A Vulnerability Assessment Framework for the IMS

Prepared by: Denver Abrey

Supervised by: Neco Ventura

Dissertation presented for the Degree of MSc (Eng) in the Department of Electrical Engineering at the University of Cape Town

Declaration

I declare that this thesis is my own work. All information drawn from various sources and used within this thesis has been referenced and acknowledged.

This thesis is being submitted for the degree of Master of Science in Electrical Engineering at the University of Cape Town.

This thesis has not been submitted before for any degree or examination in any other university.

Denver Abrey.....

Date.....

Acknowledgements

I would lke to thank the following people for their invaluable assistance during the course of this project:

Neco Ventura, my supervisor for his guidance and patience throughout the project. Thank you for your input, guidance and supervision.

Richard Spiers, for the advice, motivation and assistance with the IMS and SIP.

My parents, Desiree and Gavin Abrey, for their constant support and love during my time at UCT.

Abstract

With multimedia services being made available via more and more devices to end users, it is no longer feasible to develop a delivery platform for each new type of service. The IP multimedia subsystem (IMS) aims to provide a unified service delivery platform capable of supporting a wide range of multimedia, data and voice services. It has been developed with a focus on content delivery and rich communications, and has already begun to replace existing legacy GSM network components. The IMS is intended to be an access agnostic platform, capable of providing services over both mobile and fixed networks using a multi-access all-IP platform. By providing a feature-rich all IP platform, operators are able to deploy open IP-based networks, allowing for easy deployment and development of new, rich multimedia centric communication services. With the IMS in place, an operator may take the role of a service broker, providing them with far more revenue generating opportunities than just traditional voice and data. Application services may leverage the functionality provided by the IMS to create new services quickly while allowing them to be easily integrated into the network infrastructure.

With the IMS gaining more and more attention from telecoms operators, and already being adopted by some, the ability to assess the security of the system becomes critical to the success of the IMS platform. While the 3GPP has placed emphasis on security throughout the development of the IMS, implementation is left up to vendors looking to create their own IMS systems. Implementation specific vulnerabilities may be missed by standard quality assurance testing, as they may be triggered only by boundary or near boundary conditions, or non-standard or unexpected state transitions. While quality assurance testing generally will ensure standards compliance and that components function as expected under normal conditions, the effects of unexpected or malicious input to the system should also be examined. A number of software testing techniques exist, some of which may be performed manually while others can be automated. Manual and automated testing may be applied with or without knowledge of the source code or internal workings of the software to be tested. Test cases may then be generated, manually or automatically, using information from either the source code itself, or by examining the specification defining the way in which the software is expected to behave given various inputs.

A style of testing called "Fuzzing" is particularly suited to security testing implementation specific vulnerabilities as it is targeted towards the particular software being tested. It is also automated, capable of creating nearly correct test cases and generating a very large number of test cases to ensure greater code coverage than manual testing. This style of testing is very much a "brute force" approach, as automatically generated test cases may or may not expose any underlying errors or vulnerabilities.

By controlling the way in which test cases are generated however, the probability of exposing possible implementation errors can be improved. This is done by taking note of what types of data are expected and where, by the software being tested. Tests can also be more focussed around boundary values, to expose errors relating to poor sanity checks. By generating and mutating input data while ensuring that the underlying expected structure or format of the data is still at least partially present, a vast number of test cases can be generated that both pass validation checks and easily expose common programming and security flaws.

Generating specially crafted test cases that are likely to pass validation checks targeted at a specific piece of software or protocol has been demonstrated to be effective in discovering various types of security flaws within a number of components that form part of the Open Source IMS core. The prototype developed to create and utilize these test cases can potentially be extended to carrier implementations of IMS frameworks too.

The prototype was based on an existing open source SIP-capable fuzzing framework and modifications were made to allow the framework to interact with selected IMS components. The developed framework was then run against SIP capable components within Open Source IMS core, and it was observed to be effective in locating a number of flaws within the components. Both stateful and stateless fuzzers were able to generate test cases which triggered errors, with stateful fuzzing uncovering a broader range of errors. The prototype showed that fuzzing is a viable method of assessing the implementation of IMS SIP component.

Contents

List of Figures

List of Tables

Nomenclature

3G	Third Generation
3GPP	3rd Generation Partnership Project
AAA	Authentication Authorization Accounting
AKA	Authentication and Key Agreement
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
AUTN	Authentication Token
CK	Ciphering Key
CN	Core Network
COM	Component Object Model
CSS	Cascading Style Sheets
DDoS	Distributed Denial of Service
DoS	Denial of Service
DSL	Domain Specific Language
DWORD	Double Word
FTP	File Transfer Protocol
GDB	GNU Debugger
GSM	Global System for Mobile communications

H264	MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC)
HLR	Home Location Register
HN	Home Network
HSS	Home Subscriber Server
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I-CSCF	Interrogating Call Session Control Function
IK	Integrity Key
IMPI	IP Multimedia Private Identity
IMPU	IP Multimedia Public Identity
IMS	IP Multimedia Subsystem
IP	Internet Protocol
IPSec	Internet Protocol Security
ISIM	IP Multimedia Services Identity Module
LTE	Long Term Evolution
MAC	Message Authentication Code
MD5	Message-Digest 5
NX	No-Execute
P-CSCF	Proxy Call Session Control Function
PC	Personal Computer
QoS	Quality of Service
RADIUS	Remote Access Dial-In User Service
RAND	Random Challenge
RCE	Remote Code Execution

RES	Authentication Response
RPC	Remote Procedure Call
RTP	Real-time Transport Protocol
S-CSCF	Serving Call Session Control Function
SBC	Session Border Controller
SDF	Service Delivery Framework
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SQL	Simple Query Language
SUT	Software Under Test
TLS	Transport Layer Security
UA	User Agent
UE	User Equipment
UICC	Universal Integrated Circuit Card
UMTS	Universal Mobile Telecommunications System
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VN	Visited Network
VoIP	Voice Over IP
XMAC	Expected MAC
XRES	Expected Response

Chapter 1

Introduction

Voice over IP (VoIP) has been steadily gaining popularity, as has both demand for and consumption of digital media. Data usage within both fixed line and mobile networks has increased exponentially because of this. End users are making use of far more services than they did in years past. Today, one expects to be able to utilise a range of services from many different devices. Services such as web browsing, instant messaging, presence, voice over IP, streaming media, video calling and conferencing, application sharing, telephony, unified messaging, multimedia content, etc. have traditionally been delivered over various service-specific platforms, usually operating over IP networks such as the Internet. Accessibility of such services via the Internet has also created demand for new, innovative services able to provide for the rapidly evolving needs of today's increasingly tech-savvy end users. While service-specific platforms work well, they are often not extensible and do not support the development of new, differing services. Developing entirely new delivery frameworks instead of reusing existing components also increases development time. Delivering real-time services over the best effort Internet can also provide a sub-optimal experience for users, particularly for services like voice over IP or video conferencing. To remedy this, a platform is needed that can support a wide range of multimedia services over various different communication networks, and provide some level of quality of service (QoS) to ensure a good experience for users. This platform should also allow for reusability of common components to the greatest extent possible.

1.1 The IP Multimedia Subsystem

The IP Multimedia Subsystem (IMS) standards define a generic architecture for establishing sessions to deliver Voice over IP (VoIP) and multimedia services. It effectively provides a platform for supporting multimedia service creation and deployment. This allows operators to maintain their role as service providers, rather than just becoming raw "bit pipe" or data providers as focus shifts away from traditional circuit switched voice services. The IMS has been proposed by the 3rd Generation Partnership Project (3GPP), who have created the technical standards which define the IMS architecture. The Session Initiation Protocol (SIP) [6, 7] is used extensively within this architecture.

SIP is an IETF standardised signalling protocol used for controlling multimedia sessions such as voice and video calls over IP networks. SIP based architectures include network elements such as the User Agent (UA) and various servers. The UA is the logical end point used to create or receive SIP messages, and therefore manage a SIP session. While UAs can communicate directly, this is often infeasible and most implementations make use of at least a proxy and registration server. Additional servers such as redirection, session border controller (SBC) and gateway servers are also possible and may be used where required. SIP has gained popularity largely due to its open architecture and extensibility, as well as its familiarity due to being similar to the Hypertext Transfer Protocol (HTTP). One popular extension is SIP Instant Messaging [8], and is utilised within the IMS. SIP is based on Hypertext Transfer Protocol (HTTP) and utilises ASCII encoding, making it easy to read and understand. Unlike binary protocols, a simple raw capture of SIP is fairly readable by someone familiar with the protocol.

The Real-time Transport Protocol (RTP) is commonly used with SIP to transport any kind of media digitized with an appropriate codec. Using the G.729 or G.711 codecs, RTP can be used to transport voice, or video using a format such as H264 or MPEG.

The IMS architecture builds on SIP based VoIP and 3G architectures by adding a number of important features, detailed below.

Quality of Service (QoS)

While the Internet and many packet-switched IP networks are "best effort" services, IMS requires QoS functionality on the network level within a session to ensure that quality of service criteria can be met. Most existing packet switched networks don't guarantee quality of service, or cannot provide end-to-end QoS guarantees, and end user experience can vary depending on network conditions. Network congestion or outages can result in lost packets, requiring retransmits which would affect the quality of service of realtime applications. The IMS ensures end-to-end QoS by identifying session flows at the

bearer level and prioritising the routing of packets to ensure that real-time services operate without interruption.

Flexible Charging

IMS provides operators with the ability to charge for multimedia services instead of being limited to only charging for quantity of data consumed. Billing based on data volume alone is inflexible and may be inappropriate for many multimedia services.

Integrated services and extensibility

The IMS provides a number of built in multimedia services, as well as providing a platform which third party services can leverage to create new and innovative services.

The number of IMS deployments has been steadily increasing. IMS is also seen as the supporting architecture for voice and multimedia service delivery within LTE networks, which therefore means deployments will be expected to serve large numbers of users reliably and with a high degree of privacy and security. A report by Ericsson in April 2012 claims 63 commercial IMS systems deployed, with 94 commercial contracts worldwide with prominent operators [9].

The IMS standards have been developed with security in mind from their inception. Security policies have evolved from GSM and UMTS, and place a strong emphasis on mutual authentication between UA and the IMS core, integrity and privacy of messages [1]. While a large amount of effort has been placed on creating a set of standards that is adept at ensuring security of various IMS components, as well as the UE, it is still difficult to ensure that a given implementation of these standards is secure without some form of, preferably automated, software testing and verification.

1.2 Software security and vulnerability discovery

Many techniques can be used to assess how secure a piece of software is. Source code auditing can help identify common pitfalls and may catch some possible security related errors, but must be performed by someone with a deep understanding of both the language used to create the software, the operation of the software itself as well as the conditions under which the software will be used. Unfortunately, this technique cannot be applied to software where the source code is not available, for example, a proprietary implementation of the IMS. Black box testing can be performed on software without the availability of the source code. One prominent method of black box testing that has become popular in security testing is "fuzzing". This technique involves sending unintended or unexpected input to

the software under test and observing the results. The success of fuzz testing or "fuzzing" a piece of software is measured solely on the results of the testing.

Fuzz testing can take many forms, and may be applied to network protocols, files, input fields, environment variables or any other means that a software expects user input. By manipulating these fields in various ways, security vulnerabilities can often be exposed within the software being tested. The data and way in which the input data is manipulated depends on the fuzzer. The input data can be completely random binary or text of random length, or a more intelligent approach can be taken whereby only pieces of the inputs are manipulated in such a way that any basic validation checks within the software do not reject the manipulated input.

Many tools and even frameworks have been created with the aim of performing fuzz testing on specific pieces of software, or allowing new dedicated fuzzers to be easily created. Fuzzers themselves can be extremely simple, generating only random data and sending it to the software under test, or they can intelligently construct seemingly valid messages, manipulating only parts of the data while still maintaining a valid overall structure of a given message. These "smart" fuzzers may need to calculate checksums or length fields for their generated messages, and insert them into the correct places to ensure that they are not immediately rejected by validation checks within the software (if any are performed).

1.3 Research Motivation

The aim of the project was to assess various methods of vulnerability discovery by using software testing techniques. Once these had been evaluated, a basic vulnerability assessment framework for the IMS SIP protocol was created. The framework was tested against the OpenIMS Core implementation, and its results and performance are detailed. Overall, the efficacy as well as applicability of fuzz testing to a service delivery platform such as the IMS was evaluated.

1.3.1 Problem Definition

IMS deployments are already being utilised to provide services for both fixed line and mobile operators. A key requirement of a commercial offering of such a service to the general public is security. Users expect that they will not be vulnerable to attacks that could result in fraud, loss of revenue, identity theft, denial of service (DoS) or leaking of sensitive information. There are a number of "common"

or known attacks for both general IP networks and the SIP protocol, that are applicable to IMS deployments. Fortunately, properly securing a given IMS network can remove the risk of many of these attacks. Techniques such as mutual authentication, topology hiding, IPSec encryption all serve to further protect users from possible malicious parties. Ultimately, however, legitimate users still require access to the network to make use of the services they desire, and this leaves at least one attack vector available to malicious parties. Any components of the IMS deployment directly exposed to users (usually the P-CSCF) should be highly resilient to unexpected or undesired input. A component may be exposed to unexpected messages by accident, such as by an Internet wide scan [10], or intentionally due to an attacker attempting to locate weaknesses within the system.

While the 3GPP standards focus heavily on mitigating most known attacks, implementation of these standards may still vary across different vendor offerings. To ensure that a given deployment is in fact secure and able to handle malformed data from a malicious user, a method of verifying the resilience of a given component would be useful.

Many forms of security auditing exist today, and new methods of gauging the security of systems exposed to end users are constantly being developed. Techniques such as white and black box testing can be used to verify that the software behaves as expected under predictable or well defined scenarios. Source code auditing can be performed to locate common programming errors and running pre-defined software tests designed to exercise handling of boundary conditions, exceptional input or predefined malformed messages can assist in locating possible implementation errors. While these testing techniques are effective, they can be further complemented by providing specially crafted dynamically generated random input to the software under test to perform many unique test cases in very little time.

This technique is called "fuzzing" and has been shown to be highly effective in exposing software security flaws [11]. This method of testing often allows at least some possible implementation flaws to be discovered and resolved, before they are able to be exploited by a malicious attacker. A tool to perform "fuzz testing" on the IMS components, both internal and those exposed to the end users, is therefore a necessary and useful addition to existing security auditing methods, allowing the resilience of a component to unexpected attacks and inputs to be gauged.

1.4 Thesis Objectives

This project aims to demonstrate that automation of existing software verification and testing techniques can assist in discovering vulnerabilities specific to an IMS implementation. While many tools exist to perform fuzz testing, they are usually aimed at a specific piece of software or program. Frameworks also exist that can provide many of the necessary building blocks for creating new fuzzers.

This document aims to analyse existing methods of fuzz testing, fuzzing tools, frameworks and their applicability to test the security of and locate vulnerabilities within an IMS deployment. A new tool will then be created by modifying existing software to perform this automated testing of the SIP protocol handling within IMS components. A testbed architecture will be designed and implemented to allow for the tool to be tested against the OpenIMS Core and the results of the tests will be analysed and verified to ensure that they are legitimate vulnerabilities and that the discovered test cases reliably trigger the discovered errors or flaws. Basic investigation into whether or not the software flaws discovered are exploitable will also be researched.

1.5 Scope and limitations

Only open source tools and frameworks are analysed in this document, as closed source tools cannot easily be modified. The testing framework is limited to the SIP protocol within an IMS deployment. While the techniques shown can be extended to most, if not all other protocols used within an IMS implementation (or any protocol for that matter), the scope of this project is limited to the SIP protocol. Fuzz testing is a "brute force" technique, and while it is often successful when properly applied, there is still a possibility that the technique can not yield any useful results. Fuzz testing, by its "brute force" nature, is also not an exhaustive means of testing the ability of a given piece of software to handle unexpected or malformed input. The technique may well lead to the discovery and mitigation of many flaws, but a piece of software cannot be deemed secure by virtue of fuzzing yielding no results (Although this would certainly be an encouraging outcome). Different tools and frameworks can also lead to vastly different results, due to the way in which they generate test cases and mutate input data.

