Fuzzing for software vulnerability discovery

Toby Clarke

Technical Report
RHUL-MA-2009-04
17 February 2009

Department of Mathematics
Royal Holloway, University of London
Egham, Surrey TW20 0EX, England
http://www.rhul.ac.uk/mathematics/techreports

TABLE OF CONTENTS

Table of Contents **1**

Introduction **6**

1 The Case for Fuzzing **9**
1.1 The Need for Secure Software 9
1.1.1 Software Vulnerabilities: The Source of the Problem 10
1.1.2 The Defence in Depth Approach 12
1.1.3 Network Solutions for Software Problems 13
1.1.4 Software Vulnerabilities are a Root Cause of Information Security Risk 14
1.1.5 The Influence of End-User Testing 15
1.2 Objectives for this Project 16

2 Software Vulnerabilities **18**
2.1 Software Vulnerability Classes 18
2.1.1 Design Vulnerabilities 19
2.1.2 Implementation Vulnerabilities 19
2.1.3 Operational Vulnerabilities 19
2.2 Implementation Errors 20
2.3 The Need for Input Validation 23
2.4 Differentiation Between Instructions and Data 24
2.5 Escalation of Privilege 24
2.6 Remote Code Execution 24
2.7 Trust Relationships 25
2.8 Command Injection 26
2.9 Code Injection 27

2.10 Buffer Overflows 27
2.11 Integer Overflows 28
2.12 Signedness Issues 29
2.13 String Expansion 29
2.14 Format Strings 30
2.15 Heap Corruption 32
2.16 Chapter Summary 34

3 Software Security Testing **35**
3.1 Software Testing 35
3.2 Software Security Testing 36
3.3 Structural, 'White Box' Testing 37
3.3.1 Static Structural Analysis 37
3.3.2 Dynamic Structural Testing 41
3.4 Functional, 'Black Box' Testing 41
3.5 Chapter Summary 43

4 Fuzzing – Origins and Overview **44**
4.1 The Origins of Fuzzing 44
4.2 A Basic Model of a Fuzzer 45
4.3 Fuzzing Stages 46
4.3.1 Target Identification 46
4.3.2 Input Identification 48
4.3.3 Fuzz Test Data Generation 49
4.3.4 Fuzzed Data Execution 50
4.3.5 Exception Monitoring 50
4.3.6 Determining Exploitability 51
4.4 Who Might Use Fuzzing 51
4.5 The Legality of Fuzz Testing 53
4.6 Chapter Summary 53

5 Random and Brute Force Fuzzing **54**
5.1 Application Input Space 54
5.2 Random Data Generation 56
5.2.1 Code Coverage and Fuzzer Tracking 56
5.2.2 Static Values 59
5.2.3 Data Structures 60
5.3 Brute Force Generation 62
5.4 Chapter Summary 63

6 Data Mutation Fuzzing **65**
6.1 Data Location and Data Value . . . 66
6.2 Brute Force Data Mutation . . . 66
6.2.1 Brute Force Location Selection . . . 66
6.2.2 Brute Force Value Modification . . . 68
6.3 Random Data Mutation . . . 68
6.4 Data Mutation Limitations . . . 68
6.4.1 Source Data Inadequacy . . . 69
6.4.2 Self-Referring Checks . . . 70
6.5 Chapter Summary . . . 71

7 Exception Monitoring **73**
7.1 Sources of Monitoring Information . . . 74
7.2 Liveness Detection . . . 75
7.3 Remote Liveness Detection . . . 76
7.4 Target Recovery Methods . . . 76
7.5 Exception Detection and Crash Reporting . . . 78
7.6 Automatic Event Classification . . . 78
7.7 Analysis of Fuzzer Output . . . 79
7.8 Write Access Violations . . . 81
7.9 Read Access Violations on EIP . . . 81
7.10 Chapter Summary . . . 81

