ON VULNERABILITY TESTING OF VOIP SOFTWARE
The Megaco/H.248 System as an Example

Son Vuong, Xiaojuan Cai, Ling Yun, Wing Keong Woo
Department of Computer Science, University of British Columbia
2366 main mall, Vancouver Canada

Keywords: VoIP, Megaco, Vulnerability, Mutation, Syntax Testing, Vulnerability testing

Abstract: The ever increasing quantity of newly discovered computer security holes makes many network-based service including especially Voice over IP (VoIP) system vulnerable, hence impose a heavy impact on business development. Megaco or H.248 is a recent emerging VoIP protocol which will promote carriers to move into VoIP applications. In this paper, we present the vulnerability testing of Megaco protocol, with a focus on the mutation-based syntax testing approach. We discuss the process of vulnerability test suite generation for Megaco, which is based on parameter variation and a TTCN-3 based framework. The result of a demonstrated testing of a commercial Megaco product is also presented

1 INTRODUCTION

Computer or software vulnerability can be defined as a weakness or flaw in a system that can be exploited to violate its intended behavior. With the fast developing of the internet, the number of newly discovered vulnerabilities reported to CERT continues to more than double each year, and the existence of vulnerabilities is quickly becoming a fact of life for many network-based services, among which Voice over IP (VoIP) system is a typical example.

VoIP systems – as they operate on top of standard IP technology – are susceptible to the same problems as all other IP-based services. These problems include the unverifiability of the origin of data packets, the ability for third parties to capture and modify data packets by intercepting them en route, and the general inability to guarantee timely delivery of data.

The trend of ever increasing quantity of newly discovered vulnerabilities, the rise in frequency of major Internet-based attacks, and VoIP's fundamental exposure to these attacks, are causes for concern as VoIP technology begins to replace traditional phone systems while attempting to provide the same privacy, performance, and reliability characteristics. Hence the vulnerability testing of VoIP system is a very important issue in the way to promote the VoIP business.

Vulnerabilities can be categorized into two perspectives: at application (high) level, including both the protocol's design and implementation, and at the underlying system environment (low) level. Vulnerability testing is a process of identifying the security holes and weaknesses in the networked systems by various techniques such as injecting faults into the software, analyzing the current state of the system and searching for anomalies. In this paper we focus on the vulnerability testing of the protocol implementation and we choose Megaco protocol as an example for testing since this is a VoIP protocol that will promote carriers to move into VoIP applications. We employ the vulnerability testing methodology in which the test generation is based on a parameter variation scheme and the testing process is based on testing tools applied to TTCN-3 test specification.

Not including this section, the paper is organized as follows. Section 2 provides a brief introduction to the Megaco protocol and the potential vulnerabilities that may exist in the Megaco system. Mutation based vulnerability testing methodology, being the core of the paper, is presented in Section 3. In Section 4, we discuss the vulnerability test suite generation for Megaco and the result of its application to a sample Megaco product. Finally, in Section 5 we present concluding remarks and offers suggestions for future work.

Vuong S., Cai X., Yun L. and Keong Woo W. (2004).
ON VULNERABILITY TESTING OF VOIP SOFTWARE - The Megaco/H.248 System as an Example.
In *Proceedings of the First International Conference on E-Business and Telecommunication Networks*, pages 216-222
DOI: 10.5220/0001406102160222

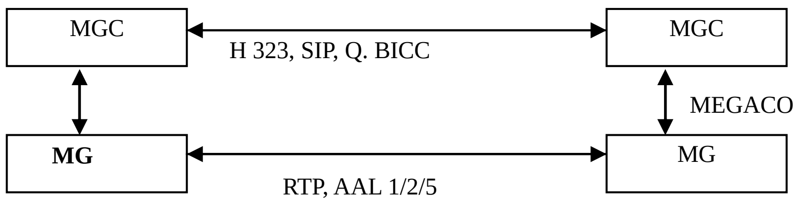

Figure 1: Megaco Protocol between the MG and MGC

2 MEGACO/H.248 SYSTEM AND POTENTIALVULNERABILITIES

Before discussing the vulnerability testing methodology and the testing of Megaco, a brief introduction of the Megaco protocol would be in order.

