. After this I will show the results from the real hardware tests that I have achieved and try to give directions for further research.

---

<sup>1</sup>more information can be found on <http://www.rfidguardian.org>

## 2 RFID Fuzzing layers

The RFID system consists of RFID readers, RFID tags and middleware. An RFID tag is basically an integrated circuit with an antenna which can respond to a RFID reader's requests using radio frequency. The middleware is the software that will use the information acquired from the readers and tags. To break the RFID fuzzing into manageable parts, Andrew Richardson[1] separated the transmission between the RFID tags and the middleware in several layers as seen in figure 1.

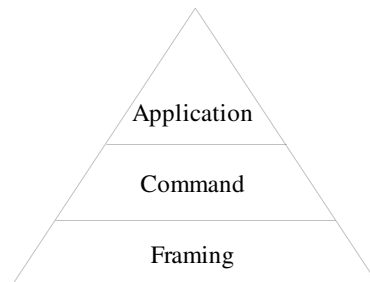


Figure 1: RFID Layering

First we have got the framing layer. This is the lowest layer in which the bits get transmitted through the air using radio waves. The format of these transmissions is strictly defined in the ISO-15693 standard [2]. However, the strictness of the interpretation is still hardware specific. This means that there are some parameters that are variable, and are implemented in different ways in different readers. These parameters can be fuzzed as well but this level of fuzzing is beyond the scope of this project.

The next layer is the command layer. This layer handles the requests and responses structure, as describe in the ISO standard. The fields in each request and response are variable, which means they can contain any value we assign to them. The corresponding valid responses to a certain request are described in the standard, but the RFID reader hardware should be able to process incorrect responses as well.

The third, highest layer is called the application layer. This layer represents the middleware application which is using the RFID information coming from the reader hardware and tags. In this layer, tags have been correctly checked, read and interpreted. When forging responses, it could be possible to alter the payload of a valid response to create attacks on the middleware.

### 2.1 Requests



Figure 2: General request format

If we want to fuzz on the aforementioned command layer, we need to get a proper view on how requests and responses are formatted. The request frame is composed of several fields, as can be seen in figure 2, starting with a "Start of frame" (SOF) and ending with an "End of frame" (EOF) delimiter.

The first field contains 1 byte worth of flags, in which information is stored about the amount of sub-carrier frequencies used by the tags, the data rate and some information about which tags should respond to this request. The second field is the 8 bit command code, which instructs the tag with a command such as "inventory" to gain information about the tag or "stay quiet" to make sure some or more tags do not respond. The most commonly used command is the inventory command, which will be used in the fuzzing setup. There are several other commands which can be sent as well, with corresponding responses which can be fuzzed but this is outside the scope of my project.

Next we have the third and fourth fields. These fields can be either a parameter field, a data field or a combination of these two. The structure of these fields depends on the command that is issued. In

the case of an inventory command the number of significant bits in the data field is defined in an 8 bit parameter field, while the stay quiet command uses this allotted space to address a specific “Unique Identifier” (UID). The last 2 byte long data field is a “Cyclic Redundancy Check” (CRC) which contains the checksum for the fields preceding this one. This field gives tag-readers the opportunity to quickly check whether the sent information was correct.

## 2.2 Responses

SOF	Flags	Parameters	Data	CRC	EOF
-----	-------	------------	------	-----	-----

Figure 3: General response format

The responses that the tags send back after receiving a command are the more interesting frames for our fuzzing research. Like the requests, the possible responses to certain requests are strictly defined in the ISO-15692 standard. These responses consist of several fields, starting with a SOF frame and trailing with an EOF frame, as can be seen in figure 3. In this case, I will assume the request was an inventory request because this command is the most common and we will use this structure in our fuzzing tool as well.

The first variable field in the response frame is the 8 bit long response flags field. This field's main use is the error flag, which specifies whether or not an error was detected by the tag. The rest of the field contains a lot of reserved bits for future use. Like the requests, the rest of the response depends on the kind of command that gets sent from the reader. In this case, the second field is the 8 bit long “Data Storage Format Identifier” (DSFID) field which indicates how the following data is structured. Next up is the same UID field we saw in the request structure, though this time it specifies the unique identifier of the responding tag. Finally a CRC check is done over the aforementioned fields resulting in a 16 bit check value.

## 3 Fuzzing on the command layer

Fuzzing the command layer can be done in several ways. I have decided to restrict this project to sending iterated responses to a reader and see if the reader crashes on a certain response. This means that I will keep track of the current state of fuzzing and exhaustively try out sending every possible response to the reader. Because there are a lot of possible responses it would be helpful to be able to choose which fields will be iterated. Using a couple of options that you can toggle on and off you should be able to fuzz multiple fields at once.