8 Case Study 1 – 'Blind' Data Mutation File Fuzzing **83**
8.1 Methodology . . . 85
8.2 FileFuzz . . . 85
8.3 FileFuzz Configuration . . . 89
8.3.1 FileFuzz Create Module Configuration . . . 89
8.3.2 The Rationale for Overwriting Multiple Bytes at a Time . . . 90
8.3.3 The Rationale for Overwriting Bytes with the Value Zero . . . 91
8.3.4 FileFuzz Execute Module Configuration . . . 91
8.4 FileFuzz Creation Phase . . . 92
8.5 FileFuzz Execution Phase . . . 94
8.6 Results Analysis . . . 94
8.7 Lessons Learned . . . 96

9 Case Study 2 – Using Fuzzer Output to Exploit a Software Fault **99**
9.1 Methodology . . . 101
9.1.1 Obtaining the Shell Code . . . 102

9.1.2 Identifying a Suitable Location for the Shell Code 102
9.1.3 Inserting Shell Code Into `Access.cpl` 108
9.1.4 Redirecting Execution Flow to Execute the Shellcode 108
9.2 Results 108
9.3 Conclusions 112

10 Protocol Analysis Fuzzing 115
10.1 Protocols and Contextual Information 116
10.2 Formal Grammars 117
10.3 Protocol Structure and Stateful Message Sequencing 117
10.4 Tokenisation 120
10.4.1 Meta Data and Derived Data Elements 120
10.4.2 Separation of Data Elements 122
10.4.3 Serialization 123
10.4.4 Parsing 124
10.4.5 Demarshalling and Parsing in Context 124
10.4.6 Abstract Syntax Notation One 125
10.4.7 Basic Encoding Rules 126
10.4.8 Fuzzing Data Elements in Isolation 126
10.4.9 Meta Data and Memory Allocation Vulnerabilities 127
10.4.10 Realising Fuzzer Tokenisation Via Block-Based Analysis . . . 128
10.5 Chapter Summary 129

11 Case Study 3 – Protocol Fuzzing a Vulnerable Web Server 130
11.1 Methodology 133
11.1.1 Establish and Configure the Test Environment 134
11.1.2 Analyse the Target 135
11.1.3 Analyse the Protocol 135
11.1.4 Configure the Fuzzer Session 137
11.1.5 Configure the Oracle 139
11.1.6 Launch the Session 143
11.2 Results 143
11.3 Analysis of One of the Defects 144
11.4 Conclusions 145

12 Conclusions 147
12.1 Key Findings 147
12.2 Outlook 149
12.3 Progress Against Stated Objectives 151

A Appendix 1 – A Description of a Fault in the FileFuzz Application 154
A.1 Description of Bug 154
A.2 Cause of the Bug 154
A.3 Addressing the Bug 155

B Appendix 2 – The Sulley Fuzzing Framework Library of Fuzz Strings 156
B.1 Omission and Repetition 157
B.2 String Repetition with \xfe Terminator 157
B.3 A Selection of Strings Taken from SPIKE 158
B.4 Format Specifiers 159
B.5 Command Injection 160
B.6 SQL Injection 160
B.7 Binary Value Strings 161
B.8 Miscellaneous Strings 161
B.9 A Number of Long Strings Composed of Delimiter Characters 162
B.10 Long Strings with Mid-Point Inserted Nulls 162
B.11 String Length Definition Routine 163
B.12 User Expansion Routine 164

C Appendix 3 – Communication with Microsoft Security Response Centre 166

Bibliography 172

INTRODUCTION

The author once spent eight days on a machine woodworking course shaping a single piece of wood to a very particular specification: accuracies of 0.5 mm were required, and our work was assessed with Vernier callipers. At the end of the course we were shown a machine we had not used before: a computer controlled cutting and shaping robot. The instructor punched in the specification, inserted a piece of wood and the machine spent four minutes producing an item equivalent to the one that took us eight days to produce. If properly aligned, we were told, the machine could be accurate to within 0.1 mm.

The example above illustrates that computers are capable achieving tasks that humans cannot. Computers excel at tasks where qualities such as speed, accuracy, repetition and uniformity of output are required. This is because computers excel at following instructions.