The developed framework serves to demonstrate the feasibility of fuzz testing IMS implementations, and tests only a limited number of the most commonly used SIP messages. As open open source tools and frameworks have been used, only the Open IMS Core was targetted with the developed framework.

Applying the framework to another vendor's IMS product is left as future work.

IMS terminology can be vague due to the constant evolution of various IMS technical standards. This document deals with the components of the IMS Core Network (CN), or "IMS Core". The assessment of other protocols used within the IMS architecture are left as future work.

1.6 Document Layout

The remainder of this document is outlined below:

Chapter 2 provides an overview of the IMS, its core protocol and security architecture, the key components and existing IMS attacks. Software testing and vulnerability discovery techniques are briefly outlined, as well as existing tools and frameworks for performing automated fuzz testing. A number of different software testing methods are detailed.

Chapter 3 discusses the design considerations for an automated vulnerability discovery tool aimed at IMS deployments. The suitability of the software testing techniques, tools and frameworks for use in creating a fuzzing tool for the IMS are then analysed. A testbed architecture and design of a tool to perform fuzz testing of IMS components is proposed.

Chapter 4 details the architecture of the proposed testbed and fuzzing tool, as well as the implementation of both the testbed and aforementioned testing tool. The framework makes use of the OpenIMS core, and builds on existing open source software and tools. The objectives, requirements and limitations of the testbed are discussed.

Chapter 5 outlines the tools developed and testing performed within the testbed. The results of the tests performed are analysed to determine whether the developed tools, testbed and overall architecture were effective in discovering vulnerabilities and security issues within the components tested.

Chapter 6 draws a set of conclusions based on the evaluation of the results performed in Chapter 5. Recommendations for extending the functionality of the developed prototype, and techniques that would be useful for increasing its effectiveness are outlined.

Chapter 2

Literature Review

2.1 IMS overview

IMS is a global, access independent, standards-based IP connectivity and service control architecture that enables various types of multimedia services to be delivered to end users by using common Internet-based protocols. It is being developed and in some cases deployed to replace legacy fixed-line and mobile networks, while providing network, service and device convergence. Network convergence aims to make the same services available across all networks by migrating existing services to the IMS, and ensuring they are accessible via many different IP networks. Device convergence is achieved by ensuring that multiple services are available on many different devices, be it a tablet, smartphone or PC. Service convergence is achieved by utilising the standards based Service Delivery Framework (SDF) within the IMS to build new services. Complexity is reduced by using the SDF, and services can more readily work with one another if built on top of the SDF [12].

2.1.1 IMS Core Key components

The Call Session Control Function (CSCF) establishes, monitors, supports and releases multimedia sessions and manages the user's service interactions. It can play three different roles: Serving-, Proxy- or Interrogating- Call Session Control Function (S-, P- and I-CSCF). The S-CSCF is the proxy server controls the communication session. It invokes the Applications Servers related to the requested service. It is always located within the user's home network. The P-CSCF is the first contact point for SIP

user agents, or the attachment point for the user equipment. The I-CSCF provides a gateway to other domains. It is used for topology hiding, or if several S-CSCFs are located within the same domain [13].

2.1.2 Related IMS Core protocols

Session Initiation Protocol (SIP) is the main signalling protocol used within the IMS.[6, 7] SIP provides considerable flexibility, and can be secured through the use of encryption. IMS SIP is an enhanced version of SIP, including several extensions as described in 3GPP TS 24.229 [14]. The main purpose of SIP is the establishment, modification and termination of multimedia session between two terminals. SIP message bodies are described using the Session Description Protocol (SDP). SDP defines a syntax for describing media flows by specifying address, port, media type, encoding etc, [15, 16]. SIP is the key protocol within the IMS architecture [13].

Diameter is an Authentication, Authorization and Accounting (AAA) protocol, that succeeds the Remote Access Dial In User Service (RADIUS) protocol.[17] Diameter is secured using IPSEC or TLS, and is used within the IMS service framework by the I-, S-CSCF, and Application Servers (ASs) to communicate with the Home Subscriber Server (HSS).

2.2 IMS Security Architecture

The IMS architecture brings with it a number of significant security challenges that need to be addressed by carriers as IMS becomes more widely deployed. As the architecture of the IMS is generally open and distributed, it provides flexibility in implementation and deployment. This also unfortunately complicates the task of security all the various interface points within it. The 3GPP aims to provide at least the same level of security and confidentiality as traditional 2nd generation (GSM) systems, as well as improve where possible [18].

The IMS security mechanisms are divided into two separate parts: access security and network domain security. Access security includes authentication mechanisms and traffic protection between the UA and core network. Network domain security specified in [19] includes traffic protection between network elements within the IMS architecture, and makes provisions for roaming and non-roaming scenarios [1, 20].

As shown in 2.1, there are five different security associations and different needs for security protection for IMS, numbered 1-5. The figure demonstrates the relationship between the UA (shown as UE in

figure) and IMS core network. Within the UA, the IMS authentication key and functions are stored on a Universal Integrated Circuit Card (UICC) and the IMS Subscriber Identity Module (ISIM) indicates a collection of IMS related security information and UICC functions [20].

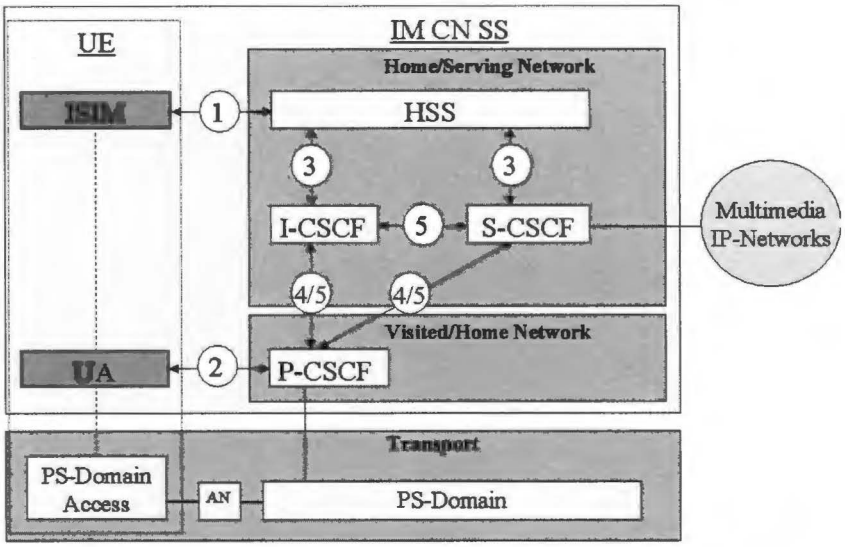

Figure 2.1: IMS Security Architecture [1]

1. Provides mutual authentication. The HSS delegates the performance of subscriber authentication to the S-CSCF, but the HSS is responsible for generating the keys and challenges [1].

2. Provides a secure link and a security association between the UA and P-CSCF. Data origin authentication is provided.

3. Provides security within the network domain internally for the Cx-interface. This association is important for securing the keys and challenges during UA registration [18].

4. Provides security between different networks for SIP capable nodes. This security association is only applicable when the P-CSCF resides in a visited network (VN), otherwise 5 below applies.

5. Provides security within the network internally between SIP capable nodes. This security association also applies when the P-CSCF resides in the Home Network (HN)

Porter et al. describe IMS security as dealing with the initial secure authentication that takes place between the ISIM and HSS. Once a user device is authenticated and allowed to interact with the

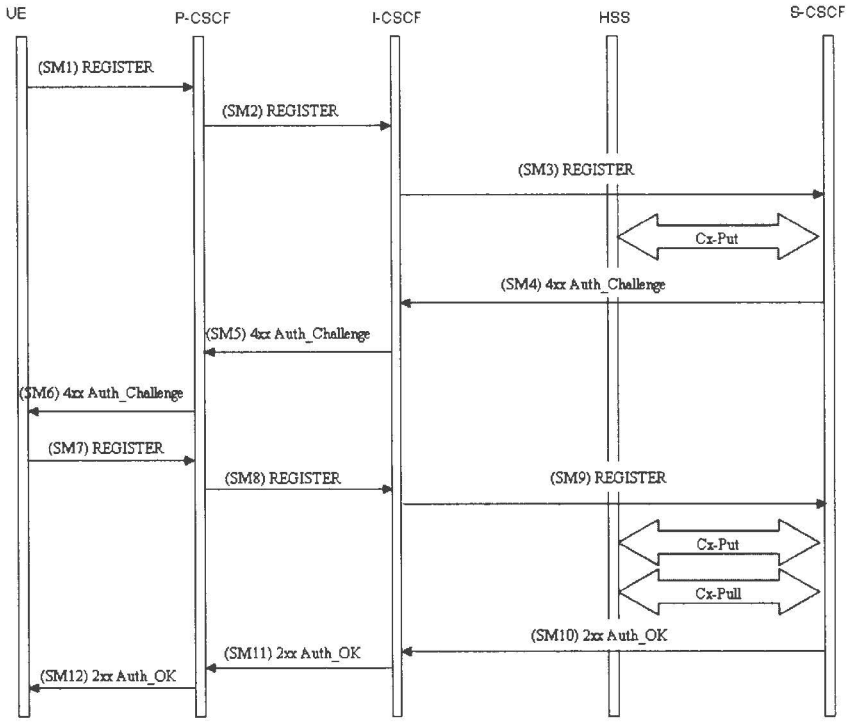

Figure 2.2: IMS Authentication Message Flow [2]

IMS, the IMS security framework provides a secure communication channel over which all information is transferred [2]. All UAs are authenticated before being allowed to use an IMS system. The HSS specifies which security algorithms are used and provides authentication credentials used for verifying those provides by the UAs. Subscriber profile information is stored within the HSS of the subscriber's home network. When registering, an S-CSCF is assigned to the subscriber by the I-CSCF. The subscriber profile data is dowloaded to the S-CSCF via the Cx-reference point from the HSS. When remote access is requested to the network, the S-CSCF makes the decision on whether a subscriber is allowed to continue with the request. A security association secures the process to ensure that information integrity and authenticity is retained.

To gain access to the IMS services, the user requires that at least an IMPU is registered with the HSS in advance. The IMPI corresponding to the IMPU must then be authenticated. A UA wishing to be registered sends a SIP REGISTER message to the SIP registrar (the S-CSCF in this case). The message authentication flow is shown in 2.2.

The SIP REGISTER is first passed to the I-CSCF, then to the corresponding S-CSCF. The S-CSCF then retrieves the user's authentication information from the HSS with the Cx-Pull method. The

S-CSCF utilizes Authentication Vectors (AV) to perform authentication and key agreement with the user. The AV consists of five elements: a random number (RAND), an expected response (XRES), a Cipher Key (CK), an Integrity Key (IK) and an authentication token (AUTN). If no valid AVs are held by the S-CSCF, an AV request is sent to the HSS with the number of AVs required.

The S-CSCF uses the received AVs for authentication challenges to users. A SIP 4xx containing the RAND and AUTN is sent to the UA to initiate an authentication challenge as shown in SM4. The challenge includes the IK and CK used by the P-CSCF. The S-CSCF keeps track of the RAND sent to the UA in case of a synchronization failure.

When the P-CSCF receives the SIP 4xx Auth Challenge (SM5), it stores the IK and CK, and forwards the rest of the message containing the RAND and AUTN to the UE. Upon receipt of the challenge (SM6), the UA verifies the MAC, XMAC and sequence number SQN, and if correct, generates and responds with its authentication information. (SM7)

Once the S-CSCF receives the UA's challenge response (SM9), it is validated, and if successful the user is authenticated. A SIP 2xx Auth_OK message is then sent back to the UA to inform it that the registration was successful, and the authentication procedure is complete [2].

The link between the UA and P-CSCF provides an initial attack vector.

2.2.1 Existing IMS attacks

Many well-known attacks have existed and been carried out on IP networks, as well as against services utilising the SIP protocol. Many of these can be carried over and adapted for use against IMS deployements.

F. Park et al. detail a number of practical attacks against IMS core components in [21]. These attacks include the following:

1. Toll fraud by bypassing the P-CSCF and communicating directly with the S-CSCF.

   By communicating directly with the S-CSCF, the UA can bypass the P-CSCF. This allows charging functions to be bypassed, as well as the possibility of impersonating another user [1, 21].

2. Possible disclosure of internal topology to end users.

   If an IMS implementation does not encrypt Via, Record-Route, Route and Path headers, information about the internal workings of an IMS deployment (such as IP addresses of various nodes) may be disclosed [21].

3. Toll fraud using a SIP CANCEL message before a call is answered, and using a direct invite to communicate with the called party thereby bypassing the IMS Core.

   Instead of fully establishing a call, a SIP CANCEL message is sent to end the call before it is answered. A UA may utilise the 180 Ringing message from the S-CSCF to determine the IP address and port number of the called UE, which can then be used to send a direct INVITE to the called UE. This completely bypasses the IMS core and avoids any charges [21].

4. Denial of Service by flooding the P-CSCF with REGISTER messages.

   Sending a large number of REGISTER messages can cause legitimate REGISTER requests not to be handled.

5. Man-in-the-Middle attack when AKA and IPSec are not utilised.

   If IPSec is not being utilised and an attacker is utilising the same layer 2 network as the IMS core and victim, the attacker can intercept and respond to INVITE requests with a SIP 302 Moved Temporarily message contaning the SIP URI or the attacker, thereby hijacking the call [21].

6. Forging SIP BYE packets to end a call.

   By forging a SIP BYE packet and sending it to directly to the UA or via the P-CSCF, a call may be ended by an attacker. This is possible as UEs and P-CSCFs usually do not authenticate BYE packets [21].

As the author mentions, the above attacks are possible when the guidelines for securing IMS deployments are not strictly adhered to, or due to backwards compatibility compromises. They are however not specific to an implementation, and generally do not expose implementation specific vulnerabilities, but rather deployment specific issues.

2.3 Software testing

2.3.1 Overview

Software testing approaches broadly fall into two categories, static and dynamic testing. Static testing refers to testing the software without executing it, and usually involves examinations, reviews or walk throughs. Dynamic testing involves actually executing the software, and working through various test

cases. Software testing approaches can also be classed as either black box or white box testing, both of which can be performed in a static or dynamic manner [22].

Static white-box testing involves examining the written-source code of the software[22], while static black-box testing involves examining the specification of the software and attempting to identify problems before they are written into the software.

Dynamic white-box testing involves testing the software with insight into the inner workings of the software itself, and basing the tests on information about the way software functions internally. In contrast, dynamic black-box testing involves testing the software without knowing how it works, or without knowledge of the internal functioning of the software. The software is treated as an opaque or black box, the contents of which are treated as unknown.

While all of the described techniques are useful in software testing, dynamic testing is most useful in locating implementation specific errors. Furthermore, dynamic black-box testing provides a useful method of testing the implementation of a piece of software without knowledge of its internals. While this process can be performed manually, it is often advantageous and preferable to automate it.

2.4 Vulnerability discovery techniques

M. Sutton et al. provide an overview of various methodologies for vulnerability discovery in [11]. The approaches mentioned in the software testing overview also provide useful starting points for vulnerability discovery. The authors detail three high level methods of vulnerability discovery, white box, black box and grey box testing. The differences between these techniques lie in the amount of information available to the researcher. White box testing is performed with access to the source code of the software, design documents and possibly even the programmers themselves. In contrast, black box testing is performed with no knowledge of the inner workings of the software, and can be seen as blind testing. Between these two extremes lies grey box testing, which has varying definitions. Grey box testing is usually performed with access to compiled binaries, and possibly some documentation [11].

2.4.1 White Box Testing

The authors detail two primary methods of white box testing in [11], source code review and use of automated source code analysis tools. Manual source code review can be highly effective, but can

quickly become impractical when dealing with medium to large sized projects with tens of thousands of lines of code. The use of source code review tools resolves this problem to some extent, but these tools can only perform heuristic or signature based analysis and cannot definitively state that a given bit of code is secure or contains vulnerabilities. They rather provide insight into *potential* vulnerabilities.

