

TO: Melanie Rieback, melanie@few.vu.nl
FROM: Andrew Richardson, chardson@umich.edu
SUBJECT: RFID Guardian Fuzzing Project Status Report
DATE: 24 August 2007

FOREWORD

For many industries, radio-frequency identification is a powerful tool to streamline the transfer of goods traveling over complex shipping networks. Combined with intelligent middleware, goods can be routed and located with great precision. As the use of such technology grows more widespread, the risks of an insecure implementation grow more pronounced. In order to aid the industry and ensure security of these networks, the RFID Guardian project has begun automated implementation security testing, or “fuzzing,” and also begun adding similar functionality to the RFID Guardian device, a privacy and access-control device for tags of multiple RFID standards. The purpose of this report is to present both my work and my suggestions on the focus of future efforts.

SUMMARY

My work this summer has focused on laying the groundwork for proper testing of RFID middleware, specifically Oracle’s Sensor Edge Server. I have done this through the use of the automated vulnerability checking application, beSTORM. In order to focus our efforts effectively, our team has split the range of potential attacks into layers, and the main focus was on the highest layer, RFID middleware. I have also spent some time with lower-level attacks, but not sufficient time for conclusive results. This work has met with some impediments, but I have either resolved them or presented suggestions to resolve them in future work.

To break RFID fuzzing into a problem with manageable parts, we have separated the transmission between the transponder and middleware into three layers. This paper attempts to approach our work on each layer in sufficient detail such that further work can resume from nearly the same point. To do so, I discuss the three layers, the work done, challenges overcome, and, finally, I detail a plan for efficiently and holistically approaching further work.

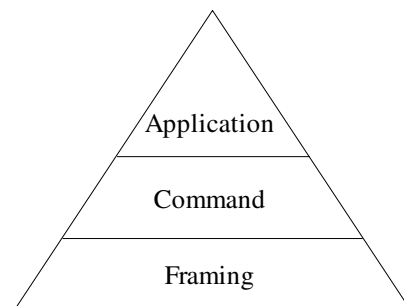


Figure 1: Layers of Interaction

DETAILS

Layers of Interaction

There are three distinct layers of interaction between a RFID transponder and the target middleware. These layers cover transmission, formatting, and content, and we refer to them as the framing layer, the command layer, and the application layer, respectively. Our efforts in fuzzing and testing this summer revolved around these three layers.

The most fundamental layer is the framing layer, as it governs the RFID transmission mechanism, the

transmission of bits and framing delimiters (i.e. a “start of frame” delimiter). While these units are strictly defined in ISO-15693, the popular High Frequency (HF) standard, the strictness of interpretation is hardware-specific, so testing the system boundaries could prove useful. The variable elements in this case are time parameters, the frequencies used, and adherence to Manchester encoding. I imagine that violating the standard in these cases would result in the message being either ignored or read incorrectly, much like a bit change in command layer fuzzing. We have not yet focused on this level.

The next highest layer is the command layer. The command layer is the highest layer defined by the ISO standard. It governs request and response structure in ISO-15693. The data between the frame delimiters, the flags set, and the cyclic redundancy check (CRC) are all variables in this layer. How stringent is the reader software with the structure of the response frame? If the reply to a “read multiple blocks” request is significantly over the 8kB maximum, is the system at risk? These are easily tested with the right hardware.

The application layer, the highest of the three, includes the middleware application. Here, tags have been read and interpreted, and the data transferred is limited to the tag id and data payload. Risks at this level might include code-injection or similar attacks with potential RFID malware and are greatest because of the middleware's complexity. This has been the focus of our work.