However, computers do not excel at writing instructions. In order for a computer to carry out a task, every single component of that task must be defined, and instructions that specify how to complete each component must be provided for the computer in a machine-readable format, termed *software*.

Although there have been advances in areas such as Artificial Intelligence, machine learning and computer generated software, for all practical purposes, computers are dependant upon humans to develop the software they require to function.

Yet, humans are fallible and make mistakes, which result in software defects (termed *bugs*). Software defects introduce uncertainties into computer systems: systems that encounter defects may not behave as expected. The field of Information

Security is concerned with (among other things) protecting the confidentiality, integrity and availability of data. Software defects threaten these and other aspects of data including system reliability and performance.

A subset of software defects render the affected system vulnerable to attacks from malicious parties. Such *vulnerabilities* (weaknesses in controls), may be exploited by criminals, vandals, disaffected employees, political or corporate actors and others to leak confidential information, impair the integrity of information, and / or interfere with its availability. Worse, such attacks may be automated and network-enabled as is the case in internet 'worms': self-propagating software which may contain a malicious payload.

As a result of threats to the security of digitally stored and processed information, a wide range of controls and mitigations have been developed. Commonly applied controls include: *network controls* (e.g. firewalls, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), encryption, integrity checking), *host-based controls* (e.g. host-based IPS / IDS, file integrity checks, authentication, authorisation, auditing) and *application controls* (e.g. input validation, authentication and authorisation). None of these controls address the root cause of the issue: the presence of software defects. Software security testing aims to identify the presence of vulnerabilities, so that the defects that cause them can be addressed.

A range of software security testing methods exist, all of which have benefits and disadvantages. One method of security testing is scalable, automatable and does not require access to the source code: *fuzz testing*, or, *fuzzing*, a form of fault injection stress testing, where a range of malformed input is fed to a software application while monitoring it for failures.

Fuzzing can, and has been used to discover software defects. Since access to the source code is not required, any application that is deployed may be fuzz tested by a malicious party. Hence, fuzzing is a powerful method for attackers to identify software vulnerabilities. With this in mind, it would be sensible for developers to fuzz test applications internally, and to do so as often and as early in the development life cycle as possible. It would also be sensible for software vendors to mandate that any application satisfying certain risk-based criteria should be fuzz tested, (alongside other quality gateways) before release into the live environment.

However, fuzzing cannot reveal all of the software vulnerabilities present in an application; it can only reveal software defects that occur during the implementation

stage of development. If every application was thoroughly fuzz tested before release, software defects would still propagate through the development life cycle, and would still occur in deployment, integration and operation of the application. Worse, fuzzing cannot provide any quantitative assurance over whether testing has been complete or exhaustive. Fuzzing is not a panacea for software defects.

This report explores the nature of fuzzing, its benefits and its limitations. We begin by exploring why software vulnerabilities occur, why software security testing is important, and why fuzz testing in particular is of value. We then focus upon software security vulnerabilities and how they are exploited by attackers. Having covered software vulnerabilities, we move on to examining the various software security testing methods employed to detect them, and place fuzz testing within the wider field of software security testing.

Having covered the background in Chapters 1 to 3, we can focus exclusively upon fuzz testing. Chapter 4 begins with an examination of the origin of fuzzing and we present a basic model of a fuzzer and an overview of the fuzzing process. Following this, we examine the test data generation aspect of fuzzing (where malformed data is created in order to be passed to the target software application), starting with the most basic forms of fuzzing: *random* and *brute force* fuzzing. We will use these basic fuzzing approaches to present some of the fundamental problems that have been solved as fuzzing has developed. We then present a more advanced approach to fuzz test data generation: *'blind' data mutation fuzzing*, identifying the problems it can and cannot solve. Next, we examine the issues around exception monitoring and the analysis of the output of fuzz testing.

Having presented the basic theory behind fuzzing, we present a case study exploring the use of 'blind' data mutation fuzzing to discover software defects in a component of the Windows XP operating system. In a second, related case study, we document the exploitation of one of the previously discovered defects to determine if it represents a security vulnerability, and to determine whether it is possible to construct software exploits based on the output of fuzz testing.