2.4.2 Black Box Testing

According to [11], black box testing is performed with no insight into the inner workings of the software under test. The only information available to the tester is what can be directly observed from the behaviour of the software during testing. Input can be manipulated and output observed, without any understanding of how it is processed. Black box testing can be performed manually or by using automated tools. Manual testing methods are often not particularly effective, unless testing for a specific vulnerability across many pieces of software suspected of having the same vulnerability.

2.4.3 Grey box testing

Grey box testing is an approach that falls somewhere between white box and black box testing. The authors of [11] define it as black box testing with additional insight provided by reverse engineering (also called reverse code engineering, or RCE) a given binary file. RCE generally refers to the process used to determine the functionality of a compiled binary without access to its source code. While the source code usually can never be recovered just from a binary, it is possible to obtain an assembly code representation of the binary, which with some technical skill can be used to locate interesting and possibly insecure code segments within the binary itself.

2.5 Fuzz Testing

According to [23], fuzzing is usually defined as a block-box software testing technique. The authors describe fuzzing as a technique which works by providing a program with specially crafted input in an attempt to trigger errors within the software that would indicate the presence of bugs and possibly crash the software. This process is iterative and repeated for as long as deemed necessary by the tester. The presence of the discovered errors and crashes are usually indicative of the existence of security vulnerabilities, which can then be analysed and possibly mitigated. Triggering a bug or crash within a program during fuzzing is viewed as a successful test case. Fuzzing is essentially creating malformed

input for a program under test and observing the results, and is applicable not just to network connected applications, but any software or application capable of accepting input. No further knowledge of the application, such as its internal design or source code is required.

Fuzz testing operates under a number of assumptions as listed in [23]:

- According to Edsger Dijkstra, testing can only show the presence of bugs, but not their absence.
- Testing may never complete due to Turing completeness.
- There are vulnerabilities within the program that can be found.
- Given enough varying inputs and enough iterations, these vulnerabilities can be exposed.

Initially fuzzing tools just created random input and passed this to the application under test. Over the years, frameworks have been developed to aid in the creation of fuzzers, and to make it easier to create fuzzing tools for various protocols. D. Aitel details a block based protocol fuzzing framework in [24], which was adopted by a number of other frameworks. This technique allows random or malformed data of various types to be placed in strategic locations within a message that otherwise appears legitimate. This vastly increases the likelihood of the application accepting the input and attempting to process it. This technique allows a framework to provide the "building blocks" necessary to create a fuzzer, and the supporting environment for creating protocol specific fuzzers.

2.5.1 Fuzzing Methodologies

There are two main techniques used for creating the input that will be provided to an application when fuzzing. These are mutation and generation based fuzzing. A third experimental technique also exists, called evolutionary fuzzing.

Mutation based fuzzing works by modifying known-good input in ways that will likely expose faults within the software being tested. Mutation based fuzzer usually work through the known-good input and substitute each byte, word or string field with random data. This is roughly a "brute-force" technique, and requires little knowledge of the protocol being utilised, or message structure of the input [11]. The mutations may be random or heuristic. Hueristic mutations may include varying string lengths in an effort to expose buffer overflows, or modifiying numerical fields (eg: length) to be very small or very large values [25]. Mutation based fuzzers are fairly easy and quick to implement, and can be effective against software with poor input validation routines. Mutation based fuzzing can

be inefficient, as many mutated inputs will simply be dropped by the software under test, resulting in wasted test cases. This is however offset by the relative ease of implementation and automation of mutation based fuzzers.

Generation based fuzzing is a more advanced technique that utilizes knowledge of the protocol message format of the software under test to create test cases. This method of fuzzing requires prior research and knowledge of the protocol or message format, and therefore requires more effort, planning and development time than mutation based fuzzing. A grammar of the protocol is then created, with parts of the input being identified as static or "fuzzable". This is then used by the fuzzing tool or framework to create test cases, where the "fuzzable" portions are substituted with data conforming to the type specified in the grammar. For example, special characters may be inserted into string fields (ie: adding '%' may expose SQL injection related errors or format string problems). Arbitrary length strings may also be substituted in the hope of locating a buffer overflow vulnerability [11]. Generation based fuzzing aims to create test cases that vary enough from valid data to expose errors within the software, but are still interpreted as valid by the software and hence still processed instead of merely being discarded. Generation based fuzzing generally produces higher quality test cases, but at the expense of more research into the protocol, message or file format used by the software being tested [25].

Evolutionary fuzzing attempts to build in generation based fuzzing by adjusting test cases based on the feedback from the software under test. They use an evolutionary, often genetic algorithm to generate test cases based on the output of a fitness function, that attempts to gauge the effectiveness of test cases based on the feedback from the software under test [26].

2.5.2 Types of Fuzzers

The main purpose of fuzzers and related tools is to generate input that is likely to expose flaws in a piece of software. Fuzzers can achieve this goal by providing the input in various ways, depending on what is being tested and what the software expects. Fuzzing tools generally focus on specific types of input. While a fuzzer can theoretically be created for any input method a piece of software may support, the most dominant types of fuzzers are: network fuzzers, file fuzzers, environment variable fuzzers, command line fuzzers, web browser fuzzers and in-memory fuzzers.

Network fuzzers operate by communicating with the software under test via network protocols, and usually attempt to locate flaws in the way the software parses or decodes messages conforming to a

specific protocol. The success or failure of test cases can sometimes be difficult to gauge, as when the software crashes, no output will be received and no more input can be processed. For this reason, many tools incorporate a process monitor of some sort, and attempt to restart the process under test should it fail.

File fuzzers work by attempting to create files that will expose flaws when read by the software being tested. In these cases, many thousands of slightly different files may be created, and an automated method of instructing the software to process each file should be utilised.

Environment variable and command line fuzzers work similarly, in that they manipulate either parameters passed to the software on the command line or via environment variables.

Web browser fuzzers attempt to expose flaws in browsers. Fuzzing by manipulating HTML is one technique employed, but browsers are far more complicated, and web browser fuzzers may attempt to locate errors in CSS parsing, JavaScript engines, COM objects, ActiveX components or any other features supported by the browser [11].

In-memory fuzzers attempt to provide their input directly to functions or processing routines within a piece of software, thereby bypassing the usual file or network transports. This can significantly improve the speed of the tool, but these types of fuzzers are difficult to implement [11].

2.5.3 Fuzzing Tools and Frameworks

There are a vast number of fuzzing tools currently available, targeted at many different file formats, network protocols and other input types. There are also a number of frameworks available that can be used for building new fuzzers. While many commercial products have surfaced, only open source software has been used within this project.

Autodafe

Autodafe [27] is a fuzzing framework aimed at discovering buffer overflow vulnerabilities in software by using what the author describes as "fuzzing by weighting attacks with markers technique" [28]. Markers are defined to be any character or string field within the users control. If one of these markers is used as an argument to an unsafe function, the weight of the marker is increased. The fuzzing framework then tests higher weighted markers first, in the hope of locating buffer overflow more quickly. Autodafe consists of a tracing component, and fuzzing engine. The fuzzing engine utilises a partial block-based

description of the protocol, and each canonical element is considered a marker. The tracer then sends this list of markers to the tracer component, which are then monitored. Functions which are deemed unsafe by the tracer have breakpoints applied, and are monitored by the tracer. Markers utilised by functions deemed unsafe receive greater weight, and are then fuzzed first. Information on buffer overflows that are triggered is then collected and presented to the user of the framework [28]. Autodafe has not been updated since August 2006 however [27].

(L)ibrary (E)xploit API (lxapi)

The website describe lxapi as "a collection of python methods designed for bug testing and exploitation of local and remote vulnerabilities". It includes a fuzzer component, and a simple GUI with block handling routines [29]. Unfortunately it is currently unmaintained and has been since 2003 [23].

Peach Fuzzer

According to the Peach Fuzzer website, Peach is an intelligent fuzzer capable of performing both generation and mutation based fuzzing. It works by utilising "Peach Pit" files, which define the structure, type information and relationships of data field to be fuzzed. It was initially written in python, but later rewritten using the Microsoft .NET Framework, primary in C#. It is cross platform capable thanks to the Mono open source .NET runtime framework. Peach was originally created by Michael Eddington, but is now under active development by Deja vu Security. Peach also includes process monitoring functionality through the use of 'agents', that allow the fuzzer to determine whether a process has entered an unknown state or crashed due to a test case. This allows interesting test cases to be recorded for future analysis [30].

SPIKE

The SPIKE framework, created by Dave Aitel, pioneered the block based method for reducing the the number of test cases generated while still thoroughly exercising a program's data handling and validation routines. The block based technique allows for the abstracted construction of protocol layers [31]. SPIKE is written in C, and requires fair knowledge of C to use, making the barrier to entry slightly higher than with frameworks written in higher level languages. It is also only available for the Linux platform [23].

Sulley

Sulley is a fuzz testing framework, aiming for extensibility by providing various building blocks that may be utilised to create targeted fuzzers. Sulley is capable of generating a wide variety of data types (such as length fields, checksums, etc), and also includes the ability to monitor network traffic or basic process state. Based on this information, the framework can be told to react appropriately, for example to restart a process in the event that a test case causes a crash. Sulley also attempts to keep track of the sequence of test cases that caused a fault. The input supplied to the software being tested that generated the fault is recorded, allowing for easy reproduction of the observed results.

Sulley utilised the block based approach popularised by the SPIKE fuzzing framework detailed above. Blocks are used to generate individual requests, and can be tied together to form a fuzzing session. Sulley also includes a basic web interface to monitor the progress of fuzzing operations [31].

VoIPER

VoIPER is a toolkit aimed at allowing researchers to test VoIP devices and services for security vulnerabilities. It is written to be easy to use, extensible and automatic. It is built on top of the Sulley fuzzing framework, and able to run on Windows, Linux and OS X due to it being written in the Python programming language [32]. VoIPER includes a large number of pre-created SIP targeted block-based protocol definitions that the underlying Sulley framework utilises to perform fuzz testing. VoIPER includes components aimed at fuzz testing certain SIP message types, including the INVITE, ACK, CANCEL, NOTIFY, SUBSCRIBE and REGISTER messages. It also includes components for testing SIP request structures, and SDP.

In addition to the above, VoIPER, like Sulley, includes a process monitoring component which allows for crash detection and recording. This allows for the easy reproduction of test cases that resulted in interesting results, such as crashes or unexpected behaviour [32].

2.5.4 Block Based Protocol Fuzzing

Block based fuzzers allow for a model of a protocol to be created using components called blocks. Each of these blocks may be designated a type, reference name and some seed data. Blocks may be defined as static or variable, where variable blocks will have their contents substituted with data generated by the fuzzing engine. By combining a number of blocks of various types, some static and some flagged as

variables, a protocol can be modelled in such a way that it is likely that a large portion of generated test cases will pass validation checks. Another common feature of the block based frameworks is the ability to perform common operations, such as checksums or size calculations on specified blocks. This allows for the framework to correctly update, where applicable, special fields in accordance with generated data. By substituting in various inputs designed to trigger faults into designated variable fields while keeping validation related fields such as checksums and sizes correct in relation to the overall message, the probability of the message being interpreted and not simply rejected is increased [24].

2.6 Types of software vulnerabilities

2.6.1 Denial of Service

Denial of Service (DoS) vulnerabilities generally provide an attacker with the ability to prevent the software from performing its intended functions. For example, a successful DoS attack against a web server would prevent it from serving legitimate requests. Denial Of Service attacks can be executed in a number of different ways. One of the most common methods of executing a DoS attack is to attempt to exhaust a computational or network resource of the target. This is often achieved by simply sending more traffic to the target than its network links can handle, thereby saturating the links and preventing legitimate traffic from reaching the target. This technique can also be extended to the application layer by issuing massive amounts of requests for a given resource. Depending on the target, this could result in memory exhaustion, socket or file descriptor exhaustion, excessive CPU time use or possibly even crash the targeted application. Any of these would prevent legitimate requests from being serviced, resulting in a successful DoS attack.

Another, more advanced method of performing such an attack, is to make use of a known implementation or design vulnerability within the targeted system of application. If it is known that a certain type of corrupt or malformed message will cause the software to either crash or consume a large amount of a given resource, this can be exploited to perform a more intelligent DoS attack. Often these vulnerabilities require very little traffic to execute, but can cause prolonged outages. In the case of a crash, a DoS may be performed with only a single message.

Alternatively, if the target is known to take a long time to service a given type of request or utilise a large amount of processing time or memory to service such a request (ie: generate a report of some sort), repeatedly sending such a request could consume all available resources on the target. This is

an example of a DoS attack exploiting a design rather than implementation flaw.

2.6.2 Remote Code Execution

Unlike DoS attacks, Remote Code Execution (RCE) attacks allow an attacker some control of the target host of application. RCE vulnerabilities are the most sought after and useful when correctly exploited, as they often allow partial or (possibly via another local exploit) full control of a target host. These attacks are performed by taking advantage of a vulnerability within the targeted application that allows for the insertion and execution of arbitrary code. To do this, an attacker must have the means to insert code into the target processes memory space, as well as influence the execution path of the target process. Inserting the code is usually just performed by embedding executable (often referred to as "shell code") into a legitimate or passable request and sending it to the target. The message contents are often stored in memory for further processing. If the execution path of the target process can then be modified to include the uploaded code, then the attacker may take full control of the attacked process. This yields the privileges of the user under which the process is running. RCE's are far more difficult to exploit than DoS vulnerabilities as specialised knowledge of both assembler and the operating system under which the target process is running is required. Furthermore, protections such as CPU No Execute (NX) bits and Address Space Layout Randomization, along with compiler features like buffer overflow prevention make these types of vulnerabilities even more difficult to take advantage of.

Most RCE vulnerabilities are a result of improper input handling within the targeted process, and one of the most basic examples is the "buffer overflow" class of vulnerabilities. This occurs when the software reads input into a buffer that is too small, and the input spills over into adjacent memory within the processes memory space. By carefully crafting the input, an attacker can place both executable code into the buffer, and with knowledge of the effect of the overflow, often influence the execution path of the process in such away that the inserted code is executed.

Chapter 3

Proposed Analysis Framework

3.1 Design Considerations

To work towards a usable security analysis framework, the functional requirements of the framework first need to be defined. By listing the requirements that the framework should meet, an understanding of the constituent components required to ensure that the framework is able to produce useful results can be obtained. This will assist in the design of the framework and allow for the development of the framework in stages.

The prototype analysis framework should meet the requirements listed below:

Support authentication using IMS-AKA algorithm

IMS uses IMS-AKA to authenticate UEs. To ensure that the framework is capable of testing messages other than SIP REGISTER messages, support for IMS-AKA should be included. This will allow testing of REGISTER messages, as well as other messages which normally are only available to authenticated UEs.

Test a range of SIP messages

IMS SIP defines a fairly large number of SIP message types, and the number of these different message types supported by the framework will influence its ability to expose flaws. The framework should

support testing various types of SIP messages, including REGISTER, INVITE, NOTIFY and SUBSCRIBE messages. Adding support for new message types should also be made trivial. Reuse of existing components should be possible.

Vary parameters within each message while attempting to maintain a valid SIP message structure

SIP messages contain a number of header and value pairs. The framework should be able to manipulate these header names as well as their values in a way that is likely to expose implementation flaws within the IMS platform under test. Delimiters between header names and values should also be varied.

Generate test cases by substituting in common boundary conditions and malicious strings

Instead of generating random data and placing it into various fields within the SIP message, mutations of values known to be likely to expose common implementation flaws should be performed and used to create malformed messages.This should improve speed, coverage and the quality of results.

Save test cases that caused the IMS implementation under test to behave in an unexpected way

To allow for easy reproduction of failures for further investigation after or during testing, test cases that trigger unexpected behaviour should be logged. The message or messages that trigger a failure of any kind should also be stored in a way that makes it easy to replicate the behaviour that was triggered by the test.

Allow for easy reproduction of test cases that produced unexpected results

To utilise stored test cases, a mechanism should exist to re-execute specific tests or re-use specific messages that triggered behaviour that should be further analysed.

Provide an indication to the user of the progress of testing

Fuzz testing can be performed almost indefinitely if random data is being used. Since the framework should utilise a finite number of mutations of "seed" data, the number of possible test cases would be known and finite too. Progress can and should therefore be exposed to the end user to provide an indication of how long testing will continue for and how quickly tests are being executed.