APPLICATION LAYER

Progress in Middleware Testing

The current work to create a security test environment has focused in two areas: middleware testing and building Guardian-centered tools. Currently we are testing Oracle’s *Sensor Edge Server* (SES), a server and database suite used to interpret, sort, and record interactions with RFID transponders. There are many different middleware implementations, including Manhattan Associates' or SAP's variants. Middleware’s uses are wide and varied, but could include tag location tracking, delivery time tracking, and tag data storage. The middleware presents many options to shipping professionals, but a weakness in such an infrastructure could prove quite damaging. To test this middleware, we have been using Beyond Security’s automated vulnerability checking application, beSTORM, and Beyond Security graciously provided this tool to us, as well as fantastic support for our project.

beSTORM is an automated vulnerability-checking tool, designed to find exploits and security holes by cycling through many potential attacks and combinations thereof. The user designs the message structure so that beSTORM always creates a well-formed message, else SES will just discard it. The user also specifies known variable parameters in this module where beSTORM should attempt attacks, such as buffer overflows, then allows beSTORM to try all combinations on the target application.

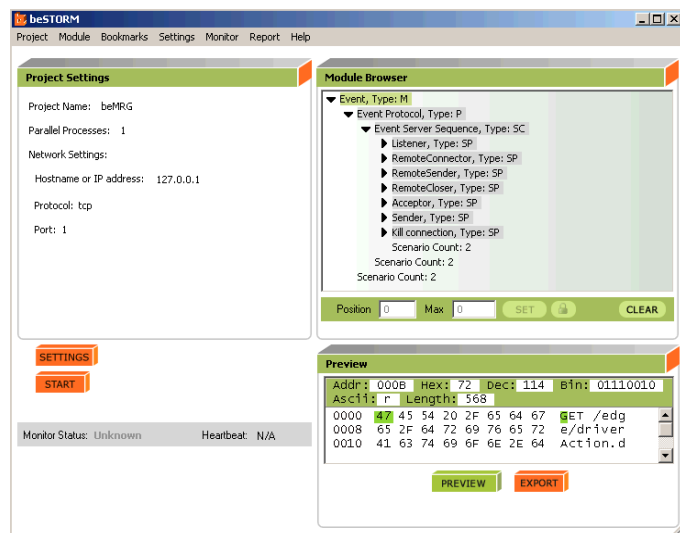


Figure 2 - beSTORM interface

The Sensor Edge Server includes support for approximately eleven UHF RFID readers and two HF RFID readers. The Guardian hardware is currently best developed for HF RFID, such as ISO standard 15693, but, at the time of writing, is not yet ready for UHF (UHF support is on the way). As a result, our options were limited to the few HF readers and the “remote driver.” The remote driver connects over a TCP port connection as client, reportedly for security reasons, and allows users to add support for unsupported readers by writing an application to send a reader’s query results in a specially formatted frame, which I will call an Event, to the Edge Server (see Table 1).

```

<Event>
<Routing>
<siteName>MySite</siteName>
<correlationId></correlationId>
<from>MySourceName</from>
<to>MyDeviceName</to>
</Routing>
<Metadata>
<type>200</type>
<subType>1</subType>
</Metadata>
<Payload>
<createTime>1146959166343</createTime>
<item id="000000000001">Tag Data</item>
<item id="000000000002"></item>
</Payload>
</Event>

```

Table 1 - Example Frame for the Remote Driver

We decided to test for vulnerabilities with the remote driver, thinking that the critical problems would be shared between the reader drivers and the remote driver. This driver was also chosen to allow us to begin testing sooner, in hopes that we might quickly uncover a vulnerability in the server. When fuzzing with an Event frame, the only parameters to actively change are the item id (tag unique identifier, UID) and the tag data field. The rest are assumed to be out of the control of a malicious tag and not pertinent to our investigation. In tests with beSTORM, it may be pertinent to test

these two separately, as the Edge Server attempts to decode tag ID and prints an error when it cannot.

Hurdles and Temporary Setbacks

The process of developing this beSTORM module to interface with the Edge Server was not without some difficulty. Connection settings and the beSTORM connection bug were the most notable issues to date. Extracting useful information from the test results was also of great concern.

The first problem along this path was the server’s remote driver client connection. Both beSTORM and the Edge Server connect as client by default, meaning the two wouldn’t connect directly with the configuration options known at the time. I first wrote a simple routing program to transfer Events from beSTORM's port to the server, pausing and waiting to resume if the connection dropped. This worked reasonably well, but I had doubts that my routing program wouldn’t cause any problems on its own.