A second way of testing could be done by changing the nature of the iteration. Instead of testing each possible response, you could also skip a certain number of responses, for example testing the UID 0x00000000 first, skipping the next three and then testing 0x00000004. This would greatly reduce the amount of responses that have to be tested, but could possibly skip that one specific response that would crash the system. A third way of iterating could be the random way, in which we generate a random response every time. These responses would be difficult to track, for we can't trace back which response corresponded with each state. A solution to this could be to use a one way hash function to produce the “random” responses.

Another way of fuzzing RFID could be by altering the requests instead of the responses. If there would be a callback after the request is constructed, like the one I am using while altering the responses, it shouldn't take a lot of time to port my code to the request field callback. A structurally different way of fuzz testing could be by manually altering the timing of the radiowaves that are sent in the responses to confuse the reader, as done in the bachelor project by Sven Gude[3]. For now I will stick with the fuzzing of the command layer, using the existing software package RFID Guardian.

### 3.1 RFID Guardian

The RFID Guardian software makes for a perfect platform to write RFID fuzzing code, because all of the RFID specific calls are already implemented. The RFID Guardian Fuzzing Project Status Report

by Andrew Richardson [1] describes that there is already some fuzzing code written in the RFID Guardian program. After discussing the possible options for programming with Rutger Hofman, the main programmer of the RFID Guardian, we decided that the most feasible way of writing fuzzing code for RFID is to hook into the `mrg_fuzzing_raw.c` file.

The idea behind the fuzzing hook is that the guardian can imitate or spoof a tag. The guardian does this by reading incoming requests from RFID readers, and then constructs a response and passes it to the device driver that will send the response out to the reader. As can be seen in appendix C, the `fuzz_raw_callback_pre` is a function that gets called after the valid response has been constructed, to be able to alter the “raw” byte stream of the response. After this function, the raw stream gets passed to the device driver. In this call we can alter the byte stream at our hearts content, giving us the opportunity to iteratively change the response.

## 3.2 Coding approach

To iteratively change the responses in an organized and structured way I decided to use and modify the existing `MRG_FUZZING` fuzzing struct, as seen in appendix D. This struct gets created in the Guardian whenever the command line parameter `-fuzz` or when the Guardian is configured using the `-fuzzing` parameter. The struct contains some fuzzing options already as well as some integers to toggle some fuzzing functions on or off. One of the existing toggles is the integer `CMD_change_en` that toggles a function which changes the request type and an integer `CRC_break_en` which will toggle a function which breaks the CRC value manually in the raw response. My code will try to improve upon the last function, creating a more advanced response altering method.

## 3.3 Fuzzing options

I have created some extra options in the fuzzing struct to enable some extra functionality in the fuzzing module. One of these options is the toggle `FUZZ_bruteforce` that will toggle my bruteforcing function on or off. By setting this integer to 0 in the code you can run the original guardian code without using my fuzzing functions. After this first general toggle I have created 3 extra options which toggle some special parts of response fuzzing on and off. The first one is the `FUZZ_flags_en` which toggles flag field fuzzing on, the second is the `FUZZ_dsfid_en` which toggles the DSFID field fuzzing and the third is `FUZZ_uid_en` which can toggle UID fuzzing on and off. This means that the response can be altered in multiple ways at the same time. This is an excellent way of scaling the fuzzing attack to try out even more different responses.

## 3.4 Bruteforce state

The final and most important addition to the fuzzing struct is the `FUZZ_bfstate` (“bfstate” is short for bruteforce state). This is an integer that keeps track of the current state of fuzzing. When designing the fuzzing solution I thought about creating a new struct with several counters in it to administrate the current states, but I have chosen to modify the existing struct. I did this because I didn’t want to clutter the project with just another struct which will only make the code more complicated for the next researcher. I’m using this bruteforce integer to update the responses each time, according to the options that were enabled. When a response is constructed and sent, the `bfstate` gets updated with the following code line: (line 120 in `mrg_fuzzing_bruteforce.c`)

```
fuzz->FUZZ_bfstate += 1;
```

The next time the bruteforce function is called, the method checks the `bfstate` again and changes the response correspondingly. In practice this means that the next possible response will be tried out. The aforementioned line can also be edited to skip a certain number of responses, by increasing the value on the right hand side, e.g. using “+=4” to skip 3 responses. This saves a lot of time while testing a broad range of possible responses.

To translate a bruteforce state into a response you have to calculate the binary value of the FUZZ\_bfstate integer and paste it over the field that you are currently fuzzing. For instance when only fuzzing the flag field, fuzzingstate 12 corresponds to the flagbits 00001101. This state says that the fifth, sixth and 8th bit in the flag field are activated. If the current value of the FUZZ\_bfstate would be the integer 13, it would respond to the flagbits 00001110, which corresponds to the flag value in which the fifth, sixth and seventh flag are enabled.

Because the FUZZ\_bfstate gets initialized in the mrg\_fuzzing\_bruteforce.c it is also possible to initialize the bfstate integer to a higher number which gives you the opportunity to continue a previous test or start from a different state. For example, if I am running a UID field only fuzzing test, and I have tested the first 1000 responses without effect, I can resume that session by assigning "1001" to the FUZZ\_bfstate in mrg\_fuzzing\_bruteforce.c to make sure the first response will be constructed with bruteforce state 1001.