Monitor the components of the implementation under test to determine whether the component failed and restart if necessary

Manual intervention in testing would slow down the process and could possibly result in inaccurate results. When failures or crashes occur in the monitored IMS components, they will need to be restarted. The framework should monitor the component under test and determine whether it has failed. In cases where the component appears to have crashed, it should be restarted and the event should be logged.

3.2 Evaluation of fuzz testing techniques and frameworks

"Dumb" fuzzers generate random data and feed this randomly generated data as input into the software under test. They have no built-in intelligence about the format or structure of the data that the program under test expects. While this means they can be rapidly developed and placed into use, they can be highly inefficient when dealing with more complex or stateful protocols. By generating only random data with no regard for the protocol itself, most test cases will likely be rejected, and code coverage will be vastly reduced. Applying this technique to the IMS SIP protocol would be of limited benefit, as most, if not all, test cases would simply be disregarded by sanity checks within the implementation. Even if message parsing were attempted, coverage would be limited as the fuzzer would perform no authentication to IMS components.

Programs which perform input validation or maintain some form of internal state may not accept or process random input at all. The software may require that messages conform to a valid format, or expect certain fields to be present in messages provided as input. Furthermore, some messages may be accepted based on the internal state of the software, which may vary based on previous inputs. In these cases, "dumb" fuzzers are not effective, as most, if not all, of their generated test cases are simply discarded as invalid by the software under test. Smart fuzzers are far more effective in such cases. Smart fuzzers are programmed in such a way that they are aware of the input format that the software expects, and, in the case of stateful protocols, are able to traverse various states of the protocol by generating test cases consisting of multiple messages. A basic example is a protocol or message format that has its fields followed by a length. In these cases, random input would simply be discarded, as the format would be invalid. A smart fuzzer may however generate a valid (possibly random) field value, and both valid and invalid lengths. If bounds checking is not performed on the length field, the

software may read past the end of the memory allocated for the given field, and trigger unexpected behaviour.

In the case of the IMS SIP protocol, a "dumb" fuzzer would be unlikely to provide any reasonable code coverage, as the SIP protocol defines a very specific format that needs to be adhered to when creating messages. The IMS components themselves also require authentication, and maintain internal state for some types of messages, further reducing the effectiveness of a "dumb" fuzzer. A fuzzer capable of providing test cases that are at least partially valid or conform to the general structure defined by the SIP standards is therefore required. A smart fuzzer is therefore necessary to generate test cases that will provide reasonable code coverage for the SIP and IMS SIP protocols. As authentication is also required, this must be built into the smart fuzzer too.

Mutation based fuzzing utilises existing input samples, and mutates them in random ways to produce malformed input to the software under test. A dumb mutation based fuzzer may merely take valid input samples and randomly mutate parts of it to generate test cases. This technique allows for the creation of a fairly simple fuzzer, able to produce test cases that are more likely to be accepted by the software under test than random data. Further intelligence can also be built into mutation based fuzzers to ensure that only selected parts of the input samples are mutated in such a way that the overall structure of the message is still close enough to valid that the software under test attempts to parse and process the message. Messages may also contain fields that need to be updated based on the content that was modified, such as checksum or length fields. In these cases this intelligence would also need to be incorporated into the fuzzer to ensure that messages are still accepted [33].

Generation based fuzzers generate input from scratch instead of utilising existing input samples. To do this, the fuzzer needs to have knowledge of both the protocol used to interface to the software being tested, as well as the intelligence necessary to create new inputs. Technically, dumb fuzzers generating completely random data would also fall into this category, but their usage with more complex protocols is limited. One technique often used to create generation based fuzzers is to split the protocol or message format up into chunks or "blocks". These can then be assembled in a valid sequence and mutated or randomized independently. Further intelligence to create length and checksum fields may also be required, and would need to be built into the fuzzer and used to ensure that generated blocks are accepted as input. The granularity of these chunks and intelligence behind their construction define the intelligence of the fuzzer itself. Generation based fuzzers are usually able to move deeper into a protocol by constructing structurally valid messages [33]. Furthermore, such fuzzers may also be able to act reasonably well as clients or servers, reacting to messages received and responding as necessary

to transition through various states within the software being tested. This is where generational fuzzers would excel when applied to the SIP protocol, because the SIP or IMS components require both an authentication handshake to be completed, as well as responses to be generated in some cases.

Peach is a smart fuzzer capable of performing both mutational and generation based fuzzing. It utilises peach "pit" files that define the structure and type of information used for fuzzing. It was originally developed in Python, and was recently rewritten in Microsoft's C# .NET. It achieves cross platform compatibility with the mono project, which allows .NET software to run on other operating systems. While peach has many features, it also comes with a fairly steep learning curve. It aims to provide all necessary features and allow them to be orchestrated through the use of the "pit" files. These are XML files containing all the required information to fuzz a given piece of software. Peach also includes various process monitoring features to allow for crash detection.

Sulley is a fuzzing framework developed in the Python programming language. Instead of aiming to be a full-featured application supporting a Domain Specific Language (DSL), it is rather a framework designed to be extensible and allow for the easy creation of fuzzers for new protocols, and variations on existing protocols. The framework uses the block-based approach popularised by the SPIKE framework detailed in Chapter 2. These blocks can be defined and re-used easily when building new fuzzers on top of the framework. This exensibility, combined with the block based approach makes Sulley appealing for use in IMS SIP implementation testing.

The VoIPER framework builds on Sulley to provide a SIP specific fuzzing tool, capable of leveraging Sulley fuzzing engine while allowing target software to be monitored for crashes. Furthermore, VoIPER is partially state-aware for some SIP message types, and provides an excellent base on top of which to build an IMS SIP specific fuzzing platform. VoIPER supports fuzzing a number of different message types, and is easily extended to allow for new types of messages to be supported. While it supports MD5 based authentication, it has no support for the IMS-AKA authentication protocol, and therefore would not be able to register against an IMS implementation using the AKA protocol.

3.3 Proposed IMS-SIP testing framework

In the previous section, two main fuzzing techniques, as well as some prominent and relevent frameworks and utilities were discussed. By noting that SIP servers generally maintain an internal state, it was determined that "dumb" fuzzing would not be suitable for testing SIP implementations, and a smart fuzzer would be required. Generation and mutation based fuzzing were both evaluated, and

both appeared suitable for use against the SIP protocol when combined with the intelligence of a smart fuzzer. A hybrid approach is therefore employed by VoIPER and the proposed framework.

While Peach is a capable fuzzing suite, Sulley provides an easier to modify framework that can be use to build application or protocol specific fuzzers. It was also noted that VoIPER builds on the underlying Sulley framework to fuzz the SIP protocol, but does not support the IMS variation of the SIP protocol or IMS-AKA authentication.

3.3.1 Fuzzers

To provide some level or comparison between the performance of "dumb" and smart fuzzers, the proposed framework implements a small number of "dumb" fuzzers for simple SIP requests. These are then sent repeatedly without observing and responding to messages received from the copmonent under test. Test cases are simply generated and sent in sequence, and the responses consumed and disregarded. In the proposed framework, the "dumb" fuzzers still have the option of performing AKA authentication against the IMS component under test to ensure that at least some of the requests will be accepted, instead of merely dropped as unauthenticated. "Dumb" fuzzers are implemented for the following SIP requests:

1. SIP ACK
2. SIP CANCEL
3. SIP REGISTER

The REGISTER "dumb" fuzzer differs slightly in that it doesn't attempt any authentication, as this is the purpose of the REGISTER request itself.

The proposed framework developed for this project implements a combination of generation and mutation based fuzzing. This is achieved by leveraging the block based approach for data representation that Sulley uses. Using pre-created "requests" consisting of a number of blocks, some of which are marked as "fuzzable", the framework is able to move further through the SIP state machine maintained within the component under test. Smart fuzzers are implemented for the following SIP message types:

1. SIP INVITE
2. SIP SUBSCRIBE

3. SIP NOTIFY

4. SIP ACK

5. SIP OPTIONS

To allow the above message types to be tested, authentication is first performed against the IMS component under test. This is achieved by sending valid SIP REGISTER messages to the component, and completing the IMS-AKA procedure. The authentication process is repeated at a set interval to ensure that the registration timeout of the component being tested is not reached. This process allows the fuzzer to be viewed as an authenticated client, allowing the SIP messages above to be processed instead of merely dropped.

To ensure that the testing framework is able to detect failures within the IMS components under test, the associated process is monitored for crashes. Process monitoring is achieved by utilising a separate component that takes care of actually running the component under test, and notifying the fuzzer if it appears to have failed or crashed in any way. If a crash is observed, the test case that caused it is noted. A log of the message or messages that caused the crash are stored, as well as basic information about the crash itself. This allows for single interesting cases to be repeated, for both validation and further research into the underlying root cause of the crash.

3.3.2 Data Representation

The proposed framework utilises the block based approach of representing data to be fuzzed, as pioneered by Dave Aitel in [24]. This has proved both simple to use, as well as flexible enough to implement fairly complex protocols by building them up from smaller pieces and combining these pieces.

Sulley introduces the concept of individual "requests", which are then connected together to form a session, as described below. To create a request, it must be initialised with a name:

```
s_initialize("request name")
```

Each request is made up of a number of primitives and blocks. Primitives may be used directly within requests, or to create reusable "blocks", which can then be utilised within requests by referring back to them. Primitives are used to represent various data types used by a protocol. Sulley supports, among others, the following types of primitives:

Static and Random Primitives

The simplest primitive available in the Sulley framework is merely a static unchanging value that can be of arbitrary length. As with requests, they can be named to refer back to them later. Static primitives are created using any of:

```
s_static("example\x00data\x01")
s_raw("example\x00data\x01")
s_dunno("example\x00data\x01")
s_unknown("example\x00data\x01")
```

All of the above represent the same data, as s_raw, s_dunno and s_unknown are aliases for the s_static primitive. There also exists a binary primitive that allows for the representation of binary data. As SIP is mostly text based, this was not heavily utilised in the propsed framework.

Random primitives allow a number of options to be set in addition to their name. These are listed below.

1. min_length: The minimum length of random data to be generated. Mandatory for random primitives.
2. max_length: The maximum length of random data to be generated. Mandatory for random primitives.
3. num_mutations: The number of mutations to create and test before reverting to the default value, optional.
4. fuzzable: Enable or disable fuzzing of this primitive, optional.

A field of fixed size is created by setting the minimum and maximum length to be equal.

Integers

While static and random primitives can be used to perform very basic fuzzing of fields, it is preferable for the fuzzing framework to know what type of data the field usually contains so values can be generated more intelligently. As many protocols have numbers or integer fields, Sulley provides a way to represent these with integer primitives. Since these fields can be of varying length, the framework provides the following primitives:

- 1 byte: s_byte, s_char
- 2 bytes: s_word, s_short
- 4 bytess: s_dword, s_long, s_int
- 8 bytes: s_qword, s_double

Integer primitives require at least a default value to be provided, but also provide the following optional paramaters:

1. endian: Endianness of bit field, ie: little or big endian
2. format: Binary or ASCII
3. signed: Field is signed or unsigned
4. full_range: Mutate through all values instead of attempting to choose just boundary values
5. fuzzable: Enable or disable fuzzing of this primitive
6. name: Set a name for the field

The full_range option limits the framework to trying only values that would likely expose errors. These include values near boundary values, as well as the maximum field value divided by 1, 2, 3, 4, 8, 16 and 32. If the full range of values were to be tested, the number of test cases generated would take far too long to be useful in most cases. For example, testing all possible values for a DWORD (4 bytes) would results in roughly 4 billion test cases. If each test takes only a fraction of a second, this would still consume a significant amount of time. Values far away from boundary values are also less likely to expose errors.

Strings and Delimiters

String primitives allow test cases to be generated around default values provided for use with the string primitive. They require at least a default, valid to be specified. Additionally, the following paramaters are exposed:

1. size: Static size for the string, if not provided dynamic sizing is used
2. padding: A character to pad the string with if size is specified and the generated string is smaller than the specified size

3. encoding: Defaults to ASCII, but others like utf_16_le may be used instead

4. fuzzable: Enable or disable fuzzing for this primitive

5. name: As with other primitives, a name may be specified

As many protocols make use of a delimiter to separate fields within messages, a delimiter primitive is also provided by the framework. Delimiters also require a default value, and may have a name specified. They may also be designated as fuzzable or not.

Blocks

Primitives can be organised and nested within "blocks". Each block must be given a name, opened and closed. Each block may further be associated with an encoder and/or a group. Blocks may also have dependencies specified.

Grouping allows a block to be tied to a group primitive, where the block will be cycled through all possible mutations for each member value of the group primitive to which it is tied.

Encoders are used to modify the rendered contents of a block (passed to the encoder as a string) into some format that is acceptable by the software under test. For example, if the protocol calls for all messages to be XOR'd with a static value, an encoder could be provided to perform this conversion on each value rendered for the block before sending the test case to the software under test.

Dependencies allow for conditional rendering of a given block. A block takes the name of a primitive, a dependent value and optionally a new comparison function.

Blocks with dependencies specified will only be rendered and output when the the primitive on which they depend is set to a specified value, and the comparison function (by default equality) returns true based on the dependent values and value of the associated primitive.

Block Helpers

Block helpers allow for more accurate representation of valid messages by providing "helpers" able to generate size and checksum fields that may be used to ensure that messages appear valid.

The sizer helper allows the framework to dynamically calculate the length of the associated block when rendering the helper. These fields are usually not fuzzed as sanity checks within the software being

tested would likely just discard a message should a size field be invalid. They can be set to fuzzable if required though.

The checksum helper is similar to a sizer, but instead of calculating the length of a block, generates a checksum based on the specified algorithm (for example, crc32, adler32, md5 or sha1).

Repeater helpers are used to simply replicate a given block a number of times. This can be helpful for testing whether an implementation can handle very large numbers of repeatable fields. Repeaters are helpful in discovering overflow vulnerabilities.

Legos

Legos represent user-defined re-usable components within the Sulley framework. Examples of legos may be e-mail addresses, hostnames or binary protocol primitives. Legos are called by name and with a value. The lego may then add delimiters, pad or encode the provided value as necessary to form a useful component of the protocol being fuzzed. Adding "legos" to Sulley significantly enhances the functionality of the framework by allowing it to provide reusable components for new protocols.

Legos are essentially user defined classes that extend the Sulley blocks.block class to add functionality, that may be reused within requests and blocks.

3.3.3 Sessions

The requests defined to interact with the IMS SIP components are tied together to form a "session" within the Sulley framework. Requests are linked together in a graph structure, and this graph is walked from the root node, while each request is fuzzed along the way.

Figure 3.1: Example Session structure [3]

Figure 3.1 shows a graphical representation of a Sulley session graph for a basic SMTP protocol fuzzer. Each node is a request. The requests are tied together as shown below to form the session.

```
sess = sessions.session()
sess.connect(s_get("helo"))
sess.connect(s_get("ehlo"))
sess.connect(s_get("helo"), s_get("mail from"))
sess.connect(s_get("ehlo"), s_get("mail from"))
sess.connect(s_get("mail from"), s_get("rcpt to"))
sess.connect(s_get("rcpt to"), s_get("data"))
```

Requests and sessions are created for the SIP protocol similarly to the above, with extra functions included to handle responses from the component being tested where necessary.

3.4 Building upon an existing framework

After evaluating and comparing a number of popular fuzzing utilities and frameworks, shown in the table 3.1, it was decided that it would be advantageous to modify an existing framework which included some of the required functionality required to target IMS SIP components.

Tool or Framework	Fuzzer Type	Features	Language	Extensibility	Data Representation
Autodafe	Capture-Replay	P, C	C	Minimal	Block based
lxapi	Generation, Mutation	-	Python	Minimal	Block Based
Peach Fuzzer	Generation, Mutation	P, C, T	C# .NET	Moderate	Peach Pit DSL
SPIKE	Generation, Mutation	C, T	C	Minimal	Block Based, C
Sulley	Generation, Mutation	P, C, T	Python	Easy	Block Based
VoIPER	Generation, Mutation	P, C, T, S	Python	Easy	Block Based

P - Process Monitoring, C - Crash Logging, T - Test Replay Support, S - Built in SIP support

Table 3.1: Tool and Framework Comparison

Instead of creating a new fuzzing utility from scratch, the VoIPER SIP fuzzer, built on top of the Sulley framework was instead chosen for use and modified as necessary. This decision was made due to the VoIPER tool being based on the already extensible Sulley framework, as well as its included library of SIP oriented fuzzers. The included SIP logic was also largely applicable to IMS SIP, and due to this large portions of the existing fuzzers were able to be reused in creating an IMS targetted fuzzing framework.