Our Beyond Security contacts then helped us rework our module, named “beMRG,” to host connections, but the problem of connection persistence arose. beSTORM supports hosting with some mechanism to reopen this connection after every Event, and our attempts with a persistent connection from beSTORM were not successful. For this reason I forsook the beSTORM host idea and worked on the routing application, again. When I decided on the current method to

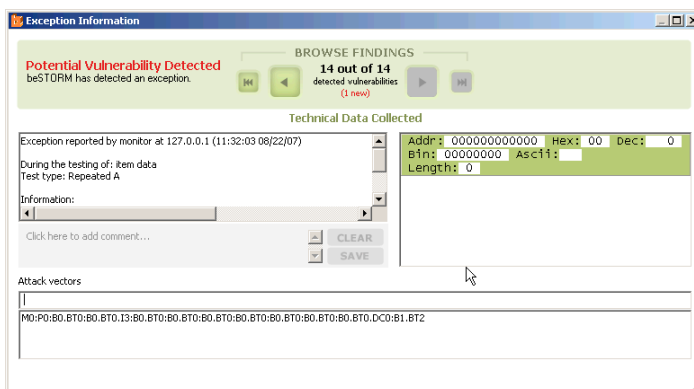


Figure 3 - beSTORM detects an error

reopen the connection by sending the same command over HTTP as a web browser, beSTORM could host the connection. While the HTTP request was not the most elegant option, this functional solution is still the method of choice and has proven sufficiently reliable.

The other main hurdle in this process was a bug in beSTORM with the connection counter that led beSTORM to report an error after every Event iteration. This is no longer an issue in beSTORM v3.01.

Another concern with fuzzing is efficiently and reliably interpreting the results in order to determine which tests are worth further attention. Because the Edge Server has adjustable log verbosity, it seemed that post-processing the log data would prove a good way to gauge the results of the tests. Using Perl, I wrote two scripts to interpret the log data. The first script watches a log for updates (beMRG\watch_edge_log.pl). If an update contains “error” or “exception,” an alert packet is sent to beSTORM, which records the exception and the approximate time for later use (*Note: this script appears to resend old updates and should be fixed before further use*). The second script is used in conjunction with beSTORM’s new grammar element, the Incrementor (unique_id_log_check.pl).

The Incrementor is used to embed a unique id number in each message. Afterwards, the second Perl script searches the logs for gaps in the unique ids to determine which events were completely lost. Initially, I implemented Incrementor functionality with a simple DLL¹, the standard way to add features to beSTORM, but beSTORM 3.0.1 gained the Incrementor element and the previous DLL was rendered unnecessary. In initial testing, a surprisingly high percentage of messages were never recorded in any of the server logs, nearly 50%. The dropped messages we retested did not cause any noticeable server issues, nor did the Edge Server drop them during manual testing. The first twenty messages had valid contents. Their unexplained loss causes concern and skepticism over future test results.

While we do approach middleware testing with the goal of finding software bugs, we must concentrate on bugs that can be recreated by either a tag or a fake transponder, like the Guardian. Because the first twenty lost events were definitely legitimate and not preceded by any malformed events, it seems a transmission issue and not a useful vulnerability. Steps to find a workaround is proposed in the “Further Work” section.

COMMAND LAYER

Guardian Fuzzing Interface

Because the reader hardware may be responsible for validating tag response frames, we much approach command layer fuzzing with a device capable of emulating real tags over the RF contactless interface. The device must also be capable of manipulating the responses and still reply within the timing requirements of the standard. The RFID Guardian is one such device. Because the RFID Guardian already has tag-emulating functionality, adding fuzzing functionality is relatively straightforward. With the addition of a few filters and a few callbacks to catch the requests and replies at various stages of transfer, we were able to run a few very basic test cases in little time.

