Security Testing OpenGGSN: 
a Case Study

Esa Tikkala, Mikko Rapeli, Kaarina Karppinen, Hannu Honkanen

VTT Technical Research Centre of Finland, Oulu, Finland

Abstract

As systems today are more and more reliant on the communication between different systems and their co-operation, network security is crucial for the overall security of the whole system. Although protocols themselves can be theoretically proven secure, it is the implementations and complete systems built upon them where the most security bugs reside. Traditional testing is not so effective in finding the security problems and checking the implementations for security problems requires a lot of effort and expertise. Security bugs are often found by doing unexpected things and looking for changes in system and environment behavior and additional side effects. These negative side effects are the main cause that prevents companies from running these tests in their real networks – requiring massive investments in parallel test networks.

As security can never be absolute – not even with vigorous testing - additional precautions are advisable. A set of different testing tools will provide a better coverage of the tested system. System should be made more rugged and more difficult for the crackers and malicious programs to take advantage of. Adopting some metrics also helps in evaluation the security of the system and the testing coverage achieved.

In this paper we illustrate the experiences of security testing OpenGGSN, an open source implementation of a Gateway GPRS Support Node (GGSN) that uses the GPRS tunneling protocol (GTP).

Keywords: Information security, security testing, OpenGGSN, GTP

I Introduction

In our Security Testing and Monitoring project we aimed at building a uniform security testing and monitoring framework by applying testing and monitoring products, which are provided by Finnish manufacturers. Development work in the project has been divided into phases, which each apply testing and monitoring methods in specific target environments. In this paper we concentrate entirely on one of the project’s test targets - OpenGGSN, which is an open source implementation of a Gateway GPRS Support Node (GGSN) [1], licensed under the GNU General Public License (GPL).

The purpose of testing the OpenGGSN was to study different approaches for improving the security attributes of a protocol implementation. OpenGGSN provides a freely available implementation of GTP protocol, with source codes, that could be tested with Codenomicon GTP test tool. In addition to using the commercial tools by Codenomicon, the availability of the sources and compilation environment allowed us to freely experiment with various source code analyzers, scanners and compilation options. These
trials enabled us to gain experience of various testing tools and their proper use. As a side product we found some security problems for which we developed fixes. Some good security and coding practices for the implementation and compilation environment are also discussed.

The rest of the paper is organized in the following way. Section II discusses the details of OpenGGSN and Section III presents the test environment and methods used. Section IV analyses the test results. Finally, Section V discusses future work and Section VI gives conclusions.

II OpenGGSN

OpenGGSN is an open source implementation of a Gateway GPRS Support Node (GGSN), licensed for free under the GNU General Public License (GPL). It is used by mobile operators as the interface between the Internet and the rest of the mobile network infrastructure as shown in Figure 1.

![Figure 1 UMTS/3G network overview.](image)

OpenGGSN implements the GPRS tunneling protocol (GTP) version 0 and version 1, and provides an SGSN emulator suitable for GPRS core network testing [2].

The GGSN is the anchor point that enables the mobility of the user terminal in the GPRS/UMTS networks. In essence, it carries out the role in GPRS equivalent to the Home Agent in Mobile IP. It maintains routing necessary to tunnel the Protocol Data Units (PDUs) to the SGSN that service a particular MS (Mobile Subscriber). Other functions include subscriber screening, IP Pool management and address mapping, QoS and PDP context enforcement. From the external IP network’s point of view the GGSN is seen as a normal IP router [3].

According to OpenGGSN project web page [2] the main design goals of OpenGGSN were stability, portability and scalability. The following design choices were made:

- Programmed in (ANSI) C in order to improve portability to other platforms.
Concurrency is implemented using a single select() loop in order to improve portability and at the same time achieve high throughput.

Application was developed in userspace only. Provides good portability at the cost of performance. Performance can be increased by implementing user plane handling in kernel space.

Conservative handling of memory allocation and error checking. Helps improve stability, but should be optimized for performance at a later stage.

A typical architecture for installation of OpenGGSN is given below:

![Figure 2 A typical installation of OpenGGSN [2]](image)

Two different networks are involved in the architecture:

- Gi. The Gi network is the interface between the GGSN and an external data network. Access to the Gi network is controlled by the GGSN. Gi is typically the Internet or a corporate intranet.

- Gn. The Gn network is connecting the SGSN with the GGSN. It is used for tunneling IP packets between the SGSN and the GGSN as well as control plane traffic and management traffic (NMS). In the example above the Gn network is allocated the address range 172.16.0.0/24. The DNS servers on the Gn network are used by the SGSN for determining the IP address of the GGSN.

OpenGGSN provides 3 components:

- gptplib; a library, which contains all functionality relating to the GTP protocol.

- ggsn, which implements a Gateway GPRS Support Node.

- sgsnemu; an application, which emulates a Serving GPRS Support Node (SGSN).