Component	Description	Reusable
Authentication Logic	Functions that assist in authenticating to a SIP server	No
Registration Logic	Registration related functions	Partly
SIP State Logic	Maintains basic internal state	Yes
Process Monitor	Monitors component under test for unexpected behaviour	Yes
Fuzzing Engine	Fuzzing functionality built on Sulley framework	Partly
Replay Mechanism	Utility to replay test saved test cases	Yes

Table 3.2: Component Reusability

Table 3.2 shows that VoIPER includes a useful amount of functionality that could be easily reused or modified for use against the SIP components of an IMS implementation. Each component was individually tested against IMS SIP components. Additions and modifications were then made as necessary to allow the components to work with IMS implementations.

3.4.1 New Functionality

VoIPER only includes basic support for Digest authentication. As IMS-AKA was not supported by VoIPER, this funtionality was implemented and added to the proposed framework. This allows it to authenticate against IMS SIP components. Only IMS-AKAv1 authentication is used within the proposed framework. Implementation details are provided in chapter 4.

3.4.2 Modifications

The proposed framework is required to register with IMS SIP components to allow further messages to be accepted instead of being dropped as unauthenticated. To do this, the framework must complete the IMS-AKA registration process. While VoIPER does include basic registration functionality, the proposed framework modifies this behaviour to allow for correct interaction with the IMS SIP components. This was necessary, as the existing implementation was unable to successfully register and have the IMS recognise it as an authenticated UE.

Some components of the fuzzing engine also required modifications, as in some cases the IMS components expected headers that were not present in the original VoIPER framework. The proposed framework adds these expected headers and also creates test cases around them. This was achieved by modifying the "requests", "blocks" and "legos" utilised by the various SIP message fuzzers.

Details of the modifications are provided in chapter 4.

3.4.3 Reused Functionality

VoIPER included functions which handled SIP responses from the IMS SIP components, and responded to them when necessary. These functions allowed the fuzzing engine to maintain a basic internal state, and were reusable with very few modifications.

The process monitor included with VoIPER and later Sulley proved generic enough to run IMS SIP components with no modification to the software itself. The process monitor uses a custom RPC protocol to communicate with the fuzzing engine, allowing it to inform the fuzzing engine of crashes or unexpected behaviour.

The replay mechanism included with VoIPER was also usable without modification. This was due to it simply reading out saved test cases from a text file and sending them to the listening socket of the IMS component under test.

Details of reused functionality are provided in chapter 4.

Chapter 4

Framework Implementation

In chapter 3, an overview of the Sulley and VoIPER frameworks was provided, as well as details on how the proposed framework targeted at IMS SIP components would build on these. This chapter details the objectives, requirements and limitations of the proposed framework. Implementation details of the framework are then provided. Details of software and hardware used for testing are also provided.

4.1 Framework Objectives

The key objectives that the vulnerability assessment framework should achieve are:

- Identify and locate implementation specific vulnerabilities and errors within IMS SIP components
- Facilitate further analysis of discovered vulnerabilities or errors
- Demonstrate that fuzzing is effective in assessing security vulnerabilities in IMS implementations
- Provide reusable functionality that may be extended to support other SIP messages or extensions

4.2 Framework Requirements

To be useful in locating and exposing implementation specific vulnerabilities within IMS SIP components, the framework should meet a number of functional requirements. These are listed below:

- The framework must be able to test at least the P-CSCF, S-CSCF and I-CSCF SIP implementations.
- Authentication must be performed to ensure messages aren't dropped before parsing is attempted
- Test cases resulting in unexpected behaviour should be logged
- The framework must allow for test cases that caused unexpected behaviour to be replayed

4.3 Limitations

The implemented framework was tested exclusively with the Open IMS Core. Testing the implemented framework on implementations other than the Open IMS Core is left as future work. Furthermore, only a subset of all SIP message types used within the IMS are implemented by the framework. While the most common message types are tested, there may be many extensions that aren't tested.

Fuzz testing itself is also essentially a method of brute force vulnerability discovery, and in some cases no flaws may be discovered. This may be due to either an excellent implementation with flawless error checking, or due to test cases that trigger existing errors simply not being generated. While the functionality of the underlying Sulley framework aims specifically to create test cases that would trigger unexpected behaviour in general, there is no guarantee that any flaws will be discovered. The fuzzing framework is only aimed at locating implementation specific vulnerabilities, and existing attacks such as those outlined in section 2.2.1 are not dealt with.

4.4 Framework Architecture

The assessment framework consists of a number of components that are used together to perform fuzzing of the IMS SIP components. The fuzzing engine itself is a modified version of the VoIPER toolkit, built on top of the Sulley fuzzing framework. In addition to the fuzzing engine itself, a number of other components are present which provide additional functionality. These are the process monitor and replay utilities, which are also detailed below.

4.5 Fuzzing Engine

The fuzzing engine coordinates and runs the selected fuzzer module for each type of SIP message supported. The fuzzer is selected via command line arguments. In addition to selecting the fuzzer, registration may be enabled or disabled, along with process monitoring via command line options.

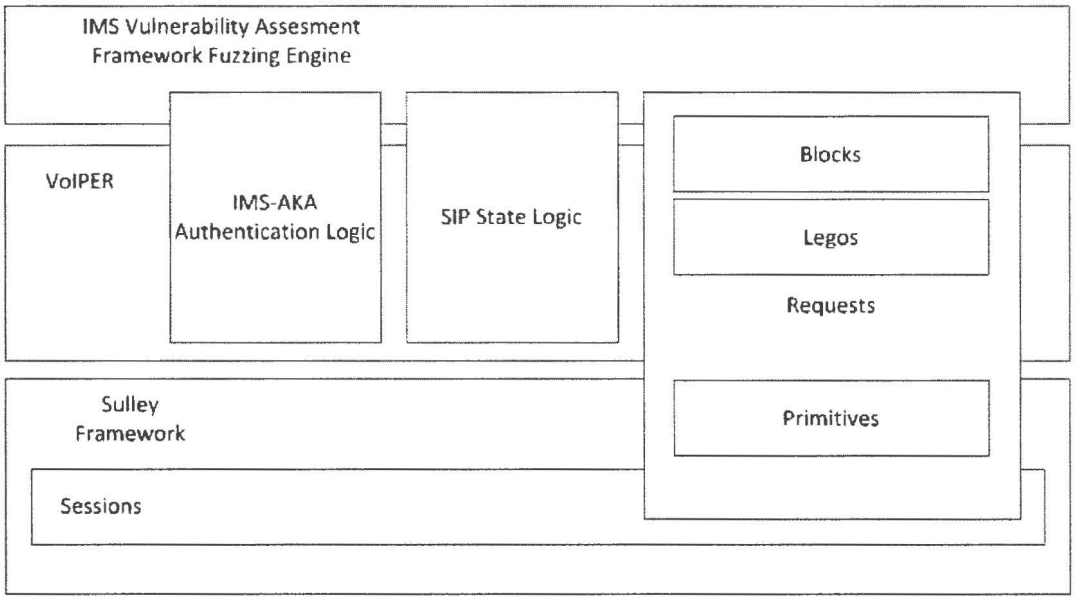

Figure 4.1: Fuzzing Engine Components

Figure 4.1 shows the logical structure of the developed prototype. The Sulley framework provides the base fuzzing capability in the form of blocks and primitives. Sulley also performs basic progress tracking of mutations through the concept of sessions. The sessions layer allows for resuming of fuzzing runs by creating a 'session' file with information about the state and progress of the currently running fuzzer. Blocks and primitives are utilised by the modified VoIPER framework, which also defines a number of legos that were modified for use with IMS SIP components. Requests are built by combining the blocks, legos and primitives in various ways to generate test cases which attempt to expose possible implementation flaws within the targeted components. The IMS-AKA authentication logic was added to the VoIPER framework, and the SIP state logic modified where necessary to interact with the IMS.

4.5.1 Authentication logic

The assessment framework implements the AKA algorithm, and authenticates when necessary using SIP digest access authentication with the AKA algorithm. A username is specified within the frame-

work configuration file, and this is used to generate both an IMPU and IMPI for authentication against the IMS SIP component being tested.

Initially a SIP REGISTER is sent to the component under test. The structure of the initial REGISTER message is shown below.

```
REGISTER sip:open-ims.test SIP/2.0
To: <sip:alice@open-ims.test>
From: <sip:alice@open-ims.test>;tag=34gvosiw5kp9alm6fhn8u2eqtdb7zxjy
Call-ID: register.34gvosiw5kp9alm6fhn8u2eqtdb7zxjy
CSeq: 59477 REGISTER
Via: SIP/2.0/UDP 192.168.0.22:50000;branch=
    ↪ z9hG4bKk34gvosiw5kp9alm6fhn8u2eqtdb7zxjy
Max-Forwards: 70
Allow: INVITE,ACK,OPTIONS,BYE,CANCEL,NOTIFY,REFER,MESSAGE
Contact: <sip:alice@192.168.0.22:50000>
Expires: 3600
```

If a SIP 401 "Unauthorized" response is received, the framework attempts to register by sending a new REGISTER request with the correct response to the received challenge. The received challenge includes a "nonce" value which the framework uses to deduce the RAND and AUTN values. These values are then used by the framework, along with the configured key in the framework configuration file to calculate the CK and IK keys, as well as generate a RES value which is then returned to the component requesting further authentication. The generation of the CK, IK and RES values is performed using an external Python MILENAGE library which implements 3GPP MILENAGE alorithms f2, f3, f4 and f5.

The second registration request now includes the response value within the Authorization header. An example of this request's structure is provided below.

```
REGISTER sip:open-ims.test SIP/2.0
CSeq: 59478 REGISTER
Via: SIP/2.0/UDP 192.168.0.22:50000;branch=
    ↪ z9hG4bKk34gvosiw5kp9alm6fhn8u2eqtdb7zxjy
To: <sip:alice@open-ims.test>
From: <sip:alice@open-ims.test>;tag=34gvosiw5kp9alm6fhn8u2eqtdb7zxjy
```

```
Call-ID: register.34gvosiw5kp9alm6fhn8u2eqtdb7zxjy
Authorization: Digest username="alice@open-ims.test", realm="open-ims.
    ↪ test", nonce="MDZgxK08lNHz1b1O6znWxxW/7l1AzQAAda5BCgFXtEQ=", uri="
    ↪ sip:open-ims.test", algorithm=AKAv1-MD5, response="
    ↪ a906ea311bb619f20b3a0fdf904300d7"
Allow: INVITE,ACK,OPTIONS,BYE,CANCEL,NOTIFY,REFER,MESSAGE
Contact: <sip:alice@192.168.0.22:50000>
Expires: 3600
Max-Forwards: 70
```

Once a successful response is received in the form of a SIP 200 "OK", fuzzing begins for the selected SIP message type.

The username and password used to generate the authentication response, as well as the authentication algorithm to use are specified in the configuration file used by the framework. Appendix C details the format of this file and the available options.

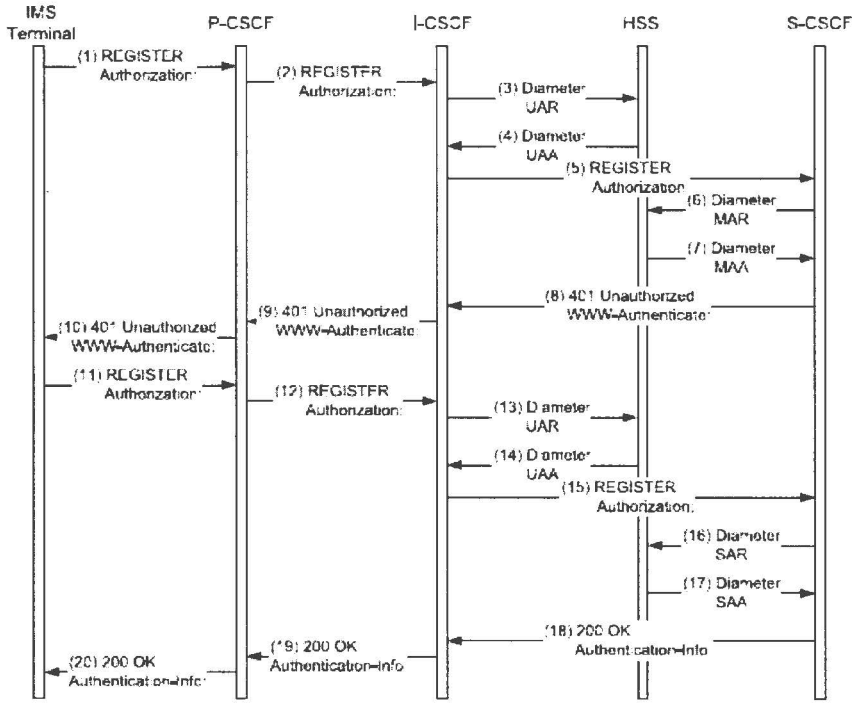

Figure 4.2: HTTP Digest Access Authentication in IMS [4]

Figure 4.2 depicts the registration procedure followed by the framework. In the above case, the

component under test would be the P-CSCF, and this would be monitored by the framework for failures. To ensure that the framework always appears as a registered user of the IMS implementation, re-registration is attempted every 50 requests (or after 50 test cases have been sent). If re-registration is not required, the framework simply continues sending test cases.

4.5.2 Requests

Requests are used by the included fuzzers to combine primitives, blocks and legos into messages that can be manipulated to generate test cases. The requests were part of the original framework, but required modification in order to ensure compatibility with IMS SIP components. Implemented SIP related requests are listed below

INVITE_STRUCTURE, INVITE_REQUEST_LINE, INVITE_OTHER, INVITE_COMMON

A number of requests are created around the SIP INVITE message in order to focus test cases on various aspects of the INVITE message. As it is one of the most commonly used and option-rich messages, more focus was placed on this message.

The INVITE_STRUCTURE request attempts to create INVITE messages with non-conformant structures. This is achieved by repeating the "INVITE sip:TARGET@HOST SIP/2.0" line, where "TARGET" and "HOST" are replaced with configured values. The "Via" header, the "Call-ID" and "To", "Contact" header values are also repeated. An SDP block is also created and repeated.

The INVITE_REQUEST_LINE request mutates only the line of the SIP message containing the INVITE keyword.

The INVITE_OTHER request attempts to introduce non-standard headers into the message. This creates a message that is for the most part incompatible with IMS implementations, and was not fully evaluated.

The INVITE_COMMON request attempts to mutate and introduce malicious values into fields that are commonly found in SIP INVITE messages. Modifications were made to this request to ensure compatibility with IMS implementations, as well as to add IMS specific headers to increase test-case coverage.

OPTIONS

Only a skeleton request exists for the SIP OPTIONS fuzzer, as the messages are generated in a Python class instead.

ACK

The ACK request contains a static SIP ACK line, while introducing fuzzed values into SIP URI fields. Field names and delimiters are also fuzzed.

CANCEL

The CANCEL request also contains a static SIP CANCEL line, and similarly to the ACK request, introduces fuzzed values into URI fields. Field names and delimiters are also fuzzed. This request was modified to include a Route header for IMS compatibility.

REGISTER

The REGISTER request introduces fuzzed values into the REGISTER line, as well as header values and SIP URIs. Legos are used for SIP URIs and Authorization headers. Strings, integers and delimiters are fuzzed within the REGISTER request.

SUBSCRIBE

As above, the SUBSCRIBE request fuzzes values within the SUBSCRIBE line itself, specifically the value of the CSeq header. Header names are left static, while their values and delimiters between field values are mutated. Legos are used to create values for IP and SIP URI fields.

NOTIFY

The NOTIFY request also introduces fuzzed data into the CSeq header. Header names remain static while values and delimiters have fuzzed data substitited in. Furthermore, the "Content-Type" header name is fuzzed, as well as the body of the NOTIFY message. Legos are once again used for SIP URIs and IP addresses.