In the Guardian, request frames from the reader are passed through software filters responsible for generating the response frames and maintaining the Guardian’s other functions, such as tag jamming for access control. The “spoof filter” is responsible for interpreting the reader’s request and generating

¹ See the dllsources.zip archive for dll sources

replies for emulated tags. To easily test the reaction of the reader to a response generated by a different command, we used a custom filter before the existing spoof filter to modify the request frame (see Appendix D, `mrg_fuzzing_filter.c`). The generation and handling of the response is then left for the spoof filter and other, subsequent filters.

We also added a callback before the reply is sent (see Appendix D, `mrg_fuzzing_raw.c`). After the response is fully generated by the spoof filter, the “raw” callback has the ability to change the data in the raw buffer, including framing delimiters and anything between. Because the fuzzing logic has complete control of the response here, this is a better location for elaborate command-layer fuzzing. The operator may use this filter to change the CRC bytes before sending the reply, in order to verify the reader's CRC check. Another test is to add a random byte after the CRC check, to see if the reader relies on the CRC being located in the last two bytes, the expected indices, or both.

We used a demo application with the Philips Pegoda reader to run tests at the command layer. In our few trials with modified requests, such as modified CRC bytes, the Pegoda software reported, “Tag in wrong state or no tag...” It should not be difficult to build a sample bank with one request frame for each request type and then iterate through each to test the reader’s reaction.

FRAMING LAYER

Guardian tag-write-only

This is a small application that sends malformed messages and pays no attention to the reader for timing (~`mrg/test/tag-write-only/main.c`). A buffer size is chosen, as well as a repeat delay, and the buffer is filled with a certain kind of bit or frame delimiter. These include:

- Repeated 1's
- Repeated 0's
- Random, valid bits
- Random half-bits² (frequently violates Manchester encoding)
- Start of Frame delimiters
- End of Frame delimiters

In our testing with random half-bits, the Pegoda reader application printed “CRC Framing error...” for each frame. When we ran the Pegoda demo application in inventory mode for twelve hours with the Guardian constantly sending random half-bits, we noticed a memory leak in the Pegoda demo software, as the real memory allocation increased from 6MB initially to 100MB in that period. It may be possible to increase the rate at which memory is consumed by tailoring the response. Also, it may be useful to verify that the reader DLL is not the source of the leak, as the DLL is still used when the demo application is not.

FURTHER WORK

To continue the work on RFID fuzzing with the current tools (Pegoda reader/demo software, Oracle's Sensor Edge Server with the remote driver), further work should cover all three layers of transmission.

² Manchester encoding requires transitions from one half-bit to the other, such as a modulated signal switching to no modulation. So, randomly chosen half bits can cause no transition over a few bit periods.

Framing Layer

Framing layer tests verify the transmission characteristics of the reader hardware. Some time should be devoted to determining these limits and if bending the rules of the system has any appreciable affect on the system's operation. The transmission characteristics in question are carefully defined in ISO-15693, part 2, and the main test targets are timing, transmission frequency, and adherence to Manchester encoding.

Command Layer

My suggestion for fuzzing at the command layer is to start with test-case trials to build an understanding of the role of the fuzzer and familiarity with the relevant variables.

Requests and replies are made from frame delimiters and various bytes. The general request format begins with a start of frame, followed by request flags, a command code, parameters, a data field, a cyclic redundancy check, and an end of frame delimiter. The request flags specify the transmission characteristics, such as data rate, whether it is an inventory request, the presence of a few optional fields, and a few other items. The command codes distinguish a request between multiple different types, including "inventory," "get system information," and "read multiple blocks," to name a few. Lastly, the CRC is used to verify that the transmission was successful.

Response frames are not much different. Start of frame, flags, parameters, data, CRC, and an end of frame make up the general format. The main flag field only contains one used option, the error flag, used to signify that there was an error and that the response is not the requested type, but an error frame.

When running either a complete fuzzing attack, or simply a sample test, the tester may consider:

- Adding arbitrary bytes to the frame, before and after both a valid and an invalid CRC field.
- Changing the request type, though it may not matter if the response is a valid response to a different request – responses contain no command code and are probably indistinguishable from random bytes.
- Whether a "read multiple blocks" request limits the size of the response frame, and what affect returning more than the maximum allowed memory size, 8KB, has on the system.