### 3.5 Usage

To use my code, we have to download a clean checkout of the Guardian software, as shown in the tutorial on the RFID Guardian website<sup>2</sup>. After this we have to replace the files inside the directory `~\projects\mrg\trunk\src\fuzzing` with the files from my source package. In the file `mrg_fuzzing.c` you can specify which of the options are turned on and off, as specified in section 3.3. The fuzzing code can be executed by configuring and compiling the Guardian package using the commands `./configure -fuzzing` and `make` or by calling the Guardian software simulator program inside the `guardian-main` folder with the command line parameter `-fuzz`.

### 3.6 Simulation and Hardware

For the ease of testing, a simulator environment was created within the RFID Guardian software. The `guardian-main` program is the actual Guardian software itself. By using the command line parameter `-port 'socket tag'` we can mimic the behavior of real Guardian hardware. Instead of using a radio to broadcast the bytes, the requests and responses get streamed onto the socket connection. I have used the standalone `reader-main` module as a reader. By feeding this program the command line parameter `-port 'socket reader'` we can instruct this program to create the socket connection to transmit bytes rather than using the hardware radio.

After I got the fuzzing module working on the simulator I compiled the code for the actual Guardian hardware and flashed the Guardian program onto the hardware. At first it didn't seem to work on the real hardware but that was because of the delay caused by the system calls that were made to print output to the screen. After we turned the verbosity off the program ran fine. At that moment we were able to start the fuzz testing on the Philips RFID reader hardware. My main issue about testing with the real Guardian hardware is that I didn't have time to implement the Guardian logging API into my code. The lack of output while running the real hardware tests makes it hard to keep track of the current state and to detect possible errors in the system.

### 3.7 Test case

When using the simulator program we can simply enable the verbosity function which will produce output for the Guardian program. Same story goes for the reader, but only if we use the simulation environment. There is an API present for logging inside the RFID Guardian software while using the real hardware, but I didn't have the time to investigate this possibility of logging. When using the real Philips RFID reader hardware it is tricky to produce readable logs because the RFID reader software that was supplied with the reader can't output detailed results to screen directly. However, we can use the same reader program used in the simulation environment (`reader-main`) to work with the Philips Pegoda reader hardware. I have done this by using cygwin to compile a cvs checkout of the guardian. This time I used the configure option `./configure -host` which builds the executables needed when using the

<sup>2</sup>the tutorial can be found on [http://www.rfidguardian.org/index.php/Tutorial:\\_installation](http://www.rfidguardian.org/index.php/Tutorial:_installation)

Pegoda reader. To be able to interface with the Pegoda reader, I have used the 3rd party proprietary drivers from Philips which are located on a separate cvs trunk. When the tree was complemented with the 3rd party software we are able to build the same `reader-main` program as before, with the difference that we are using the command line parameter `-port 'Philips Pegoda reader'`. In addition to this changed parameter, we have to run this command using the right environment variables so the corresponding libraries can be detected. When the program is called using the following command, we can basically run the same tests as with the simulator:

```
../../bin/env.bash ./mrg_reader_main.exe -port 'Philips Pegoda reader'
```

### 3.8 Logs

The logfiles<sup>3</sup> produced by the tests are screen dumps coming directly from the `reader-main` program, when running the guardian hardware with the options `FLAG&DSFID` fuzzing turned on. Because the guardian hardware can't display output in realtime the same way as the simulator due to timing reasons, I have submitted the guardian output from the simulator. The verbose output lines I wrote in the software using the `mrg_diag_printf()` method produce the same output when running in the simulated environment and on the real hardware. Because it is not possible to display this information on the real hardware due to timing errors, I have used the Guardian output of the simulator in the logs instead. This could be fixed by using the logging API for the Guardian hardware, which I did not implement in my code. By supplying the outputlogs of the Guardian simulator program I am trying to give a better view on what my code really does, though it does not imply that the results will be the same on the real hardware.

## 4 Results

While running tests on the simulator we are looking for implementation mistakes. For these tests we used the Guardian tools but the next logical step would be to do the same tests with for example Philips software. Though I haven't found any valuable results like a crash, bug or error in the simulator software yet, I did find some errors corresponding to unimplemented functions. In the test case, I ran the following tests on the simulator programs:

- Flag Only
- DSFID Only
- Flag & DSFID
- UID Only

When sending incorrect flag fields and DSFID's the simulator program will still simply display the given UID and discard the rest of the information. I think this is because the reader program just checks the CRC value of the response it gets and then displays the given UID field. I haven't run the complete UID fuzz test because it takes an unreasonable amount of time to check  $2^{64}$  possible UID's. This time can be shortened by increasing the state with a bigger number, as described in section 3.4, but this would then still take a very long time. This is also due to the fact that the timing between the different requests can't be too short. This forces me to take about 1 second between each response before sending the next one. By fuzzing the UID field at random we can cover a wider range of different bits, but this brings up other problems as described earlier in section 3.