4.5.3 Protocol Blocks

Blocks have been created for each header and request line used in the above listed requests. Where only a simple string or delimiter is required, primitives are used instead as there would be no benefit to creating a new nested block for these parts of a given message. An example of a simple block is provided below, which is utilised within the NOTIFY request:

```
if s_block_start("notify_request_line"):
        s_static("NOTIFY sip:USER@HOST SIP/2.0\r\n")
s_block_end()
```

The framework then substitutes the configured values into the placeholders denoted by USER and HOST in the block shown above.

4.5.4 State logic

Since the framework only aims to mimick a SIP UA well enough to have the IMS SIP components accept messages from it, only basic state tracking for SIP transactions is maintained. Dictionaries are used internally to keep track of sent transactions, and their expected responses. An initial message is sent, and when a response is received, the state dictionary within the fuzzer is used to determine what action to take based on the SIP response code. An example of one of these dictionaries is provided below.

```
self.cancel_transaction_dict = {sip_parser.r_1XX :
                (self.cancel.process,
                        {sip_parser.r_4XX : (self.ack.process, None),
                         sip_parser.r_5XX : (self.ack.process, None),
                         sip_parser.r_6XX : (self.ack.process, None),
                         sip_parser.r_2XX : (None, None),
                        }
                ),
                sip_parser.r_4XX : (self.ack.process, None),
        }
```

The above dictionary is taken from the SIP INVITE fuzzer class. This class initially sends an INVITE. The above dictionary is then used to deal with responses. In the above case, a SIP 1xx response is

expected, after which the cancel.process function method will be called to generate a SIP CANCEL message and this will be sent to the component under test. Thereafter a SIP 4xx, 5xx, 6xx response from the IMS SIP component will trigger the framework to generate and send a SIP ACK message back. Should a 2xx RESPONSE be received, the dialog is over. If a 4xx message is initially received instead of a SIP 1xx message, the dialog also ends.

Dictionaries are similarly used in all other fuzzers to generate appropriate responses to messages received from the IMS SIP component under test. The authentication logic utilises a dictionary base approach to handling the authentication process.

4.5.5 Legos

Legos are used where primitives or blocks are not flexible enough to achieve the desired output or output format. Legos are simply Python classes that extend the built in Sulley block class, and as such may be used similarly to blocks. Legos allow for further flexibility as options may be passed in from the request using the Lego, and these can be used in any way within the new Lego class.

Legos that appear within VoIPER and are utilised within the assessment framework are detailed below.

q_value Lego

This is used to create a q_value field value that is used within various requests. It is simply made up of a number of primitives, and takes a "fuzzable" option to turn fuzzing of its internal primitives on or off, all at once.

to_sip_uri, from_sip_uri, to_sip_uri_basic, from_sip_uri_basic Legos

These legos simply create outputs resembling SIP URIs, inserting placeholder keyword such as "USER", "LOCAL_IP" and "PORT" into the output, which is then replaced with configured values before being sent to the IMS SIP component under test.

4.5.6 Configuration File

The configuration file for the implemented framework allows the following options to be set:

- user: Username to use for registration.

- password: Password to use for registration
- target_user: User to direct INVITE requests toward (if another UA is registered with this user, it will receive the invite requests if they are interpreted by the IMS component being tested)
- do_cancel: If set, the framework will attempt to cancel INVITE requests after sending them
- do_register: This instructs the framework to perform registration before initiating fuzzing. Registration is also attempted every 50 requests thereafter to ensure the framework always appears as an authenticated UA.
- auth_type: Set the authentication algorithm (aka or md5)
- auth_digest_username: The username to use when performing SIP digest authentication using the AKA-v1 algorithm.
- local_ip: The source IP that the fuzzer should bind to when sending messages to the component under test.

The full configuration file used during testing appears in Appendix C.

4.6 Supporting Components

4.6.1 Process Monitor

The process monitor script runs an agent that listens to and reacts to commands it receives from the fuzzing engine component of the framework. The process monitor is started and binds to a TCP port awaiting commands. When the fuzzer is run with process monitoring enabled, the specified start command is sent to the process monitor, which then spawns and monitors the process. Once the process is running, the fuzzing engine begins sending test cases to the now monitored process, which in this case is an IMS SIP component. When a crash or exit is detected, basic information regarding the unexpected exit is logged, and the fuzzer is informed that the test case resulted in unexpected behaviour. The fuzzing engine then logs the test number and test case contents for playback later, allowing further analysis. The process monitor communicates with the framework using a custom RPC protocol.

4.6.2 Replay Utility

The replay utility allows for specific saved test cases to be replayed, re-sending the test case that caused the framework to log and store the test case. This utility simple reads the contents of a stored test case and sends it via a socket to the component under test. While simple, it is useful to verify that test cases cause repeatable errors and assist with further analysis of the observed failures.

4.7 Other Modifications to VoIPER and Sulley

During testing, it was noted that IMS components expect the same source port to be used for all SIP messages. Originally a socket was created for every test case, resulting in random source ports being used for messages sent to the component under test. Modifications were made to both the VoIPER and Sulley frameworks to allow for a single socket to be reused for all test cases and for authentication. This resolved problems with the IMS not responding to messages originating from different source ports.

The authentication state dictionary also required modification to allow the framework to properly respond to challenges issues by the component under test. Changes were made to allow the framework to respond correctly to these challenges, as well as to support the AKAv1 authentication algorithm.

Further modifications were also made to allow the framework to exit on Ctrl-C being pressed, instead of merely halting the thread that caught the KeyboardInterrupt exception.

4.8 Testbed Architecture

To verify that the framework was able to meet its identified objectives and fulfil the listed requirements, a testbed was created around the framework. The testbed consisted of the vulnerability assessment framework with its supporting components interfacing with the SIP components of a running instance of the Open Source IMS Core. Figure 4.3 shows the assessment framework testing the P-CSCF component. Both the I- and S-CSCF components were also tested using a similar setup.

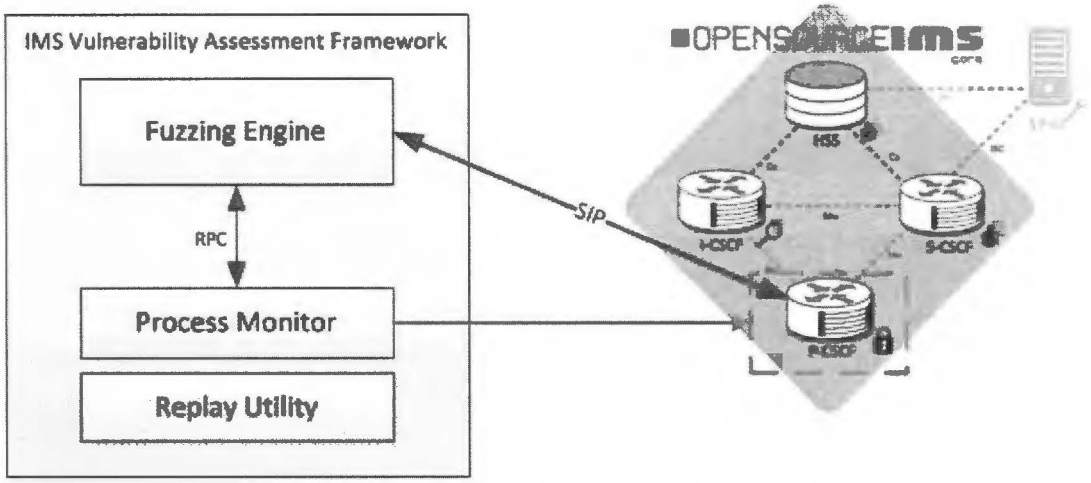

Figure 4.3: Testbed Architecture [5]

To gain more insight into the errors detected using the testbed, debugging was enabled for all IMS SIP components by ensuring that all running SIP components were passed the "-D -D" parameters. The testbed was built and run within a virtual machine running under Oracle's VirtualBox PC emulator. Details of the hardware and software configurations appear in Appendix A and B.

4.8.1 Fraunhofer FOKUS Open IMS Core

The Open IMS core provides open source implementations of various core IMS components. The core consists of three IMS CSCFs implemented on top of the SIP Express Router using the C language, as well as a Java implementation of a Home Subscriber Server (HSS) [5].

The CSCF (Call/Session Control Function) is an essential node within the IMS that processes SIP signalling.

UAs register with the core by interfacing with the P-CSCF. The P-CSCF acts as a SIP proxy server, meaning that all requests traverse the P-CSCF. The P-CSCF authenticates the UA, establishes security associations, and asserts the identity of the UA to the other nodes within the network. As the P-CSCF is trusted, other nodes do not require further authentication from the UA. The P-CSCF also verifies the correctness of SIP requests, and prevents SIP requests that don't follow the SIP protocol. The P-CSCF also generates charging information [4].

The I-CSCF is another SIP proxy located at the edge of an administrative domain. The I-CSCF has an interface to the HSS and uses information retrieved about the user to route SIP requests to the

appropriate destination (usually an S-CSCF). The I-CSCF may also interface with application servers (AS) to route requests toward services rather than users where required [4].

The S-CSCF forms the central node within the signalling plan, and performs both SIP server as well as sesion control functions. It also acts as a SIP registrar, meaning it maintains a binding between the UA location and SIP record. The S-CSCF has an interface to the HSS, which is used to retrieve authentication vectors which are in turn used to authenticate users. It also downloads the user profile and included triggers, which can indicate that certain SIP messages should be routed via one or more ASes. The S-CSCFs functions are to provide SIP routing services, as well as enforce the network operator's policies [4].

The Home Subscriber Server (HSS) forms the central repository for all user-related information. It is a evolved version of the HLR (Home Location Register) found within GSM networks [4].

4.8.2 Component Testing

The proposed framework will test all SIP capable components of a given IMS implementation. These are the P-, S-, and I-CSCF. Each SIP component will be tested and monitored separately, while the others will run as normal with the remainder of the IMS framework.

Chapter 5

Evaluation Results and Analysis

Using the framework described in Chapter 4 within the testbed detailed, each SIP component of the Open IMS Core was tested using various fuzzers within the framework. This chapter collates and analyses these results. Each fuzzer was run against each IMS SIP component using the testbed setup.

5.1 Testing Procedure

5.1.1 Overview

The testbed detailed in Chapter 4 was utilised to perform fuzz testing on each IMS SIP component. The remaining components were run as normal, without any process monitoring. The component under test was executed by passing the appropriate command line arguments to the fuzzer. This then informs the process monitor to execute and monitor the SIP component to be tested. Information about any failures observed are then communicated back to the fuzzer using a custom RPC protocol. When a crash is encountered, the fuzzer is informed, the test case number and contents are logged, and the process is restarted if it crashed. In some cases, if the process hangs or fails to respond, manual intervention may be required. As the fuzzer utilises a state or "session" file, re-running the fuzzer again with the same session file allows testing to resume from the last test case instead of re-running all test cases. As there are usually 10 000 to 100 000 test cases, the ability to resume fuzzing can result in significant time savings.

5.1.2 Data Collection

Any test cases causing failures in the component under test were stored within a directory whose name corresponded to the fuzzer being run, underneath a directory named according to the component being tested. The number of failures for each component and fuzzer were noted and tabulated. Fuzzers were classed as "smart" or "dumb" to allow for performance comparisons.

To reduce the time needed to perform a run of each fuzzer against each SIP component, a bash script was created that could be run by simply giving the name of the desired fuzzer to be run, as well as the name of the component against which the tests should be run. The script appears in Appendix E.

5.1.3 Testing Metrics

To gauge the performance of the tested fuzzers against the IMS SIP components, the following metrics were identified and used to form a basis for comparison:

1. Fuzzer Type ("smart" or "dumb")
2. Number of test cases that caused detectable failures

Fuzzers that uncovered significantly more test cases that resulted in unexpected behaviour were deemed to perform better than those which generated little or no test cases that triggered unexpected behaviour in the component under test. The number of test cases generated by each fuzzer is also noted.

5.2 Results

After running each fuzzer against each SIP component within the IMS testbed, the number of crashes triggered in each component by each fuzzer was recorded. Figure 5.1 provides an overview of the number of test cases that triggered errors in each component.

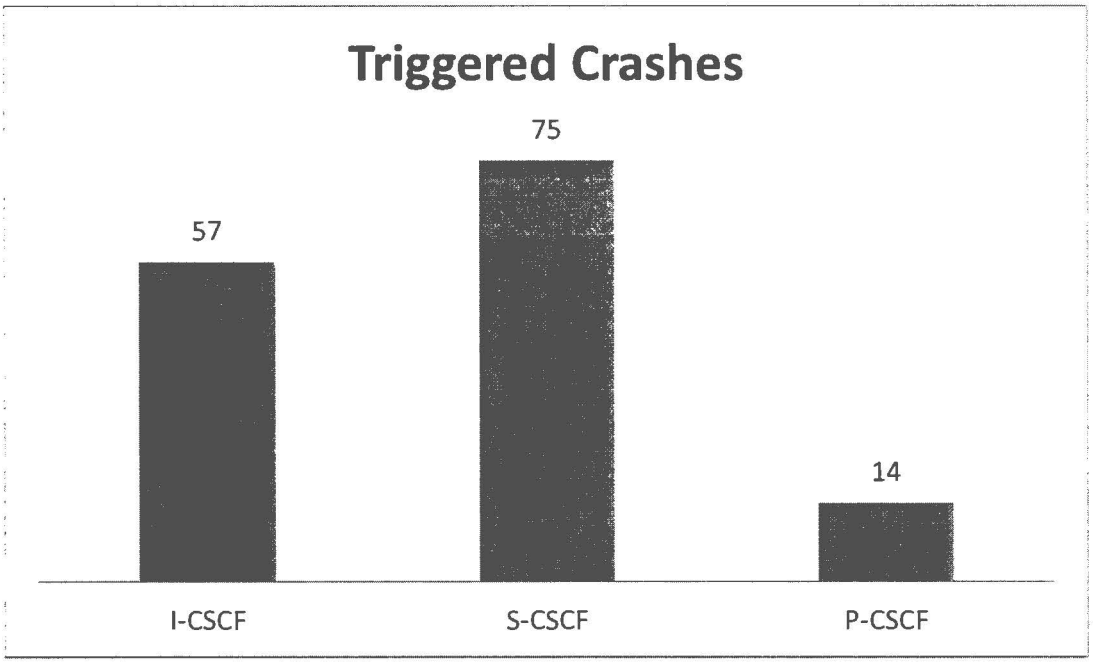

Figure 5.1: Number of test cases found to cause crashes per component

Figure 5.1 allows for easy identification of the component that appeared to be most susceptible to failing when tested by the implemented framework. In this case it was the S-CSCF, but the ability of the framework to effectively detect vulnerabilities and expose implementation errors is shown by the fact that test cases were discovered that caused failures in all tested components.

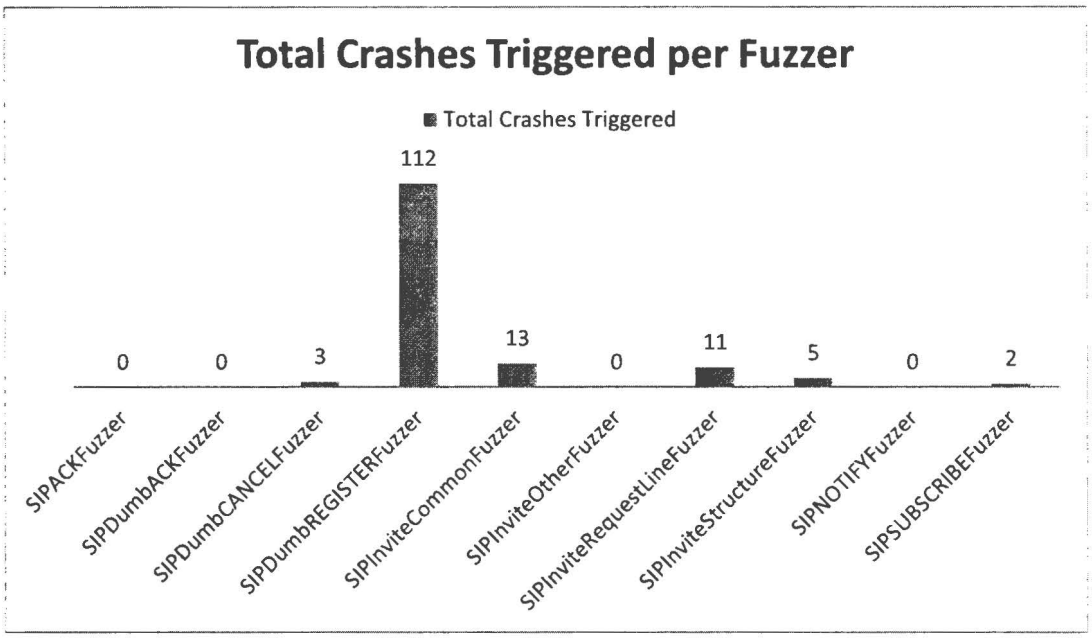

Figure 5.2: Number of test cases found to trigger crashes per fuzzer

5.2.1 I-CSCF

Table 5.1 shows the results obtained when running each fuzzer against the I-CSCF component of the Open IMS Core.