We then explore the theory behind protocol analysis fuzzing, a form of 'intelligent' fuzzing where the structure of input is analysed in order to construct a protocol-aware fuzzer. Protocol analysis fuzzing is then applied in a third case study, where a protocol-aware fuzzer is found to be capable of detecting defects in a vulnerable web server.

CHAPTER

ONE

THE CASE FOR FUZZING

1.1 The Need for Secure Software

The primary focus for software development is satisfying a functional specification. Software functional testing (particularly User Acceptance Testing) is usually employed to evaluate whether requirements are satisfied and identify any that are not. Yet, other factors such as *performance* (the ability to support a number of concurrent active users) and *reliability* (the ability to satisfy requirements over a period of time without interruption or failure) are important to users, particularly in mission-critical applications such as those deployed within aerospace, military, medical and financial sectors. To this end, functional testing may be complimented by other forms of software testing including (but not limited to) unit, integration, regression, performance and security testing, at a range of stages in the software development life cycle, all of which are aimed at identifying software defects so that they may be addressed.

Yet, it is not difficult to find examples of dramatic, high-impact software failures where software defects were not detected or addressed with disastrous results.

- AT&T, January 1990: A bug due to a misplaced `break` led to losses of 1.1 billion dollars, when long distance calls were prevented for 9 hours [10, 19].
- Sellafield UK, September 1991: Radiation doors were opened due to a software bug [10].

- Coventry Building Society UK January 2003: A software failure meant that £850,000 was withdrawn from Automatic Teller Machines over 5 days without being detected by the accounting system [10, 19].
- 1999: The NASA Mars Lander crashed into the surface of Mars due to a software error relating to conversion between imperial and metric units of measure [20, p. 11].

The prevalence of such incidents suggests that there are many hurdles to overcome in order to produce software that is reliable, i.e. that it will perform as intended.

One such hurdle is the capacity of an application to respond in a robust manner to input, regardless of whether that input conforms to defined parameters. George Fuechsel, an early IBM programmer and instructor is said to have used the term *"garbage in, garbage out"* to remind students of the inability of computers to cope with unexpected input [41]. However, in order to achieve reliable performance, the capacity to validate input by the application has become a requirement. Failure to properly validate input can result in vulnerabilities that have a security implication for users. As Information Technology (IT) increasingly processes data that impacts on our daily lives, security vulnerabilities have greater potential to threaten our well-being, and the impetus to ensure their absence is increased.

1.1.1 Software Vulnerabilities: The Source of the Problem

The software development process does not produce secure software applications by default. Historically, this has been due to a number of factors, including:

- the increasing level of software complexity;
- the use of 'unmanaged' programming languages such as C and C++, which offer flexibility and performance over security. [20, 9];
- a lack of secure coding expertise, due to a lack of training and development;
- users have no awareness of (let alone metrics for comparing) application security[1];

[1]While there are many proposed software metrics, Hogland and McGraw suggest that only one appears to correlate well with the number of flaws: Lines of Code (LOC) [20, p. 14]. In other words, the number of defects is proportional to the number of lines of code. To some, this may be the only reasonable metric.

- functionality and performance usually drive purchasing decisions: security is rarely considered at the point of purchase;

- a 'penetrate and patch' approach to software security, where software vulnerability testing is performed after the software is released, and security is often retro-fitted, rather than implemented at the design or inception, this is both costly and unwieldy, often resulting in a poor level of security at high cost [32, 28];

- software testing has focused upon assuring that functional requirements are satisfied, and there has been little resource dedicated to testing whether security requirements are satisfied.

In order to produce a suitably secure software application, a considerable amount of investment in the form of time and money may be required, and security considerations will have to feed into, and impact upon, every phase of the life cycle of a software application. Hence, there must be a compelling argument for funding for security over other competing requirements.

Fortunately, there is an economic argument for addressing software security defects, and for doing so as early as possible in the development life cycle. The cost of addressing defects rises exponentially as the development stages are completed as is shown in Table 1.1 [30].