The same story goes for the tests on the real RFID Guardian hardware in combination with the Philips Pegoda RFID reader, though the `reader-main` program comes up with more detailed errors when using the real hardware. The software program didn't crash, but it did display some predictable errors because of badly formatted responses. For instance, when the first response was sent (the first

<sup>3</sup>the logs can be found on <http://www.few.vu.nl/~ppeerde/bp>

bruteforce state), the Guardian's verbose output would show the following fuzzing notification if verbosity was turned on:

```
In the raw filter.
[0x01 0x01] 0x00 0x00 0xf4 0xf0 0xf0 0xf0 0x00 0x00 0x00 0x00 0x69 0xfe [0x01 0x07]
In bruteforce function now
stream->FLAGS & DSFID after bruteforcestate 0: 0x00 0x00
response after bruteforcing:
[0x01 0x01] 0x00 0x00 0xf4 0xf0 0xf0 0xf0 0x00 0x00 0x00 0x00 0x69 0xfe [0x01 0x07]
```

after this response was sent we can see the output that the reader gives is normal behavior because it is a valid formatted request without any invalid fields. This produces a simple inventory notification in the verbose output of the reader-main program using the Philips Pegoda reader:

```
[rfid reader]
slot request resp eof d(req) d(resp) d(eof)
[ 0] 5.040261 0.000000 5.042535 0.000000 0.000000 0.000000
After inventory/anticollision, my database is:
0 0x00000000f0f0f0f4
```

Another case is that the same verbose output is given but without the UID popping up on the fifth line in the message. This seems to be popping up now and then, without any pattern whatsoever. This is probably due to the timing between the inventory requests, where the next request either comes too quick or too slow.

The last and most interesting case is when an error message pops up on the reader-main side. This seems to be random as well, because it pops up in different bruteforce states on each test run. It could be possible that, like the second case, timing of the requests is affecting the output. As an example, the error message pops up in bruteforce state 12 when only fuzzing the flag field, using the following response, which would be outputted to the screen by the Guardian program when verbosity is turned on:

```
In the raw filter.
[0x01 0x01] 0x00 0x00 0xf4 0xf0 0xf0 0xf0 0x00 0x00 0x00 0x00 0x69 0xfe [0x01 0x07]
In bruteforce function now
stream->FLAGS after bruteforcestate 12: 0x0C
response after bruteforcing:
[0x01 0x01] 0x0c 0x00 0xf4 0xf0 0xf0 0xf0 0x00 0x00 0x00 0x00 0x46 0xbe [0x01 0x07]
```

This response gives the following error message in the reader-main program using the Philips Pegoda reader, which tells us that the response collided and gives a fatal error because of unimplemented functions:

```
[rfid reader]
mrg_pegoda_rcve.c:237 MRG: unimplemented routine
    "Fill in noncolliding bits in collision response" -- PLEASE IMPLEMENT
mrg_pegoda_rcve.c:248 MRG: unimplemented routine
    "Read valid data, now pack the collision data" -- PLEASE IMPLEMENT
slot request resp eof d(req) d(resp) d(eof)
[ 0] 22.246118 0.000000 22.248391 0.000000 0.000000 0.000000
    [MRG] mrg_rfid_reader_main.c:431 *** Fatal error:
    Inventory failed: mrg_pegoda_rcve.c:223 [0] No error
```

However, this fatal error doesn't crash the program and doesn't make the program leak memory. The pattern of the error seems totally at random, I have seen it occur in bruteforce states 45, 47 and 48 when fuzzing FLAG&DSFID and in bruteforce states 12 and 17 when fuzzing only the FLAG field, but this error seems to occur at random on different states each test run.

## 5 Further research

As Andrew Richardson already mentioned in his report, there is still a lot of work that needs to be done in the command layer and on the other layers of RFID Fuzzing. Next to response fuzzing in this project it could for instance also be possible to fuzz the requests the readers send out. My code can easily be reused in altering requests instead of responses when using a different callback in the software. By improving my code and perhaps the `reader-main` programs code it would be easier to identify which response was responsible for a certain response. This could be done by implementing the available logging API in the RFID Guardian software into the fuzzing code. This can also be done in combination with modifications to the `reader-main` program for instance with sequence numbers for each inventory request or timestamps, though these might be hard to analyze when tests are running in high pace.

While coding and testing I found out that it would be very handy to be able to edit the fuzzing options and bruteforce state I spoke of in section 3.3 on the fly, while running the Guardian program. At the moment you still have to recompile the whole Guardian program after each option change, which makes it very time consuming to run multiple tests. This improved implementation involves altering the UI of the Guardian software which could take a lot of time to dive into, but this would make running multiple tests a whole lot easier.