The Appendix at the end of this report includes some information on the Guardian fuzzing implementation and ISO-15693 part 3 best describes the request/response structure that makes up the command layer.

Application Layer

Unfortunately, most work with the application layer was spent preparing for detailed fuzzing tests, but no exhaustive tests have yet been run. The first step before running proper, exhaustive tests is to sort out the 50% transmission problem – a problem because all of the first twenty Events seem completely legitimate, but the majority of them were not properly recorded. Because the beSTORM test continues to run, despite the dropped messages, we do know that the Edge Server is connecting after every request (beSTORM makes one request before waiting for the connection -- no timeout is currently set).

Transmission over the socket was checked with beSTORM's included network sniffer, but future debugging efforts might utilize a different network sniffer to be certain that the Edge Server is getting every Event in full. Beyond that, consider a test where the module reports an exception to beSTORM in

every round, like the watch_edge_log.pl Perl script. This would slow Event transmission to the tester's click rate and help clarify whether the solution could be time-based.

In an effort to improve the module functionality, consider using beSTORM's binary elements, as defined by the grammar in the User Guide, and embedding some combination of them in both the ID field and the Tag Data field when using the remote driver. Another way to improve the tests is to discuss Java vulnerabilities and how to best structure the modules toward the Java-based Sensor Edge Server with the Beyond Security contacts.

Hardware and Application-Layer Integration

A physical RFID fuzzer that tests multiple middleware applications without custom drivers could be used more often and with less setup time than a software-only implementation, like the remote driver. To allow for this, application layer fuzzing must be integrated with the hardware device and potential exploits sent over the contactless interface. The framing and command layers depend on the contactless data transmission and integrate with hardware from the start.

Because the only variable at the application layer are the UID and data fields of the RFID transponder, beSTORM could integrate with the Guardian hardware by continually refilling a buffer which contains the simulated transponder's UID and payload data via a connection to the Guardian hardware. beSTORM could potentially update this buffer at some regular rate to vary the attack on the middleware.

Another option is to integrate various layer tests and look for any appreciable effects. This is best saved for after each layer has been well tested on an individual basis, and should especially be considered if vulnerabilities were found in any layer. Assuming sensible initial design, layered tests are individual enough that they should combine without much additional effort.

Recommended Order of Approach

Because there are many variable options and levels on which to base an attack, it is easy to get caught up in trying all of them while not concentrating enough on any one. For this reason, I have reflected upon my efforts and present this recommended order of operations. I also suggest taking care to not switch between tasks or layers too frequently, as such behavior could be a unnecessary distraction.

- Basic hardware and protocol familiarity
 - Learn ISO specifications and RFID Guardian abilities, such as Guardian Protocol
 - Use the existing fuzzing filters with verbosity and a simulator build of the guardian and reader (see Appendix)
- Framing layer test cases
 - Send long/short bits and frame delimiter timings
 - Try frames at incorrect frequencies
- Command layer test cases
 - Try each wrong response type (and random data) for each request type
 - Set the error-flag with the otherwise original response frame
 - Send a "read multiple blocks" reply with more/less data than requested, and with more than allowed by the standard.
 - Try raw response modifications, such as an added byte before and after the CRC or other

malformed responses, as the reply buffer is completely modifiable in the relevant callback.

- Application layer fuzzing
 - Read the document files in the beMRG module folder
 - Learn Event structure (table 1) and test connections with Hyperterminal or equivalent
 - Run a sample test using beMRG
 - Fix 50% loss issue, if not based on malicious message contents
 - Test tag id and data separately.
 - Quickly add exception/error logging to the post-processing (second) Perl script.
 - Be careful to not spend too much time on log processing Perl scripts. It's probably more useful to break the server and have to figure out what worked than to have less time for testing and still risk being unsure of the cause.
 - Tailor model to Java vulnerabilities.
 - Hardware integration of application layer fuzzing, if the remote driver tests show vulnerabilities.