According to OpenGGSN developers OpenGGSN can be used for testing and developing new systems and equipment by mobile operators, infrastructure vendors, test equipment manufacturers, consultants, systems integrators and universities [2].
III Conducting the testing

Testing was divided in several use cases where different methods and tools were used. Methods included robustness, black box and white box testing along with source code and execution analysis.

Codenomicon GTP Test Tools

Figure 3 illustrates the Linux based testing environment consisting of the OpenGGSN version 0.84-1 in a chroot environment to keep it in a separate "sandbox". Once the GGSN server was started it provided the implementation necessary for the testing. With the Codenomicon GTP Test Tools the test tool was run on a separate workstation connected with a cross over RJ-45 LAN cable directly to the workstation running the GGSN.

![GGSN test setup](image)

In order for the systems to communicate properly the testing host IP address was set to be in the same subnet as the GGSN and any firewalls on the machines were disabled. A direct connection was used so the robustness tests do not accidentally affect other devices like routers.

The GTP test tools, containing more than 100,000 test cases, can be used to test the protocol implementation for development flaws using a unique fault injection technology. The test tool uses a black box approach for the security and robustness testing of products. [4]

Flawfinder

Flawfinder is a program that examines source code and reports possible security weaknesses "flaws" sorted by risk level. It can be used to find and remove some typical potential security problems.[5] Flawfinder was used to inspect the OpenGGSN sources for possible programming errors. The reported errors were then correlated with the results of the Codenomicon GTP test tool.

RATS
RATS - Rough Auditing Tool for Security - is an open source tool for scanning C, C++, Perl, PHP and Python source code and flagging common security related programming errors such as buffer overflows and TOCTOU (Time Of Check, Time Of Use) race conditions. RATS can identify potential trouble spots on which to focus, along with describing the problem, and potentially suggest remedies. It also provides a relative assessment of the potential severity of each problem, to better help an auditor prioritize. [6]

RATS was used to analyze the OpenGGSN sources for potential security problems. The reported problems were then correlated with the results of the Codenomicon GTP test tool.

GCC Warnings

Compilers can detect bad programming habits and warn about them. Compilers like GNU Compiler Collection (gcc) also support additional security related checks such as “-Wformat-security” and even fixes like “-funsigned-char”. [7]

OpenGGSN sources were analyzed with gcc “-Wall –g” compiler flags.

Valgrind

Valgrind is a tool for memory debugging, memory leak detection, and profiling. There are multiple tools included with Valgrind. Most known is the Memcheck which keeps track of the validity and addressability of memory. Memcheck can also detect off-by-one bugs where a program reads or writes outside an allocated block by a small amount. [8]

OpenGGSN was run with Valgrind during a subset of Codenomicon GTP test.

Gprof

GNU profiler (gprof) is a performance analysis that investigates the behavior of a program using information gathered as the program runs, as opposed to static code analysis. The usual goal of performance analysis is to determine which parts of a program to optimize for speed or memory usage. Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, operating system hooks, and performance counters. [9]

We ran into a problem with this tool as it requires the test target to call exit or return from the main function correctly to log the results. The default OpenGGSN implementation had only a crude timer based exit and no proper handling for signals generated for example from control-c key press (SIGINT) that was required to function correctly with the gprof. A proper signal handler had to be made to get the logging function correctly. Another unfortunate side effect was that the gprof tools internal timer signal interrupted a select loop and caused the OpenGGSN server to jam while reading data from an empty socket. This was due to improper signal handling in the GGSN implementation, which incorrectly trusted a file descriptor set (FDS) after select system call was interrupted by a signal and returned with an error value. After some heavy manual reading and debugging the error was found and fixed by clearing FDS if select returned an error.
GNU coverage tool (gcov) is a test coverage program that can be used to analyze programs efficiency and to discover untested parts of the program. Gcov can be used along with the other profiling tool, gprof, to assess which parts of the code use the greatest amount of computing time. Coverage tools can be used together with testsuites, to see how much of the program is exercised by the testsuite and what kinds of test cases need to be added to make sure software is tested thoroughly. [10] OpenGGSN was compiled with execution coverage logging “-fprofile-arcs” and “-ftest-coverage” from the GCC suite and tested against the Codenomicon GTP test suite. The build process was modified to use these variables.

IV Test Results

Test results were analyzed from the perspective of system developers with both software and hardware knowledge. Used methods include stress and robustness testing, white and black box testing combined with source code and execution analysis.

Codenomicon GTP Test Tools

During first tests, the Codenomicon GTP test tools reported problems in OpenGGSN. The tester and the test target were jammed to the last test case indicated by "possible denial of service" note. Testing and debugging revealed that this was caused by a bug in the GTPIE parsing routine. It was found that running any test case containing a message with more than 256 information elements in the payload caused the OpenGGSN to go into an infinite loop and thus causing an Denial of Service condition and unnecessary consumption of CPU resources. This bug required a complete restart of both the GTP test tool and the GGSN implementation. This made testing quite labor intensive as the testsuite contained thousands of test cases of which several consecutive tests failed. This triggered the need to fix the problem as illustrated in Figure 4.
The problem lies in the parsing of information elements in GTP messages, which is implemented in the gtpie_decaps function of gtp/gtpie.c file.