In my project I have restricted myself to only fuzzing the responses to inventory requests, but a large variety of other commands and responses can be fuzzed as well. In further research we could perhaps fuzz the responses to different requests, or make a universal accepting method that will fuzz the response no matter what the request was. This can be done by extending upon the if statement in `mrg_fuzzing_bruteforce.c` where the incoming request is being checked for the inventory request.

Another helpful task would be to implement the routines that weren't implemented in the reader as seen in section 4. In this way we can make the Guardian's reader module (`reader-main`) more robust for further use and make sure the error doesn't occur on future fuzz testing the RFID Guardian software. Because of the little time I got in this project, I wasn't able to figure out how to use the Guardian logging API to produce real output data. If this possibility would be investigated in further research we could improve the link between the original request and the corresponding response. To further improve this link we could also implement a socket connection between the `reader-main` module and the Guardian program that could improve the communication between the programs while fuzz testing to produce more reliable logs.

The next logical step to continue this research could also be running the same tests on Philips software. While robustness testing our own created reader software is interesting and rewarding, running automated tests on real corporate software could prove companies great value.

## 6 Conclusion

The RFID Guardian software proved to be a very extendible software package. The fuzzing module that was already written was a great start for me to implement the fuzzing code that I have written. With the available verbosity functions and predefined structs I was able to implement extra functionality by just reading and editing the existing sourcecode.

While I haven't succeeded in getting clear results in the means of e.g. a program crash or memory leaks, this project is still a step forward in creating automated tools to test RFID Security. With the log files and verbose output in mind it is possible to trace back the response that could have caused a crash, though it is still hard to link the response to a certain situation. While running the tests on the real hardware I found out that some errors occur in the `reader-main` module, which can potentially be solved in further research making the module more robust in the future.

By expanding the fuzzing module in the Guardian software even more we can create a very valuable tool in robustness testing, not only in our own research environment, but in every existing RFID environment which processes RFID tag responses.

## References

- [1] Andrew Richardson. Rfid guardian fuzzing project status report. emailed from chardson@umich.edu to melanie@few.vu.nl, August 2007. Report sent from Andrew Richardson to Melanie Rieback on RFID Fuzzing.
- [2] Joint Technical Committee ISO/IEC JTC 1. International standard iso/iec 15693-3. Final Draft ISO/IEC FDIS 15693-3, December 2000.
- [3] Sven Gude. Rfid fuzzing on the framing layer. Bachelors Thesis Vrije Universiteit, June 2008.

# APPENDIX

## A mrg\_fuzzing\_bruteforce.c

```

/* Peter Peerdeman Bachelor Project 2008 */
/* Run the iterative external fuzzing function */
/* which will iterate through possible responses each time */
/* this function is called, and fix the crc afterwards */

#include <string.h>
#include <stdlib.h>
#include "mrg_fuzzing.h"
#include "mrg_fuzzing_private.h"
#include "mrg_fuzzing_filter.h"
#include "mrg_fuzzing_bruteforce.h"

/* This creates a macro to write the crc field in the response */
#define MRG_RFID_RAW_WRITE_FIELD_FUZZ(stream, field) \
    do { \
        memcpy(stream->data + stream->append, \
               &(field), \
               sizeof field); \
        stream->append += sizeof field; \
    } while (0)

/* This function will alter the symbol_stream every iteration
 * using the bruteforce state inside the fuzzing struct */
int mrg_fuzzing_bruteforce(mrg_symbol_stream_t *stream,
                          struct MRG_FUZZING *fuzz)
{
    if (fuzz->verbose) {
        mrg_diag_printf("In bruteforce function now\n");
    }
    /* The fuzzing method assumes an inventory request is made! */
    if (fuzz->req_command == MRG_FRAME_15693_INVENTORY) {

/* Here the raw response gets altered using the current
 * fuzz->bruteforcestate, depending on the fuzzing options */

        /* UID FUZZING */
        if(fuzz->FUZZ_uid_en) {
            /* Here the UID gets a new value, depending on current state */
            memcpy((stream->data + stream->append - 11),
                   &(fuzz->FUZZ_bfstate), 8);
            /* stream->data[stream->append - 4] = fuzz->FUZZ_bfstate;*/
            /* show the uid after fuzzing */
            if(fuzz->verbose) {
                mrg_diag_printf("response->UID of raw stream after uid bruteforce: \n
                                0x%02X 0x%02X 0x%02X 0x%02X 0x%02X 0x%02X 0x%02X 0x%02X \n",
                                stream->data[stream->append - 11],
                                stream->data[stream->append - 10],
                                stream->data[stream->append - 9],
                                stream->data[stream->append - 8],
                                stream->data[stream->append - 7],
                                stream->data[stream->append - 6],
                                stream->data[stream->append - 5],
                                stream->data[stream->append - 4]);
            }
        }
    }
}