CONCLUSION

An RFID fuzzing device has potential to help verify the security of middleware implementations. There are many important aspects of the protocol that should be checked to ensure that no vulnerabilities exist at a fundamental level. RFID middleware deserves sincere testing to ensure security of the databases that support the entire implementation and that the malware risks are relatively low.

We have begun adding functionality to the RFID Guardian device for testing the framing and command layer, and we have worked with Beyond Security and their fuzzing product, beSTORM, to develop a fuzzing module with which to test RFID middleware, such as Oracle's Sensor Edge Server. This work is not yet completed, and exhaustive testing is still needed before drawing conclusions on the security of these systems. Current project status and further-work suggestions should provide a solid groundwork on which to base these efforts.

APPENDIX A – Getting Started

Oracle Sensor Edge Server Resources

Sensor Edge Server tutorials:

http://www.oracle.com/technology/products/sensor_edge_server/tutorials.html

Sensor Edge Server forum:

<http://forums.oracle.com/forums/forum.jspa?forumID=211>

Oracle Database and Application Server documentation:

<http://www.oracle.com/technology/documentation/index.html>

Some information regarding the remote driver:

<http://forums.oracle.com/forums/thread.jspa?messageID=1295672�>

An Edge Server blog with configuration information:

<http://edgeserver.de/>

René Nyffenegger provides useful information needed for Edge Server setup with database support:

http://www.adp-gmbh.ch/ora/admin/backup_recovery/archive_vs_noarchive_log.html

Also, a developer guide for the Sensor Edge Server was unofficially released (via email), and for lack of a better location, is stored in the beMRG directory.

beSTORM Module – beMRG

The module was last located in D:\upload\beMRG of my machine, including small documents on specific aspects. The latest beSTORM installers (v 3.0.1) are stored in D:\upload\installers\

beSTORM's User Guide is an invaluable resource, but there are unmentioned resources in the beSTORM root directory, such as beSTORM_advanced_features.txt document describing dll implementation and tailing logs with Perl scripts.

Guardian Fuzzing Implementation

The fuzzing structure declared in `~mrg/src/fuzzing/mrg_fuzzing_private.h` currently includes flags to enable and disable a few existing modifications (command change, crc change, verbosity). These flags are set in the function `"mrg_fuzzing_create(...)"` in:

`~mrg/src/fuzzing/mrg_fuzzing.c`

where `~mrg` is the topmost directory of the rfid-guardian checkout.

The filter to modify requests before interpretation is located in the function `"mrg_fuzzing_filter_before(...)"` of:

`~mrg/src/fuzzing/mrg_fuzzing_filter.c`

The callback to modify responses after calculating the 16-bit CRC is the function `"fuzz_raw_callback_post(...)"` of:

`~mrg/src/fuzzing/mrg_fuzzing_raw.c`

The fuzzing struct is named `"fuzz,"` and the fuzzing, request, and response structures are in use in both functions, which should be enough to clarify proper usage.

DLL Sources

Available in the beSTORM module directory as `~beMRG/dllsources.zip`

APPENDIX B – beMRG Readme

=====
beMRG
=====

This is the readme for beMRG. It's more of a skeleton or tree to point to other docs on more specific topics. To be useful, you should have already installed Oracle's Sensor Edge Server, successfully. At time of writing, there are three drive backups of C:\ in D:\backup on the machine "vague", in case "vague" is still the machine of choice and something happened to the current install, you can restore using:

Start -> Accessories -> System Tools -> Backup

Read the docs in this order:

~beMRG\README FIRST.txt

This file, a general readme for beMRG, a beSTORM module for use in fuzzing Oracle's Sensor Edge Server.

~beMRG\FILE LISTING.txt

This doc lists different files in ~beMRG and their purposes. It should include both subfolders and their contents.

~beMRG\File-Tail for perl\FILE-TAIL AND PERL NOTE.txt

A brief note on perl setup and FILE::TAIL setup, for tailing logs or other files.

~beMRG\SES INTEGRATION.txt

A note about integrating with SES* and setting up the remote driver.