2.1 Brief Introduction of Megaco

Media gateway control protocols were born out of the need for IP networks to interwork with traditional telephony systems and enable support of large-scale phone-to-phone deployments. Media gateway control protocol decomposes signaling part from the media connection so that the media gateways can work more efficiently. Megaco stands for "Media Gateway Control". Megaco/H.248 is a collaborative effort of the ITU and IETF. Reference for Megaco is available in RFC 3015 (Rfc3015, 2000). Megaco/H248 addresses the relationship between Media Gateways (MG) and Media Gateway Controllers (MGC) as shown in Figure 1. This relationship has a master/slave structure where masters are MGCs (sometimes called call agents or softswitches) and slave devices are MGs which execute commands sent by master devices.

2.2 Megaco Security Features and Potential Vulnerabilities

The Megaco specification requires that the control connection between MG and MGC be protected by IPSec, or in case the underlying operating system does not support IPSec, an interim AH solution should be employed. Despite this protection, if there exists software bugs in the implementation, it is still possible for an attacker to break into a MG from the user end and exploits the software bugs to bring the MGC down. There also exist some insider attacks. Furthermore, infrastructure vulnerabilities that cause the DoS attacks on MGs or misbehaving MGCs are unavoidable. For example, a DoS attack to a MGC could occur when the attacker sends a large amount of UDP packets to the protocol's default port 2944 or 2945, thereby keeping the target MGC busy processing illegal messages, and thus preventing it from using its resources to offer normal service.

Aside from the aforementioned security problems in signalling control, media security is another issue which refers to the prevention of eavesdropping or the altering of a voice stream between caller. In this paper, we only focus on the testing of the software implementation bugs.

3 VULNERABILITY TESTING METHODOLOGY

Our purpose of vulnerability testing is to identify software bugs that may cause security problems, such as buffer overflow, that an intruder could exploit by carefully crafting the input data in an attempt to compromise the security of the system. We are taking a black box testing approach based on syntax testing. The *errors* to be injected are generated based on parameter mutation. Details are described in the subsequent sections.

3.1 The Testing Methodology

Our testing methodology is shown in Figure 2. The general idea is to use syntax testing with parameter variation (mutation) to create error sentences.

We start from the specification of the protocol, then model the protocol in a formal description language such as SDL or use formal grammar notation such as BNF to describe the protocol exchange. The abstract test cases for vulnerability testing are derived from the formal description, but with the parameter mutation to generate *errors* as in the syntax testing. This will be detailed in the next section. We describe the abstract vulnerability test suite (VTS) in Testing and Test Control Notation version 3 (TTCN-3).

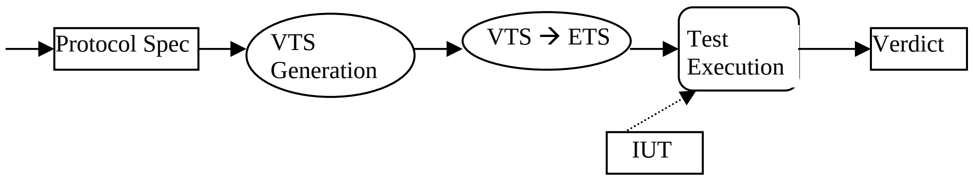

Figure 2: Vulnerability Testing Process

The reason we choose TTCN-3 for vulnerability test specification is that, unlike TTCN-2 which is specifically designed for conformance testing, TTCN-3 has been developed with the objective of being a flexible and powerful test specification language applicable for all types of testing, including robustness testing. Although TTCN-3 is conceptually extended from TTCN-2, it is syntactically very different from TTCN-2, and is therefore named differently. Being a standard, the use of TTCN-3 will ensure wide portability and understandability, and the generated test suite will be shared with the community for verification without any modification required.

After the vulnerable test suite (VTS) is specified, we need to use TTCN-3 tools to convert the VTS to Executable Test Suite (ETS). Then we can feed the ETS into the test tool (test execution module) to execute against the implementation under test (IUT), and finally we obtain the test verdicts. The process of ETS translation and execution using a toolset (e.g. *TTthree* and *µTTman*) is illustrated in Figure 3. This process was applied to a real-life Megaco product IUT. First, the VTS is generated via some creative parameter variation heuristics and specified in TTCN-3, and the VTS specification is stored in the file *megacoTest.ttcn3*. The abstract VTS specification in TTCN-3 is then transformed into an ETS (via the *TTthree* tool) for actual application in the vulnerability testing of the IUT (via *µTTman* tool).