```

```

}

/* FLAG & DSFID FUZZING */
if(fuzz->FUZZ_flags_en && fuzz->FUZZ_dsfid_en) {
    /* If full DSFID iteration is complete iterate flag 1 time */
    div_t flagcounter;
    flagcounter = div(fuzz->FUZZ_bfstate, 256);
    stream->data[stream->append - 13] = flagcounter.quot;
    stream->data[stream->append - 12] = flagcounter.rem;
    /* display the current state */
    if(fuzz->verbose)
        mrg_diag_printf("stream->FLAGS & DSFID after bruteforcestate %d: 0x%02X 0x%02X \n",
                        fuzz->FUZZ_bfstate,
                        stream->data[stream->append - 13],
                        stream->data[stream->append - 12]);
} else {

/* FLAG only FUZZING */
if(fuzz->FUZZ_flags_en) {
    stream->data[stream->append - 13] = fuzz->FUZZ_bfstate;
    /* display the current flag */
    if(fuzz->verbose) {
        mrg_diag_printf("stream->FLAGS after bruteforce: 0x%02X \n",
                        stream->data[stream->append - 13]);
    }
}

/* DSFID FUZZING */
if(fuzz->FUZZ_dsfid_en) {
    stream->data[stream->append - 12] = fuzz->FUZZ_bfstate;
    /* display the current flag */
    if(fuzz->verbose) {
        mrg_diag_printf("stream->DSFID after bruteforce: 0x%02X \n",
                        stream->data[stream->append - 12]);
    }
}
}

/* Update the CRC bytes */
/* set the append marker back 3 bytes, before old crc */
stream->append -= 3;
mrg_crc_t crc = mrg_crc(stream->data + 1,
                        stream->append - 1);
MRG_RFID_RAW_WRITE_FIELD_FUZZ(stream, crc);
/* set the append marker back to initial, after the EOF */
stream->append += 1;

/* Display whole response after bruteforcing */
if(fuzz->verbose) {
    mrg_diag_printf("response after bruteforcing:\n");
    int i;
    for (i = stream->consume; i < stream->append; i++) {
        if (stream->is_marker[i] != 0) {
            mrg_diag_printf("[0x%02x ", stream->is_marker[i]);
        }
        mrg_diag_printf("0x%02x", stream->data[i]);
        if (stream->is_marker[i] != 0) {
            mrg_diag_printf("] ");
        } else {
            mrg_diag_printf(" ");
        }
    }
}

```

```

        }
    }
    mrg_diag_printf("\n");
}

/* Increment the fuzzstate with 1 each iteration */
    fuzz->FUZZ_bfstate+=1;
}
return(0);
}

```

## B mrg\_fuzzing\_bruteforce.h

```

/* Peter Peerdeman Bachelor Project 2008 */

#ifndef MRG_FUZZING_BRUTEFORCE_H__
#define MRG_FUZZING_BRUTEFORCE_H__
#include "mrg_fuzzing.h"
#include "mrg_symbol_stream.h"

/** Start bruteforcing. */
int mrg_fuzzing_bruteforce(mrg_symbol_stream_t *stream,
                          mrg_fuzzing_t *fuzz);
#endif

```

## C mrg\_fuzzing\_raw.c

```

/*
 * Copyright 2007, Rutger Hofman, VU Amsterdam
 *
 * This file is part of the RFID Guardian Project Software.
 *
 * The RFID Guardian Project Software is free software: you can redistribute
 * it and/or modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation, version 3 of the License.
 *
 * The RFID Guardian Project Software is distributed in the hope that it will
 * be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with The RFID Guardian Project Software. If not, see
 * <http://www.gnu.org/licenses/>.
 */

#include <string.h>

#include "mrg_util.h"

#include "mrg_fuzzing.h"
#include "mrg_port.h"

#include "mrg_fuzzing_private.h"
#include "mrg_fuzzing_raw.h"

#include "mrg_fuzzing_bruteforce.h"

```

```

/*
 * See #mrg_port_raw_callback_t in include/mrg_port.h for a description
 * of the callback function and how to use it.
 *
 * @param port
 * @param c_stream this must be const because all the write stream parameters
 *       have const. If you want to break your 'const' promise, just cast the
 *       thing to a 'mrg_symbol_stream_t *' without const.
 * @param arg
 */
static mrg_port_raw_callback_result_t
fuzz_raw_callback_pre(mrg_port_t *port,
                     const mrg_symbol_stream_t *c_stream,
                     void *arg)
{
    mrg_symbol_stream_t *stream = (mrg_symbol_stream_t *)c_stream;
    mrg_fuzzing_t *fuzz = arg;
    size_t i;

    /*
     * Modify stream->data[i] for stream->consume <= i < stream->append,
     * unless stream->is_marker[i] is set.
     */

    /* Print raw frame buffer if verbosity is enabled */
    if (fuzz->verbose) {
        mrg_diag_printf("In the raw filter.\n");
        for (i = stream->consume; i < stream->append; i++) {
            if (stream->is_marker[i] != 0) {
                mrg_diag_printf("[0x%02x ", stream->is_marker[i]);
            }
            mrg_diag_printf("0x%02x", stream->data[i]);
            if (stream->is_marker[i] != 0) {
                mrg_diag_printf("] ");
            } else {
                mrg_diag_printf(" ");
            }
        }
        mrg_diag_printf("\n");
    }

    /* Peter Peerdeman Bachelor Project 2008 */
    /* Run the iterative external fuzzing function */
    /* which will iterate through possible responses each time */
    /* this function is called*/

    if (fuzz->FUZZ_bruteforce) {
        mrg_fuzzing_bruteforce(stream, fuzz);
    }

    /* Mangle the CRC bits of the raw frame if enabled */
    /* Stream is the datastream containing the raw response */
    /* Consume and Append describe the relevant endpoints of the stream */

    if (fuzz->CRC_break_en) {
        if (fuzz->req_command == MRG_FRAME_15693_GET_SYSTEM_INFORMATION) {
            if (stream->is_marker[stream->append -3] == 0 &&
                stream->is_marker[stream->append -2] == 0) {
                if (fuzz->verbose)
                    mrg_diag_printf("stream->data CRC of raw stream
                                     before mangling: 0x%02X 0x%02X\n",