~beMRG\HOWTO.txt

This doc covers aspects of running the beSTORM module and the various perl scripts.

~beMRG\barebones image guide\index.html

This image & caption guide exists to aid the process of getting used to server, database, beSTORM module, and Perl script usage. It also contains necessary machine & user passwords.

~beMRG\MODULE EDITING.txt

This doc is intended as a supplement to the beSTORM user guide's section on modules and beSTORM grammar.

~beMRG\beSTORM - User Guide.pdf

Now that you have some idea what's going on, you should read through the User Guide to get a better handle on beSTORM

*(SES = Oracle Sensor Edge Server)

APPENDIX C – beMRG File Listing

=====
PURPOSE

=====
This is a file listing of the various files/folders, and their function in the beMRG beSTORM fuzzing module.

Groups: I. ~beMRG (perhaps D:\upload\beMRG)
II. ~beMRG\taillogs
III. ~beMRG\File-Tail for perl
IV. ~beMRG\Logs
V. ~beMRG\barebones image guide

=====
I. ~beMRG

I.1 - settings.bsp

A beSTORM file, modify to edit the beMRG module. See HOWTO.txt for information on using this module. Also, beSTORM modules such as this one are described using the beSTORM grammar, which is detailed in the beSTORM user guide.

I.2 - other beSTORM files

bookmarks.bsp - beSTORM bookmarks listing file.
bookmarks.bsp.1 - beSTORM backup file. Ignore.

I.3 - watch_edge_log.pl

IMPORTANT NOTE

This Perl script appears to be printing old events, due to the Tail use of FILE::TAIL. If you run it with beSTORM, wait for a while, close the script and restart both a while later. You should soon see timestamps printing out from the last run. For example, I ran both, then did another test in a half hour. When I did, the script started printing out events timestamped from thirty minutes before. This means that the beSTORM report is probably getting false errors. Do not trust the script before fixing this problem.

END NOTE

Perl script to do many tasks related to watching the most recently modified sensor edge server log, See the ~beMRG\taillogs section below to see what this script outputs. It also prints some information to the terminal and sends errors to beSTORM via UDP packets.

I.4 - unique_id_log_check.pl

Perl script to search every log in the Edge Server's log directory and check for missing unique_ids. Prints out a uniqueID_postcheck#.log log of files tested and missing ids. Assumes that the unique_id sent by beSTORM should never decrease and so uses the highest found id (per file) when checking. Outputs the missing ids in an inclusive format, i.e. in this case, 54 and 71 to 76 are all missing:

M: SESnameid50: SESuniqueid[54]

M: SESnameid50: SESuniqueid[71..76]

The script could use a bit of polishing (some unused variables, I'm sure). Log directory to check:

C:\edge1\product\10.1.3.1\OracleAS_1j2ee\home\applications\edge\edge\log

I.5 - other .txt files

See README FIRST.txt for information on the various readmes and notes around ~beMRG

=====
II. ~beMRG\taillogs
=====

Logs created by the perl scripts. A new set of II.1 and II.2 is made at every launch of watch_edge_log.pl. II.3 is made by unique_id_log_check.pl.

II.1 - cleanedgelog.#.log

Any new line from the log being tailed by the perl script, except for lines containing a certain expression as defined in the script. Any call to \edge by the web browser prints out a debug line that is not useful to us (but we probably want all other debug info), so we copy over every line not containing such an expression. A little easier to look through.

II.2 - sortedlog.#.xml

This is a very simple XML file, which is a little cleaner way to display some of the errors and exceptions caught. The perl script that generates this log may need more work to make this file truly useful.

II.3 - uniqueID_postcheck.#.log

This log is output from the unique_id_log_check.pl perl script. It includes a listing for each file checked, the SESnameid in use for filtering events, and the date the file was last modified. The files are processed oldest-first, and they print out in the same way. At the end is a quick Percentage-missing count. It's only intended for a rough idea about how well the session was received.