Fuzzer	Triggered Crashes
SIPACKFuzzer	0
SIPDumbACKFuzzer	0
SIPDumbCANCELFuzzer	2
SIPDumbREGISTERFuzzer	53
SIPInviteCommonFuzzer	0
SIPInviteOtherFuzzer	0
SIPInviteRequestLineFuzzer	0
SIPInviteStructureFuzzer	2
SIPNOTIFYFuzzer	0
SIPSUBSCRIBEFuzzer	0

Table 5.1: I-CSCF Results

A total of 57 failures were triggered by test cases generated by the fuzzers. While the majority appeared to be a result of the "dumb" REGISTER fuzzer, it should be noted that, unlike the other fuzzers, the component under test would be obliged to attempt to parse the REGISTER messages, even if they contained invalid credentials. A further 2 crashes were triggered by the "dumb" CANCEL and SIP INVITE structure fuzzers.

As the I-CSCF is placed on the edge of administrative domains, it may provide a method for malicious third parties to interact with it. For example, if a legitimate customer with access to the I-CSCF or a trusted operator's network is compromised, an attacker may have the opportunity to send arbitrary traffic to the I-CSCF.

5.2.2 S-CSCF

Table 5.2 shows the results of running each fuzzer against the S-CSCF IMS SIP component. The fuzzing framework produced the largest number of test cases that triggered crashes for this component.

Fuzzer	Triggered Crashes
SIPACKFuzzer	0
SIPDumbACKFuzzer	0
SIPDumbCANCELFuzzer	1
SIPDumbREGISTERFuzzer	46
SIPInviteCommonFuzzer	12
SIPInviteOtherFuzzer	0
SIPInviteRequestLineFuzzer	11
SIPInviteStructureFuzzer	3
SIPNOTIFYFuzzer	0
SIPSUBSCRIBEFuzzer	2

Table 5.2: S-CSCF Results

The S-CSCF had a total of 75 crashes triggered by the fuzzers run against it. This demonstrates that this component would be most likely to contain an exploitable DoS or RCE vulnerability, as failures were triggered by more test cases in this component than any other.

Unlike the P-CSCF, the S-CSCF saw failures with a larger number of different fuzzers. This is likely due to the S-CSCF being the final processor of most SIP requests, while the P-CSCF and I-CSCF act largely as proxies, transporting messages between the UA and the S-CSCF, which processes and responds to the SIP messages.

While the S-CSCF is less exposed to outside users than the P-CSCF, malicious or 3rd party application servers may have the ability to interface and thus attack the S-CSCF. The "smart" fuzzers were far more effective in creating test cases that triggered crashes against the S-CSCF. This shows that the flaws were discovered slightly deeper within the protocol processing code in this component, and also highlights the usefulness of fuzzers that contain some intelligence about the protocol being utilised for testing.

5.2.3 P-CSCF

Table 5.3 shows the results obtained when running each fuzzer against the P-CSCF IMS SIP component. This component experiences the fewest failures.

Fuzzer	Triggered Crashes
SIPACKFuzzer	0
SIPDumbACKFuzzer	0
SIPDumbCANCELFuzzer	0
SIPDumbREGISTERFuzzer	13
SIPInviteCommonFuzzer	1
SIPInviteOtherFuzzer	0
SIPInviteRequestLineFuzzer	0
SIPInviteStructureFuzzer	0
SIPNOTIFYFuzzer	0
SIPSUBSCRIBEFuzzer	0

Table 5.3: P-CSCF Results

The P-CSCF experienced failures with only 14 test cases. As the P-CSCF is usually the first point of contact for the UA, this component would also form the most readily accessible attack vector for malicious users. The results above show that REGISTER messages could be used to trigger undesirable behaviour, meaning that malicious users may not even require legitimate credentials to perform DDoS or possibly RCE attacks on the P-CSCF component.

Only a single other test case was generated that was capable of causing this component to fail. By observing that this component experiences the lowest number of failures, it appears that it would provide a limited attack surface to a malicious adversary.

5.2.4 Discussion of "dumb" versus "smart" fuzzer results

Fuzzer Type	Total Crashes Triggered	Unique Fuzzers that Triggered Crashes
"Dumb"	115	2
Smart	31	4

Table 5.4: Summary of results by fuzzer type

While the REGISTER fuzzer (classified as "dumb" fuzzer above") created the largest number of error-triggering test cases, it was one of only two of the "dumb" fuzzers that produced any useful results. The REGISTER message was a special case, as the components being tested would attempt to parse these messages without any prior setup or authentication, as the REGISTER itself forms part of the authentication process. If we remove the results of the REGISTER fuzzer, only 3 other error-generating

test cases were discovered by "dumb" fuzzers (versus the 31 error-generating test cases discovered by 4 different "smart" fuzzers). A larger number of smart fuzzers however generated test cases that triggered errors, showing that by performing basic state tracking and authentication, errors deeper within the implementation can be discovered. "Deeper" in this case is used to mean that the test cases would only trigger an error after both authentication was performed, and possibly some initial setup. For example, if testing SIP CANCEL messages, an INVITE would be sent first, before sending invalid CANCEL messages.

The results therefore demonstrate that the "smart" or protocol and state aware fuzzers were able to provide greater coverage within the protocol instead of only generating test cases which exercise the components' ability to parse a single message, as the "dumb" fuzzers do.

5.3 Analysis of Discovered Flaws

Three distinct modes of failure were observed when test cases triggered errors. The different failure modes were not confined to a particular fuzzer, and in many cases the same fuzzer triggered both failure modes. To perform the analysis, the GNU debugger (gdb) was used. The component under test was run under gdb instead of under the process monitor, and a test case known to trigger each of the different failure modes was replayed to the component. When the crash, exit or buffer overflow occurred, the process could then be halted and its current state inspected with gdb.

5.3.1 Segmentation Faults

Two different symptoms of segmentation faults were observed. The first type was characterised by the parent process crashing and all children being terminated. In the case of the S-CSCF, this was found to be caused by an error in the 'scscf.so' module used by the SIP Express Router process. The observed errors were a result of an attempt to reference a structure via a NULL pointer within the 'scscf.so' module. Identical behaviour was observed with the I-CSCF in the 'icscf.so' module. [34]

The second type of segmentation fault was characterised by all child processes terminating as a result of the parent process entering an infinite loop, and its CPU usage rising to 100%. The parent process itself then died too. This was observed to be specific to the S-CSCF. The cause of this was located, and appeared to be a result of the child threads dying without the parent process' knowledge. The

parent process then attempted to yield control to the terminated threads unsuccessfully. A timeout for the yield then expired, and the parent died shortly after, resulting in a segmentation fault. [34]

5.3.2 Exit with error code 0

When the S-CSCF failed with this error, it was observed to be a result of the 'scscf.so' module attempting to free a block of memory multiple times. This is referred to as a "double free" vulnerability, and is in some cases exploitable by an attacker to gain control over the software in which this error occurs. Identical behaviour was observed when testing the P-CSCF. This error was caught and handled gracefully, allowing the process to exit and return a 0 status code. [34]

5.3.3 Buffer overflows

The source of the "buffer overflow" errors was discovered to be a result of a buffer length variable being overwritten, and then being read as a negative value. While labelled by the operating systems built-in protection mechanism as a buffer overflow, these were a result of invalid arguments being passed to a function used to copy memory blocks from one place to another. The error was located in the 'tm.so' module used by all components within the Open IMS Core. [34]

Chapter 6

Conclusion and Future Work

With IMS gaining popularity and being increasingly adopted by operators around the world, security within these deployments has become of critical importance. As vendors may implement IMS compatible networks and components slightly differently to each other, it is extremely useful to have access to a tool or framework that can provide some level of insight into the quality and fault resilience of these implementations. Many toolkits exist to test various other protocols, such as FTP, HTTP, SQL servers, binary protocols and even standard SIP, but there were limited tool available aimed at gauging the security and robustness of IMS deployments.

This project detailed the motivation for an IMS-SIP specific security testing framework, as well as evaluating existing similar (but not identical) tools, frameworks and utilities. The evaluation concluded that existing tools could be modified to create a framework capable of interacting with IMS SIP components, and that would allow for fuzz testing of these components for implementation related security flaws. The architecture and implementation was detailed, and the developed tools tested against the IMS SIP components.The framework made use of existing open source frameworks and SIP related tools to allow for rapid prototyping of an IMS-SIP specific fuzzing framework. Adaptions were made where necessary to ensure that the developed framework interacted correctly with the IMS SIP components, and was also able to authenticate to them where necessary.

The results showed that a number of flaws were discovered when using the Open IMS Core as a target. While the Open IMS Core is not intended to be used in production, the results demonstrated that the class of security testing tools known as "fuzzers" are useful in discovering flaws within IMS implementations. The results also suggested that state-aware fuzzers performed better than merely

feeding random data to the SIP components to attempt to locate flaws.

Furthermore, basic analysis of the flaws discovered showed that the framework was effective at not only locating flaws, but also useful in re-triggering these flaws reliably. This ability allowed for easier investigation into the source of some of the discovered implementation flaws, which would assist in resolving them to avoid further failures.

6.1 Conclusions

Based on the results obtained in Chapter 5, the following conclusions have been drawn:

- Existing open-source tools provide a solid basis for creating protocol specific fuzzers. This was shown by utilising an existing framework and existing SIP oriented testing tool to develop a prototype of a new fuzz testing framework capable of interacting with IMS SIP components. The interoperability was then verified by implementing a test bed and running the newly developed framework against the IMS SIP components.
- Stateful fuzzing located a wider range of flaws within the tested components
- Fuzzers with little or no intelligence also performed well, but didn't uncover as broad a range of flaws as the "smart" fuzzers
- Fuzz testing is a viable method of assessing the implementation of IMS SIP components

The results obtained may be combined with knowledge of the existing attacks mentioned in section 2.2.1, which are not dealt with by the developed framework, to assist in hardening both the deployment and implementation of a given IMS framework.

6.2 Recommendations and Future Work

The current prototype deals only with the SIP protocol and has been tested only with the Open-IMS Core implementation. Future work would involve adding protocol support for other protocols heavily utilised within IMS implementations such as Diameter. Additionally, a larger number of SIP messages could be tested by extending the existing prototype, improving test coverage for a given implementation.

Other work has utilised stateful approaches to tracking the effectiveness of a fuzzer, as in [35]. This technique could be applied to the IMS dialect of SIP, and possibly Diameter to provide deeper test coverage as well as more accurately gauge effectiveness of the fuzzing techniques chosen.

IMS frameworks generally support a number of different authentication mechanisms. The current prototype utilises the commonly supported IMS-AKAv1 mechanism. Other authentication mechanisms may be added to improve support for IMS implementations from other vendors, as well as to allow for testing of the REGISTER messages themselves.

Process monitoring is supported by the prototype, but may also be extended to save more details about the state of the process when a crash is encountered. Utilising the linux ptrace system calls, or running the tested components under a debugger would yield further insight into the failure modes of the software under test.

Bibliography

[1] 3GPP. 3G security; Access security for IP-based services. TS 33.203, 3rd Generation Partnership Project (3GPP), October 2010.

[2] Thomas Porter. *Practical VoIP Security.* Syngress, 2006.

[3] P Amini and A Protnoy. Sulley fuzzing framework, 2010.

[4] Gonzalo Camarillo and Miguel-Angel Garca-Martn. The 3g ip multimedia subsystem: Merging the internet and the cellular worlds. 2008.

[5] Open IMS core - FHoSS: the open IMS core HSS. `http://www.openimscore.org/docs/FHoSS/main.html`. Online; accessed 10-July-2013.

[6] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543 (Proposed Standard), March 1999. Obsoleted by RFCs 3261, 3262, 3263, 3264, 3265.

[7] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665, 6878.

[8] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle. Session Initiation Protocol (SIP) Extension for Instant Messaging. RFC 3428 (Proposed Standard), December 2002.

[9] Ericsson. Current status of ims deployment strategies & future developments. Technical report, 2012.

[10] SHODAN - computer search engine. `http://www.shodanhq.com/`. Online; accessed 19-August-2014.

[11] P. Amini M. Sutton, A. Greene. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, 2007.

[12] G. Mayer M. Poikselka. *The IMS: IP Multimedia Concepts and Services, 3rd Edition.* Wiley, 2006.

[13] Gilles Bertrand. The ip multimedia subsystem in next generation networks. *Rapport technique, ENST Bretagne*, 7, 2007.

[14] 3GPP. IP multimedia call control protocol based on Session Initiation Protocol (SIP) and Session Description Protocol (SDP); Stage 3. TS 24.229, 3rd Generation Partnership Project (3GPP), September 2013.

[15] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327 (Proposed Standard), April 1998. Obsoleted by RFC 4566, updated by RFC 3266.

[16] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. Updated by RFC 6157.

[17] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko. Diameter Base Protocol. RFC 3588 (Proposed Standard), September 2003. Obsoleted by RFC 6733, updated by RFCs 5729, 5719, 6408.

[18] Michael T Hunter, Russell J Clark, and Frank S Park. Security issues with the ip multimedia subsystem (ims): A white paper. *College of Computing, Georgia Institute of Technology*, 2007.

[19] 3GPP. 3G security; Network Domain Security (NDS); IP network layer security. TS 33.210, 3rd Generation Partnership Project (3GPP), June 2010.

[20] Kai-Di Chang, Chi-Yuan Chen, Jiann-Liang Chen, and Han-Chieh Chao. Challenges to next generation services in ip multimedia subsystem. 2010.

[21] Frank S Park, Devdutt Patnaik, Chaitrali Amrutkar, and Michael T Hunter. A security evaluation of ims deployments. In *Internet Multimedia Services Architecture and Applications, 2008. IMSAA 2008. 2nd International Conference on*, pages 1–6. IEEE, 2008.

[22] R. Patton. *Software Testing.* Sams, 2000.

[23] Noam Rathaus and Gadi Evron. *Open source fuzzing tools.* Syngress, 2007.

[24] Dave Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 2002.

[25] Charlie Miller and Zachary NJ Peterson. Analysis of mutation and generation-based fuzzing. *White Paper, Independent Security Evaluators, Baltimore, Maryland (securityevaluators.com/files/papers/analysisfuzzing.pdf)*, 2007.

[26] Jared DeMott, Richard Enbody, and William F Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. In *Black Hat and Defcon*, 2007.

[27] Autodafe. http://autodafe.sourceforge.net/. Online; accessed 30-October-2014.

[28] Martin Vuagnoux. Autodafe: An act of software torture. In *22nd Chaos Communications Congress, Berlin, Germany*, 2005.

[29] lxapi. `http://lxapi.sourceforge.net/`. Online; accessed 27-June-2013.

[30] Michael Eddington. Peach fuzzing platform. *Peach Fuzzer*, 2011.

[31] Pedram Amini and A Portnoy. Fuzzing frameworks. In *Black Hat USA*. Blackhat, 2007.

[32] VoIPER. `http://voiper.sourceforge.net/`. Online; accessed 1-Oct-2014.

[33] Matt Hillman. 15 minute guide to fuzzing. `https://www.mwrinfosecurity.com/knowledge-centre/15-minute-guide-to-fuzzing/`, August 2013. Online; accessed 30-October-2014.