```

```

        stream->data[stream->append - 3],
        stream->data[stream->append - 2]);
stream->data[stream->append - 3] ^= 0xF3;
stream->data[stream->append - 2] ^= 0x8F;

    if (fuzz->verbose)
        mrg_diag_printf("stream->data CRC of raw stream
            after mangling: 0x%02X 0x%02X\n",
            stream->data[stream->append - 3],
            stream->data[stream->append - 2]);
    }
}
}

return MRG_PORT_RAW_CALLBACK_CONTINUE;
}

static mrg_port_raw_callback_result_t
fuzz_raw_callback_post(mrg_port_t *port,
    const mrg_symbol_stream_t *c_stream,
    void *arg)
{
    mrg_symbol_stream_t *stream = (mrg_symbol_stream_t *)c_stream;
    size_t i;
    mrg_fuzzing_t *fuzz = arg;

    /* Example logging from the raw filter; do this /after/ the send */
    if (fuzz->log != NULL && fuzz->enabled) {
        mrg_log_command_t command = MRG_LOG_COMMAND_DATA;
        size_t size = 0;
        char *b = fuzz->buffer;

        mrg_log_write(fuzz->log, &command, sizeof command);
        for (i = stream->consume; i < stream->append; i++) {
            if (stream->is_marker[i] != 0) {
                snprintf(b, fuzz->buffer_size - size, "[0x%02x ",
                    stream->is_marker[i]);
                b = strchr(b, '\\0');
                size = b - (char *)fuzz->buffer;
            }
            snprintf(b, fuzz->buffer_size - size, "0x%02x", stream->data[i]);
            b = strchr(b, '\\0');
            size = b - (char *)fuzz->buffer;
            if (stream->is_marker[i] != 0) {
                snprintf(b, fuzz->buffer_size - size, "] ");
            } else {
                snprintf(b, fuzz->buffer_size - size, " ");
            }
            b = strchr(b, '\\0');
            size = b - (char *)fuzz->buffer;
        }
        mrg_log_write(fuzz->log, &size, sizeof size);
        mrg_log_write(fuzz->log, fuzz->buffer, size);
    }

    return MRG_PORT_RAW_CALLBACK_CONTINUE;
}

```

```

int
mrg_fuzzing_raw_start(mrg_port_t *port, mrg_fuzzing_t *fuzz)
{
    fuzz->raw_port = port;

    fuzz->buffer_size = 5 * sizeof(mrg_frame_response_t) + 6 * 2 + 1;
    fuzz->buffer = mrg_malloc(fuzz->buffer_size);
    if (fuzz->buffer == NULL) {
        return -1;
    }

    if (fuzz->lock == NULL) {
        fuzz->lock = mrg_lock_create();
        if (fuzz->lock == NULL) {
            mrg_free(fuzz->buffer);
            return -1;
        }
    }

    fuzz->enabled = 1;

    if (mrg_port_raw_callback_register(port,
                                      MRG_RFID_13_56_15693,
                                      fuzz_raw_callback_pre,
                                      fuzz,
                                      MRG_PORT_RAW_CALLBACK_PRE) == -1) {
        return -1;
    }

    if (mrg_port_raw_callback_register(port,
                                      MRG_RFID_13_56_15693,
                                      fuzz_raw_callback_post,
                                      fuzz,
                                      MRG_PORT_RAW_CALLBACK_POST) == -1) {
        return -1;
    }

    return 0;
}