First, the VTS specification in TTCN-3 is compiled into a programming language, e.g. Java or C/C++. We are using the *TTthree* tool developed by Testing Technologies™ to compile the VTS into Java, specifically from *megacoTest.ttcn3* to *megacoTest.jar*. Next, we develop a test adapter, written in Java, to map the abstract test system interface referred to in TTCN-3 into a real test system connected to the IUT. An example is the mapping of an abstract port in TTCN-3 into a real TCP or UDP port opening. In addition, we develop an *encoder*, also written in Java, to map TTCN-3 data structures into real messages for injecting the test messages, and a *decoder* to map real messages into TTCN-3 data structures for receiving the responses. For convenience, the above three java programs are compiled into a single file: *megacoTestAdapter.jar*.

To facilitate the testing, it is desirable to have a test manager to inject the test cases in sequences, receive the responses, and display the results. We are using the *µTTman* tool, also developed by Testing Technologies. The *µTTman* tool uses a Module Loader File written in XML to reference the test suite, the test adapter and the codec (voice encoder/decoder) to be used, which in our case, are the *megacoTest.jar* and *megacoTestAdapter.jar*.

3.2 VTS Generation – Mutation Method

We adopt syntax testing method to identify software bugs. Syntax testing is a way to test system robustness. In a sense, if a system is very robust, then it is really hard to be broken into, hence more secure. In syntax testing, the test cases, i.e. the input to the software, are created based on the specifications in languages understood by the interface software (Beizer, 1990). The motivation for syntax testing springs from the fact that each interface has a language, whether it is hidden or open, from which effective tests can be created with a relatively small effort (Kaksonen, 2000). The communication protocol between two entities is a perfect interface language.

The syntax for the interface language is usually represented by formal grammars such as "Backus Naur Form" (BNF). Following the rules of the grammar, the defined formal language will produce a right sentence which is basically sequences of bytes. The selection of test cases in syntax testing could start with single-error sentences that follow the defined syntax. By single-error, we mean only one grammar element in the right sentence is replaced with some error or exceptional value. An exceptional element value is an input that has been designed to provoke undesired behavior in the implementation, and we regard these as parameter mutation (or parameter variation) from the normal valid value. An example would be to replace a valid integer value by a float number in the sentence. The exceptional input is usually not considered seriously

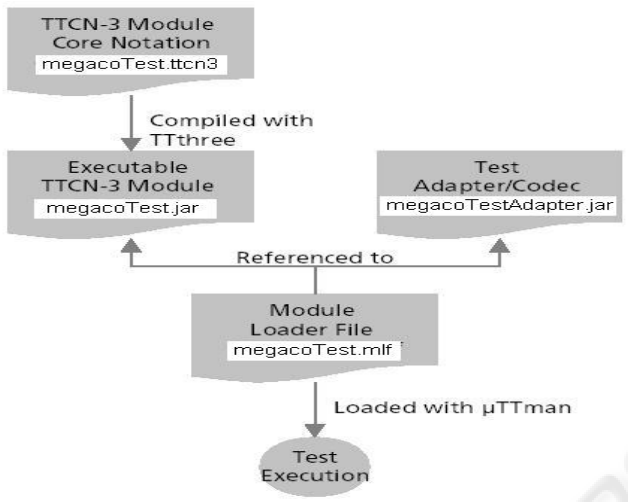

Figure 3**:** Tools for VTS-to-ETS Translation and ETS Execution

by most software developers during the developing process, thus easily leading to havoc when being exploited. The provocation knowledge is often acquired by experience and may be protocol specific. Single-error testing is likely to reveal most faults assuming the faults are mutually independent and a fault is triggered by one error in a sentence. After all the sentences with one error are tried, the testing proceeds to pairs of errors, three error combinations, and so on. The number of test cases grows exponentially by the number of combined errors (Beizer, 1990).

4 VULNERABILITY TESTING OF EXAMPLE MEGACO/H.248 SOFTWARE

Based on the mutation concept and syntax testing approach, we design test suite and test a sample Megaco system in this section. At the current stage of the testing process, we have not covered the test suite generation for the whole protocol yet and our test focuses only on the Megaco command "***ServiceChange***".