[34] Denver D Abrey and Neco Ventura. Vulnerability discovery and analysis within the open source ims core. In *Southern Africa Telecommunication Networks and Applications Conference (SATNAC)*, 2011.

[35] Humberto J. Abdelnur, Radu State, and Olivier Festor. Kif: A stateful sip fuzzer. In *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*, IPTComm '07, pages 47–56, New York, NY, USA, 2007. ACM.

Appendix A

Software Details

This appendix details the software and software version numbers where applicable that were used to create both the vulnerability assessment framework and the testbed.

The following software was used in creating the vulnerability assessment framework:

- VoIPER, version 0.07
- Sulley, included with VoIPER
- Python, version 2.6.6
- ThreeGPP python library

The following software was used in when creating the framework and testbed.

- Ubuntu Desktop Linux Distribution, version 10.10
- Open IMS Core, revision 1175
- Berkeley Internet Name Domain (BIND), version 9.7.1-P2

Appendix B

Hardware

This appendix describes the hardware environment used to host the development and testbed environments.

Oracle Virtualbox was used to provide a virtualised hardware platform on which to run both the development environment and testbed. The following configuration was utilised:

- CPU: 1
- Execution cap 100%
- Memory: 2048MB
- Chipset: PIIX
- I/O APIC: Enabled
- VT-x/AMD-V: Enabled
- Nested Paging: Enabled
- Display Video Memory: 12 MB
- 3D Acceleration: Disabled
- 2D Video Acceleration: Disabled
- Storage Controller: IDE
- IDE Secondary Master (CD/DVD): Empty

- Controller: SATA
- SATA Port 0: Ubuntu80GBImage.vdi (Normal, 76.34 GB)
- Audio Host Driver: Windows DirectSound
- Controller: ICH AC97
- Network Adapter 1: Intel PRO/1000 MT Desktop (NAT)

Appendix C

Framework Configuration File

This appendix contains the configuration file used when running the vulnerability assessment framework against the IMS SIP components.

```
# This is the user/password combo that register will use
user = alice password = alice
# This is the user to target on the device you are testing
target_user = bob
# Set this if you want INVITEs to be cancelled
do_cancel = True
# Set this if you want the fuzzer to register with the target device
# before fuzzing
# The REGISTER will be resent every 50 requests
do_register = True
# DDA: New mods for IMS and difficult SIP server compat.
auth_type = aka
auth_digest_username = alice@open-ims.test
local_ip = 192.168.0.22
```

Appendix D

Command Listing

This appendix provides the commands used to initiate and run the fuzzers against the various IMS components. An example is also provided, along with typical output.

Initating a fuzzing session

```
./gofuzz.sh -f <Fuzzer> -c <Component>
```

Where <Fuzzer> is any of:

- SIPInviteStructureFuzzer
- SIPInviteRequestLineFuzzer
- SIPInviteCommonFuzzer
- SIPInviteOtherFuzzer
- SDPFuzzer
- SIPDumbACKFuzzer
- SIPDumbCANCELFuzzer
- SIPDumbREGISTERFuzzer
- SIPSUBSCRIBEFuzzer

- SIPNOTIFYFuzzer
- SIPACKFuzzer

<Component> may be any of:

- icscf
- pcscf
- scscf

For example, running the below command using the script shown in Apendix E would start the SIP SUBSCRIBE message fuzzer against the S-CSCF component, utilising the correct port. The script assumes all components are running on their standard port numbers on the same host that the script is running on.

```
./gofuzz.sh -f SIPInviteCommonFuzzer -c pcscf
```

Example output of the above is shown below. Only the output of the script and fuzzer are shown. Debug output from the component under test will also be displayed, but has been omitted, as this is generated by the OpenIMS core itself, and not the fuzzing script or framework. The below demonstrates the fuzzer performing a successful registration beginning to send the INVITE test cases to the component under test.

```
listen port: 50000
DDA: using socket from fuzzer_parents
DDA: added already bound port 50000
DDA: not adding socket for already bound port
sending register init
wait time 0.8
SA-EXPECTING [99, 98, 59]
got something from generator off queue

REGISTER sip:open-ims.test SIP/2.0
To: <sip:alice@open-ims.test>
From: <sip:alice@open-ims.test>;tag=1iwxnfcrdhskm7z9golpbeq20j3a5vyt
```

```
Call-ID: register.1iwxnfcrdhskm7z9golpbeq20j3a5vyt
CSeq: 55511 REGISTER Via: SIP/2.0/UDP open-ims.test:50000;branch=
    ↪ z9hG4bKk1iwxnfcrdhskm7z9golpbeq20j3a5vyt
Max-Forwards: 70
Allow: INVITE,ACK,OPTIONS,BYE,CANCEL,NOTIFY,REFER,MESSAGE
Contact: <sip:alice@open-ims.test:50000>
Expires: 3600
Source is generator. Sending
grabbing last socket from socket_list in transceiver
transceiver sending
SIP/2.0 401 Unauthorized - Challenging the UE
To: <sip:alice@open-ims.test>;tag=4f9fd3a079105910b2e6fe0ca6590818-b326
From: <sip:alice@open-ims.test>;tag=1iwxnfcrdhskm7z9golpbeq20j3a5vyt
Call-ID: register.1iwxnfcrdhskm7z9golpbeq20j3a5vyt
CSeq: 55511 REGISTER
Via: SIP/2.0/UDP open-ims.test:50000;received=192.168.0.22;rport=50000;
    ↪ received=192.168.0.22;branch=
    ↪ z9hG4bKk1iwxnfcrdhskm7z9golpbeq20j3a5vyt
Path: <sip:term@pcscf.open-ims.test:4060;lr>
Service-Route: <sip:orig@scscf.open-ims.test:6060;lr>
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY,
    ↪ PUBLISH, MESSAGE, INFO
Server: Sip EXpress router (2.1.0-dev1 OpenIMSCore (x86_64/linux))
Content-Length: 0
Warning: 392 192.168.0.22:6060 "Noisy feedback tells:  pid=6328
    ↪ req_src_ip=192.168.0.22 req_src_port=5060 in_uri=sip:scscf.open-ims
    ↪ .test:6060 out_uri=sip:scscf.open-ims.test:6060 via_cnt==3"
WWW-Authenticate: Digest realm="open-ims.test", nonce="5SWEn4erMG+8BZ2vH+
    ↪ hBgLSvndJxRAAA0poY9hpTIk4=", algorithm=AKAv1-MD5, qop="auth,auth-
    ↪ int"

TM: 1 transactions being monitored
```

```
SA-GOT-CORRECT-BRANCH
SA-GOT-CORRECT-R-CODE 59 # [99, 98, 59]
sending register final: auth_type == aka
wait time 0.8
SA-EXPECTING [98, 99]
got something from generator off queue
REGISTER sip:open-ims.test SIP/2.0
CSeq: 55512 REGISTER
Via: SIP/2.0/UDP open-ims.test:50000;branch=
    ↪ z9hG4bKk1iwxnfcrdhskm7z9golpbeq20j3a5vyt
To: <sip:alice@open-ims.test>
From: <sip:alice@open-ims.test>;tag=1iwxnfcrdhskm7z9golpbeq20j3a5vyt
Call-ID: register.1iwxnfcrdhskm7z9golpbeq20j3a5vyt
Authorization: Digest username="alice@open-ims.test", realm="open-ims.
    ↪ test", nonce="5SWEn4erMG+8BZ2vH+hBgLSvndJxRAAA0poY9hpTIk4=", uri="
    ↪ sip:open-ims.test", algorithm=AKAv1-MD5, response="
    ↪ a8bb8ff944fcb9a86c59d69ad40680fe"
Allow: INVITE,ACK,OPTIONS,BYE,CANCEL,NOTIFY,REFER,MESSAGE
Contact: <sip:alice@open-ims.test>
Expires: 3600
Max-Forwards: 70


Source is generator. Sending
grabbing last socket from socket_list in transceiver
transceiver sending
DDA: not adding socket to list again!
TM: 1 transactions being monitored
SIP/2.0 200 OK - SAR succesful and registrar saved
CSeq: 55512 REGISTER
Via: SIP/2.0/UDP open-ims.test:50000;received=192.168.0.22;rport=50000;
    ↪ received=192.168.0.22;branch=
```

```
    ↪ z9hG4bKk1iwxnfcrdhskm7z9golpbeq20j3a5vyt
To: <sip:alice@open-ims.test>;tag=4f9fd3a079105910b2e6fe0ca6590818-14fd
From: <sip:alice@open-ims.test>;tag=1iwxnfcrdhskm7z9golpbeq20j3a5vyt
Call-ID: register.1iwxnfcrdhskm7z9golpbeq20j3a5vyt
P-Associated-URI: <sip:alice@open-ims.test>
Contact: <sip:alice@open-ims.test>;expires=3600
Path: <sip:term@pcscf.open-ims.test:4060;lr>
Service-Route: <sip:orig@scscf.open-ims.test:6060;lr>
Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, SUBSCRIBE, NOTIFY,
    ↪ PUBLISH, MESSAGE, INFO
P-Charging-Function-Addresses: ccf=pri_ccf_address
Server: Sip EXpress router (2.1.0-dev1 OpenIMSCore (x86_64/linux))
Content-Length: 0
Warning: 392 192.168.0.22:6060 "Noisy feedback tells:  pid=6330
    ↪ req_src_ip=192.168.0.22 req_src_port=5060 in_uri=sip:scscf.open-ims
    ↪ .test:6060 out_uri=sip:scscf.open-ims.test:6060 via_cnt==3"


TM: 1 transactions being monitored
SA-GOT-CORRECT-BRANCH
SA-GOT-CORRECT-R-CODE 98 # [98, 99]
INVITE sip:bob@open-ims.test SIP/2.0
CSeq: 1073741814 INVITE
Via: SIP/2.0/UDP open-ims.test:50000;rport;branch=
    ↪ z9hG4bKg4o8pstaynlu2k0he715fxq9brjivcz6
Route: <sip:orig@scscf.open-ims.test:6060;lr>
From: "alice" <sip:alice@open-ims.test>;tag=somefromtagval
Call-ID: g4o8pstaynlu2k0he715fxq9brjivcz6@TheKlatchianHead
To: "bob" <sip:bob@open-ims.test>
Contact: <sip:alice@open-ims.test:50000>
Max-Forwards: 70
P-Preferred-Identity: <sip:alice@open-ims.test>
```

```
Privacy: none
Require: precondition
Require: sec-agree
Proxy-Require: sec-agree
Content-Length: 100

v=0
o=- 1190505265 1190505265 IN IP4 open-ims.test
s=Opal SIP Session
c=IN IP4 open-ims.test
t=0 0
m=audio 5028 RTP/AVP 101 96 3 107 110 0 8
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=rtpmap:96 SPEEX/16000
a=rtpmap:3 GSM/8000
a=rtpmap:107 MS-GSM/8000
a=rtpmap:110 SPEEX/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 5030 RTP/AVP 31
a=rtpmap:31 H261/90000

DDA: not adding socket to list again!
TM: 2 transactions being monitored
```

Viewing and replaying test cases

The "gofuzz.sh" script stores session and crash log files in a directory named $BASEDIR/<Component Name>/<Fuzzer Name>/. In the previous example, the crashlog files would be located in 2014/P-CSCF/SIPInviteCommonFuzzer directory. A session file (sulley.session) in this directory stores information about the current progress of the currently running fuzzer. This allows for a fuzzing session

to be resumed later without having to re-run all test cases from the beginning. The crash log files contain the message or test case that triggered a crash of failure in the component under test. These files can be used to reliably trigger crashes or failures for further investigation later on, and assist in debugging the cause of the failure by providing a reliable means to trigger the observed failure or crash. The filenames include the test case number that caused the crash.

To view a crash log file, the unix cat command can simply be invoked as below, which will print the output of the file to the current terminal.

```
denver@lightning:~/voiper/trunk/2014/I-CSCF/SIPInviteCommonFuzzer$ cat 1
    ↪ _12332.crashlog
INVITE sip:bob@192.168.0.22 SIP/2.0
CSeq: 1 INVITE
Via: SIP/2.0/UDP 192.168.0.22:50000;rport;branch=
    ↪ z9hG4bK7kn6vha8jr203w1ibf9zmelyxdosguc4
Route: <sip:orig@scscf.open-ims.test:6060;lr>
From: "alice" <sip:alice@open-ims.test>;tag=somefromtagval
Call-ID: @TheKlatchianHead
To: "bob" <sip:bob@192.168.0.22>
Contact: <sip:alice@192.168.0.22:50000>
Max-Forwards: 70
P-Preferred-Identity: <sip:alice@open-ims.test>
Privacy: none
Require: precondition
Require: sec-agree
Proxy-Require: sec-agree
Content-Length: 100

v=0
o=- 1190505265 1190505265 IN IP4 192.168.0.22
s=Opal SIP Session
c=IN IP4 192.168.0.22
t=0 0
m=audio 5028 RTP/AVP 101 96 3 107 110 0 8
```

```
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
a=rtpmap:96 SPEEX/16000
a=rtpmap:3 GSM/8000
a=rtpmap:107 MS-GSM/8000
a=rtpmap:110 SPEEX/8000
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
m=video 5030 RTP/AVP 31
a=rtpmap:31 H261/90000
denver@lightning:~/voiper/trunk/2014/I-CSCF/SIPInviteCommonFuzzer$
```

A utility script called crash_replay.py, which forms part of the modified VoIPER framework, can be utilised to replay a specific test case, reproducing the failure as desired. This script simply reads the file and sends it to the component under test on the same host and port that was utilised when performing the fuzz testing run. An example invocation of the script, which will replay the above test case, is shown below.

```
python crash_replay.py -f ./1_12332.crashlog -i localhost
```

Appendix E

Fuzzing Script

This appendix contains the fuzzing script (saved as "gofuzz.sh") created to run a given fuzzer against the selected IMS SIP component. The script was created to make running a single fuzzer against a SIP component easier by allowing only a single command to be used instead of having to start up the process monitor and fuzzing engine. Additionally it assists in placing session and crash log files in the correct directory, as well as providing useful names for the crash log files.

```
#!/bin/bash

FRAMEWORKHOME="/home/denver/voiper/trunk"
declare -A cscfpath[icscf]="I-CSCF"
declare -A cscfpath[scscf]="S-CSCF"
declare -A cscfpath[pcscf]="P-CSCF"
declare -A cscfport[icscf]=5060
declare -A cscfport[scscf]=6060
declare -A cscfport[pcscf]=4060

#export FUZZER=$1
#export CDIR=${cscfpath[$2]}
export BASEDIR="2014"

function help {
```

```
        echo "Usage: $0 [-b <basedir>] -f <fuzzer> -c <icscf|
            ↪ scscf|pcscf>"
}

while getopts hbf:c: opt; do
        case $opt in
        f)
                FUZZER=$OPTARG
        ;;
        c)
                COMPONENT=$OPTARG
                CDIR=${cscfpath[$COMPONENT]}
        ;;
        b)
                BASEDIR=$OPTARG
        ;;
        h)
                help
                exit 0
        ;;
        esac
done
if [[ -z $FUZZER ]] || [[ -z $COMPONENT ]] || [[ -z $CDIR ]]; then
        echo "Error, please specify both -f and -c with arguments"
        exit 1
fi

python $FRAMEWORKHOME/sulley/nix_process_monitor.py \
-c $BASEDIR/$CDIR/$FUZZER/$COMPONENT-$FUZZER.crashbin \
2>&1 | tee -a procmon.log &

python $FRAMEWORKHOME/fuzzer.py -f $FUZZER -i 192.168.0.22 \
```

```
-p ${cscfport[$COMPONENT]} -a "$BASEDIR/$CDIR/$FUZZER" -e \
-c3 -S "/opt/OpenIMSCore/$COMPONENT.sh" \
-t "./ser_ims/cfg/killser $COMPONENT" \
2>&1 | tee -a fuzzer.log
```

Appendix F

Accompanying CD-ROM

This document is accompanied by a CD-ROM containing the following:

- An electronic version of this document
- The prototype software developed as part of this project, and used to demonstrate fuzzing against the IMS SIP components
- A copy of all reference material
- Any auxiliary scripts used to supplement the developed prototype