```

```

int
mrg_fuzzing_raw_init(int *argc, char *argv[])
{
    return 0;
}

```

```

int
mrg_fuzzing_raw_end(void)
{
    return 0;
}

```

## D mrg\_fuzzing\_private.h

```

/*
 * Copyright 2007, Rutger Hofman, VU Amsterdam
 */

```

```

* This file is part of the RFID Guardian Project Software.
*
* The RFID Guardian Project Software is free software: you can redistribute
* it and/or modify it under the terms of the GNU General Public License as
* published by the Free Software Foundation, version 3 of the License.
*
* The RFID Guardian Project Software is distributed in the hope that it will
* be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with The RFID Guardian Project Software. If not, see
* <http://www.gnu.org/licenses/>.
*/

#ifndef MRG_FUZZING_PRIVATE_H__
#define MRG_FUZZING_PRIVATE_H__

#include <stddef.h>

#include "mrg_os.h"
#include "mrg_fuzzing.h"
#include "rfid-filter/mrg_rfid_filter_spoof.h"

struct MRG_FUZZING {
    int            enabled;

    mrg_lock_t    *lock;
    mrg_port_t    *filter_port;
    mrg_port_t    *raw_port;

    mrg_log_t     *log;
    mrg_spoof_t   *spoof;

    size_t        buffer_size;
    char          *buffer;

    /* Any other fields that are needed for a fuzzer */
    int            CMD_change_en;    /**< change request type (command) */
    int            CRC_break_en;     /**< mangle CRC bytes before xmit */
    int            RMB_overflow_en;  /**< return more data than permitted */
    int            verbose;          /**< enable verbose mode for fuzzing */
    int            req_command;      /**< record req command before spoof */
    int            FUZZ_bruteforce;  /**< enable response bruteforcing */
    int            FUZZ_flags_en;    /**< enable flag bruteforcing */
    int            FUZZ_dsfd_en;     /**< enable dsfid bruteforcing */
    int            FUZZ_uid_en;      /**< enable uid bruteforcing */
    int            FUZZ_bfstate;     /**< holds current bruteforcing state */
};

#endif

```

## E mrg\_fuzzing.c

```

/*
* Copyright 2007, Rutger Hofman, VU Amsterdam
*
* This file is part of the RFID Guardian Project Software.

```

```

*
* The RFID Guardian Project Software is free software: you can redistribute
* it and/or modify it under the terms of the GNU General Public License as
* published by the Free Software Foundation, version 3 of the License.
*
* The RFID Guardian Project Software is distributed in the hope that it will
* be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
* You should have received a copy of the GNU General Public License
* along with The RFID Guardian Project Software. If not, see
* <http://www.gnu.org/licenses/>.
*/

#include "mrg.h"

#include "mrg_fuzzing.h"

#include "mrg_fuzzing_private.h"
#include "mrg_fuzzing_filter.h"
#include "mrg_fuzzing_raw.h"

static mrg_fuzzing_t    fuzzing_blank; /* Statically NULled */

/** Create a fuzzing struct */
mrg_fuzzing_t *
mrg_fuzzing_create(mrg_log_t *log, mrg_spoof_t *spooft)
{
    mrg_fuzzing_t *fuzz = mrg_malloc(sizeof *fuzz);

    if (fuzz == NULL) {
        return NULL;
    }

    *fuzz = fuzzing_blank;
    fuzz->log = log;
    fuzz->spooft = spooft;
    fuzz->CMD_change_en = 0; /* toggle command change on/off */
    fuzz->CRC_break_en = 0; /* toggle CRC break on/off */
    fuzz->RMB_overflow_en = 0; /* toggle RMB overflow on/off */

    fuzz->FUZZ_bruteforce = 1; /* toggle bruteforce fuzzing on/off */
    fuzz->FUZZ_bfstate = 0; /* initial bruteforce state, edit for resume */
    fuzz->FUZZ_flags_en = 1; /* enable FLAG field bruteforcing */
    fuzz->FUZZ_dsfid_en = 1; /* enable DSFID field bruteforcing */
    fuzz->FUZZ_uid_en = 0; /* enable UID field bruteforcing */

    fuzz->verbose = 1;
    fuzz->req_command = 0;
    return fuzz;
}

/** Clear a fuzzing struct */
int
mrg_fuzzing_clear(mrg_fuzzing_t *fuzz)
{

```

```
    mrg_free(fuzz);

    return 0;
}

/** Register the fuzzer in the filter chain and the port flush */
int
mrg_fuzzing_start(mrg_port_t *port, mrg_fuzzing_t *fuzz)
{
    if (mrg_fuzzing_filter_start(port, fuzz) == -1) {
        return -1;
    }
    if (mrg_fuzzing_raw_start(port, fuzz) == -1) {
        return -1;
    }

    return 0;
}

/** Init fuzzing module. Invoked from mrg_init(). */
int
mrg_fuzzing_init(int *argc, char *argv[])
{
    if (mrg_fuzzing_filter_init(argc, argv) == -1) {
        return -1;
    }
    if (mrg_fuzzing_raw_init(argc, argv) == -1) {
        return -1;
    }

    return 0;
}

/** End fuzzing module. Invoked from mrg_end(). */
int
mrg_fuzzing_end(void)
{
    if (mrg_fuzzing_filter_end() == -1) {
        return -1;
    }
    if (mrg_fuzzing_raw_end() == -1) {
        return -1;
    }

    return 0;
}
```