4.1 VTS Generation for Megaco

To perform vulnerability testing using the syntax testing approach, first we need to obtain the formal grammar that defines the protocol syntax. We can find the complete augmented BNF (ABNF) specification for Megaco in RFC 3015. A sample excerpt of the ABNF specification for "*ServiceChangeRequest*" that we employ in the vulnerability testing is shown below:

megacoMessage = LWSP [authenHeader SEP] message
message = MegacopToken SLASH Version SEP mId SEP messageBody
mId = (((domainAddress / domainName)[":" portNumber]) / mtpAddress / deviceName)
messageBody = (errorDescriptor / transactionList)
*transactionList = 1 * (transactionRequest / transactionReply /transactionPending / transactionResponseAck)*
*transactionRequest = TransToken EQUAL TransactionID LBRKT actionRequest *(COMMA actionRequest) RBRKT*
serviceChangeRequest = ServiceChangeToken EQUAL TerminationID LBRKT serviceChangeDescriptor RBRKT

An example of a valid and typical "ServiceChange" Megaco message that the protocol grammar should be able to generate is given below:

```
MEGACO/1 [192.168.1.101]
Transaction = 9998 { Context = - {
            ServiceChange = ROOT {Services {
                  Method=Restart,
                  ServiceChangeAddress=44445,
Profile=ResGW/1}}}}
```

We choose the command "*ServiceChange*" sending from MG to MGC for registration as our starting point of the vulnerability testing process. By analyzing the grammar component in this command and following the guideline of the PROTOS project (PROTOS, 1999), we designed the exceptional element categories, partially shown in Table 1.

Table 1: Exceptional element categories

Name	Description
ipv4-ascii	Malformed IPv4 addresses in ASCII and special purpose addresses
overflow-general	“a“ (0x61) character overflows up to 128KB
utf-8	Malformed UTF-8 sequences
overflow-space	Overflows of “ “ up to 128KB
fmtstring	Format strings
megaco-version	Malformed ”MEGACO/1”

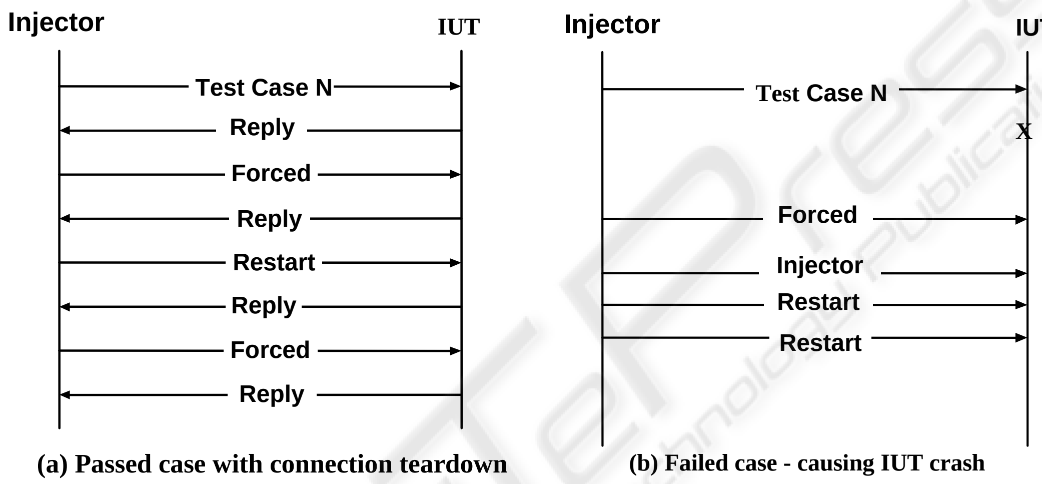

Figure 4: Vulnerability Test Sequence Diagram

Our test cases can be categorized in groups according to which element (parameter) in the sentence (command) that we want to fill it with the exceptional value. A total of 1771 test cases are generated with some groups listed in Table 2.

Table 2: Megaco Vulnerability Test Groups

Name	**Exceptional Elements**	**Test Cases**
MegacopToken	Empty, overflow-general, overflow-space, fmtstring, utf-8, ansi-escape	193
Version	megaco-version	75
DomainName	ipv4-ascii	106
TransToken	Empty, overflow-general, overflow-space, fmtstring, utf-8, ansi-escape	193

An example test case in the test group “*TransToken*” with the exceptional element category of “overflow-general” will replace the valid value in the “*TransactionToken*” field with a string containing variable length of character ‘a’ so that we can use it to test whether the IUT will have buffer overflow problem.