The implementation has a bug that does not check if there are too many information elements in the message thus causing the software to loop infinitely in the while-loop.

In addition, handling routine for the error situation had to be implemented outside the while-loop.

--- openggsn-0.84.orig/gtp/gtpie.c
+++ openggsn-0.84/gtp/gtpie.c
@@ -188,7 +188,7 @@
     memset(ie, 0, 4 * GTPIE_SIZE);
     while (p<end) {
       while ((p<end) && (j<GTPIE_SIZE)) {
         if (GTPIE_DEBUG) {
           printf("The packet looks like this:\n");
           for( i=0; i<(end-p); i++) {
             (unsigned long) p, (unsigned long)
           end);
         return 0; /* We landed at the end of the packet: OK */
       }
     else if (!(j<GTPIE_SIZE)) {
       if (GTPIE_DEBUG) printf("GTPIE too many elements.\n");
         return EOF; /* We received too many information elements */
       else {
         if (GTPIE_DEBUG) printf("GTPIE exceeded end of packet. %lx
%lx\n",
           (unsigned long) p, (unsigned long)
         end);
---

Figure 4 Fix for the OpenGGSN DoS

Once the GGSN implementation was fixed and we were able to execute the complete Codenomicon GTP testsuite we took the other tools into use.

Flawfinder

Flawfinder found several issues in the source code and these findings were manually checked for validity. Our analysis did not reveal any security problems and most of the flawfinder findings came from potentially bad C programming habits.
RATS

RATS provided several warnings and suggestions for better coding practices but our analysis did not find any clear security problems.

GCC Warnings

None of the reported warnings seemed to be exploitable security related bugs and they were also very easily fixed.
OpenGGSN could have some more protection against possible stack based buffer overflows by using a new GCC feature: “-fstack-protector” that checks and guards for buffer overflows, such as stack smashing attacks.

Valgrind

The only error found was a false positive since Valgrind did not understand a Linux system call correctly and falsely interpreted it as an error. Also, all of the 14 warnings were not bugs but bad programming habits and could be fixed by freeing memory before exit.

Gprof

Some improvement and optimization targets were identified but no action was taken as they were not security errors. The fixes done to get the tool running properly in the first place could have increased the overall quality of the GGSN but did very little in raising the security level. Only if the malicious person or program was already running with administrative lever rights they could have been utilized.

Gcov

Running gcov against the Codenomicon GTP test suite revealed that we did not get a full 100% coverage with testing although all the test cases in the GTP test suite passed.

Future work

Unfortunately the OpenGGSN website has been abandoned and it is no longer available. The implementation is still available from sourceforge archives [11] and it provides a good test bed for various testing tools and methods.
Some new tools such as Fuzzers could be tested with the now fixed implementation to see if new problems are found. Integrating the GGSN implementation in a real network with all the other relevant entities could also be an interesting experiment. It would also enable testing the system with real user load.
VI Conclusion

A common problem with the source code scanners and analyzers is that they do not find all the errors and that they produce a vast amount of false positives. These tools can aid the developers but they do not replace manual code inspection. For example Flawfinder seems to be good for checking the source code tree for potentially bad habits, but that is all. The tool logs must be carefully analyzed since most of the warnings are not actual bugs. Also, Flawfinder does not like highly portable code, which relies on the most basic functions and system calls. Compilers and their inbuilt features are often forgotten. Authors should pay more attention to compiler warnings and fix the potential problems early on. Compilers also facilitate various security related features that can enhance the security of the programs simply by enabling the right compiler flags.

The project group developed patches for some of the problems. These patches provided a fix to a remote Denial-of-Service condition in GTPIE parsing and signal- and select error path handling in ggsn.c, added stack protection, fixed signedness issues, freeing used memory at exit and updating the “gengetopt” command line option parser. A signal handler for the SIGINT signal (ctrl + c) was also added for successful usage of the GNU profiler tool.

In the end the patched version of OpenGGSN:
- passed all the test cases with the Codenomicon GTP test suite [4],
- had no security problems to be found by Flawfinder [5], and
- did not leak any memory during robustness tests with Valgrind [8].

Despite all the testing we did not achieve a full testing coverage which is a clear sign that bugs might remain.

We also learned that even though there are tools that are easy to use and produce lots of helpful data it only helps you if you know your how to utilize this information. Especially finding the root causes of the bugs and fixing them properly requires solid expertise.

Testing and analyzing large open source implementations not only provide realistic test benches but also give something back to the open source community - if the testing and development results are shared. Too often the results are dismissed and forgotten due to the effort required to make it public or simply because fixing the target systems is not part of ones job or allowed by the companies. Sometimes the biggest problem is finding the right party to whom one can tell about the security issues. All these issues are manageable and the open source community is setting an example for the whole software industry how to do it.

References