=====
III. ~beMRG\File-Tail for perl
=====

Some files in case you do not have FILE::TAIL for perl. Read the FILE-TAIL AND PERL NOTE.txt included there for more information.

=====
IV. ~beMRG\Log
=====

This is beSTORM's directory for run-time logs from the beMRG module (settings.bsp)

=====
V. ~beMRG\barebones image guide
=====

This image & caption guide exists to aid the process of getting used to server, database, beSTORM module, and Perl script usage. It also contains necessary machine & user passwords.

APPENDIX D – RFID Guardian Fuzzing Code

FILE: ~mrg/src/fuzzing/mrg_fuzzing_filter.c

```
static mrg_rfid_filter_verdict_t
mrg_fuzzing_filter_before(const mrg_frame_request_t *g_req,
                          mrg_frame_response_t *g_resp,
                          mrg_rfid_filter_verdict_t verdict,
                          void *v_descr)
{
    mrg_fuzzing_t *fuzz = v_descr;
    mrg_tag_t *tag;
    mrg_frame_15693_request_t *Req = (mrg_frame_15693_request_t *) &g_req->frame_15693;
    uint8_t newcommand = 0;

    mrg_lock(fuzz->lock);

    if (! fuzz->enabled) {
        goto exit;
    }
    /* record command type in fuzzing struct for later use */
    fuzz->req_command = Req->command;

    tag = mrg_port_get_specimen(fuzz->filter_port);
    /* If fuzzing verbosity is enabled, print request frame */
    if (fuzz->verbose) {
        mrg_diag_printf("In the fuzzing filter.\n");
        mrg_diag_printf("  Request[1]:  %s\n", mrg_frame_req_string(tag, g_req));
        if (fuzz->CMD_change_en) {
            mrg_diag_printf("Req.flags before: 0x%02X\n", Req->flags);
        }
    }
    /* If changing the command type of the request is enabled, do */
    if (fuzz->CMD_change_en && Req->command == MRG_FRAME_15693_GET_SYSTEM_INFORMATION) {
        /* change command in this line */
        newcommand = Req->command = MRG_FRAME_15693_INVENTORY;
        if (newcommand == MRG_FRAME_15693_INVENTORY) {
            Req->req.inventory.afi = 0;
            Req->req.inventory.mask_length = 0;
            Req->req.inventory.mask = 0;
            Req->flags |= MRG_FRAME_15693_INVENTORY_FLAG |
                MRG_FRAME_15693_NB_SLOTS_FLAG;
            Req->flags &= ~(MRG_FRAME_15693_AFI_FLAG |
                MRG_FRAME_15693_INV_OPTION_FLAG);
        }
    }
    /* If fuzzing verbosity is enabled, maybe print some messages */
    if (fuzz->verbose) {
        if (fuzz->CMD_change_en) {
            mrg_diag_printf("Req.flags after: 0x%02X\n", Req->flags);
        }
        if (fuzz->verbose) {
            mrg_diag_printf("In the fuzzing filter.\n");
            mrg_diag_printf("  Request[2]:  %s\n", mrg_frame_req_string(tag, g_req));
        }
    }
exit:
    mrg_unlock(fuzz->lock);

    return MRG_RFID_FILTER_CONTINUE;
}
    if (fuzz->CMD_change_en) {
        mrg_diag_printf("Req.flags before: 0x%02X\n", Req->flags);
    }
    }
    if (verdict == MRG_RFID_FILTER_RESPOND) {
        mrg_diag_printf("  Response: %s\n", mrg_frame_resp_string(tag, g_req, g_resp));
    }
}

exit:
    mrg_unlock(fuzz->lock);

    return MRG_RFID_FILTER_CONTINUE;
}
```

```
-----  
FILE: ~mrg/src/fuzzing/mrg_fuzzing_raw.c  
-----
```

```
static int  
fuzz_raw_callback_pre(mrg_port_t *port,  
                      const mrg_data_stream_t *c_stream,  
                      void *arg)  
{  
    mrg_data_stream_t *stream = (mrg_data_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");  
    }  
  
    /* 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 0;  
}
```