4.2 Test Execution on an Example IUT

We have compared the implementations available to us that include the Megaco Erlang toolkit, a test equipment that has Megaco function implemented and an evaluation version of a viable commercial Megaco software. In the initial stage of our testing development, the evaluation version of the commercial software was chosen as a sample implementation to help us create a test suite for vulnerability testing.

The vulnerability test sequence diagrams are shown in Figure 4 with the left side being for passed cases and the right side for failed ones. *Forced* command in the chart serves the purpose of tearing down the Megaco connection. For the passed cases, the test packet is first sent to the IUT. Then, if a reply is returned by the IUT, it is treated as a passed case and it is the reply is disregarded by the tester. After a certain period of time, a valid case is sent to the IUT to verify if the IUT still functions normally. On the other hand, when a failed test case is applied, we expect that it will crash the IUT or make the IUT non-functional. This can be verified by three consecutive retries without receiving a response, thus indicating that the service is no longer available. All the test results are logged in a log file for further analysis or visualization. A failed test case is shown in the appendix.

All the 1771 of test cases have been tested with the sample MGC via UDP and TCP transport. The partial result is summarized in Table 3. Altogether, we found 10 failed test cases when conducting test over TCP and 5 over UDP transport, both in the test group of "DomainName".

Table 3: Vulnerability Test Result

Test Group	UDP Result	TCP Result
MegacopToken	Passed	Passed
Version	Passed	Passed
DomainName	***Failed***	***Failed***
TransToken	Passed	Passed

5 CONCLUSION AND FUTURE WORK

We generated vulnerability test suite for the Megaco protocol based on the method of robustness testing in which heuristically derived exceptional elements are used to produce mutations of the correct behaviour to create the vulnerability test suite. Since TTCN-3 is a standard test specification language for various types of testing, including robustness testing, the generated vulnerability test suite (VTS) for the Megaco protocol is specified in TTCN-3 and TTCN-3-based tools (e.g. *TTthree* and *µTTman*) are used in the vulnerability testing process. The result of applying the generated VTS to a sample IUT demonstrated well the effectiveness of the testing approach.

What we have done so far is simply to test a single command (*Service*) with a single error (a single parameter variation) while the program (IUT) is at a certain state. It is reasonable to expect that the protocol behaviour and thus IUT behaviour in response to a certain input is state sensitive even from a vulnerability point of view. Not all sentences are acceptable in every possible state of a software component. A state-dependent error can be generated by inputting a correct sentence in an incorrect state. Therefore, the next step of our plan will be to consider the protocol Finite State Machine (FSM) and perform a state-dependent mutation for vulnerability test suite generation, and to consider multiple errors (i.e. muti-mutations) in a single vulnerability test case.

ACKNOWLEDGEMENTS

This research is supported in part by Industry Canada - Directorate of Telecom Engineering and Certification. We would like express our gratitude to Peter Chau, Colman Ho, Os Monkewich, Jinmei Yang, Ardashir Bahi, Yan Bai and Sergio Gonzalez for the fruitful discussions and comments that led to the improvement of earlier drafts of the paper.

REFERENCES

RFC3015, 2000: http://www.ietf.org/rfc/rfc3015.txt

Kaksonen, R., Laakso, M., Takanen, A., 2000, Vulnerability Analysis of Software through Syntax Testing, Available: *http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/index.html*

Beizer B., 1990, Software Testing Techniques, Second Edition, ISBN 0-442-20672-0

PROTOS, 1999-2003, "PROTOS - Security Testing of Protocol Implementations". University of Oulu. *http://www.ee.oulu.fi/research/ouspg/protos*.

APPENDIX

Sample Vulnerability Test Cases for Megaco

```
# A passed test case:

MEGACO/1 [aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa]
Transaction = 9998 {
Context = - {ServiceChange = ROOT
                      {Services { Method=Restart}
        }
       }
}

# Failed test case: long string of "a" IP_addr

MEGACO/1
[aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa]
Transaction = 9998 {
```

```
Context = - {ServiceChange = ROOT
                         {Services {Method=Restart}
          }
        }
}

# Reply received from MGC
MEGACO/1 [142.103.10.92]
Reply=9998{
Context=-{ServiceChange = ROOT
         {Services{ServiceChangeAddress=2944}
         }
     }
}

# Forced command message to disconnect

MEGACO/1 [192.168.1.101]
Transaction = 9998 {
Context = - {ServiceChange = line/1
        {Services {Method=Forced, Reason="905
Termination taken out of service"}
         }
    }
}
```