If the figures in the Table 1.1 seem less than compelling, it's worth considering that the total cost to all parties of a single Microsoft Security Bulletin likely runs into millions of dollars, and the total cost of the more significant internet worms is likely to have reached billions of dollars worldwide [16]. Hence, if the data processed by an application has any value, it may be costly not to define and test security requirements.

Phase	Relative Cost to Correct
Definition	$1
High-Level Design	$2
Low-Level Design	$5
Code	$10
Unit Test	$15
Integration Test	$22
System Test	$50
Post-Delivery	$100

Table 1.1: The exponential rise in cost in correcting defects as software development advances through life cycle phases [30].

1.1.2 The Defence in Depth Approach

The traditional approach to information security has been termed ‘defence in depth’. This means applying a multi-layered approach to security, so that if a security system failure occurs, i.e. an attacker is able to circumvent a control such as a network firewall, other controls are implemented and will act to limit the impact of such a failure; for example, an Intrusion Prevention System (IPS) might detect the malicious activity of an attacker and limit their access, or alternatively, a host-based firewall might prevent an attacker from accessing their target.

The author supports defence in depth; a layered approach is sensible when an attacker only has to find one weakness in controls, while security management staff have to ensure that every control is suitable and operating as expected. However, defence in depth may have a side-effect of making vulnerable software more palatable to customers.

1.1.3 Network Solutions for Software Problems

Information security practitioners have had to apply pragmatic solutions to protect vulnerable software applications. This often meant applying a 'walled garden' network security model, where restricted network access mitigated the risk of remote attacks. However, such an approach provides little defence from insider attack and is restrictive to business and personal usage.

Furthermore, the 'walled garden' model has become increasingly impractical for business that feature large numbers of remote employees, customers, sub-contractors, service providers and partner organisations, many of whom require feature-rich connectivity, often to back-end, business critical systems. This 'breaking down' of network boundaries has been termed *de-perimeterization* by the Jericho Forum thought leadership group[2], who have set out the Jericho Forum Commandments[3], which aim to advise organisations how to maintain IT security in the face of increasing network de-perimeterization.

For many organisations, security, while not leaving the network, is being additionally applied at the end-point: server and desktop operating system builds are being hardened; firewalls and host-based IDS/IPS are placed on end-points, and the 'internal' network is no longer trusted.

Implementing network security-based solutions to address software security problems (i.e. software vulnerabilities) may have contributed to a climate where software is assumed to be insecure, and ultimately, that it is acceptable to produce insecure software.

Software patch management or vulnerability mediation is aimed at managing the risks relating to software vulnerabilities by ensuring that all applications are fully patched where possible, or by configuring 'work-arounds' which mitigate risks. Patch management is critical in that it controls and mitigates risk arising from *known* software vulnerabilities. Patch management does not, however, do anything to stem the tide of new vulnerabilities, nor does its influence extend beyond known, patched vulnerabilities to address undisclosed or un-patched vulnerabilities.

[2] http://www.opengroup.org/jericho/
[3] http://www.opengroup.org/jericho/commandments_v1.2.pdf

1.1.4 Software Vulnerabilities are a Root Cause of Information Security Risk

By failing to identify and focus upon the root causes of risks such as software vulnerabilities there is a danger that the Information Security response becomes solely reactive. This is typified by signature based IPS / IDS, and anti virus solutions: they will always be 'behind the curve' in that they can only respond to existing threats, and can never defend against emerging, previously unseen attacks.

If the objective of Information Security as a profession is to address the root causes of information technology risk (one of which is security vulnerabilities arising from insecure software) it will need to move beyond a purely reactive stance and adopt a strategic approach. This will require more investment on the part of the sponsor of such an activity, but offers the potential of a greater degree of assurance and potentially reduced operational and capacity expenditure in the long run.

Due to the level of investment required, the development of secure coding initiatives has been largely left to governmental and charitable organizations such as the Cyber Security Knowledge Transfer Network (KTN) Secure Software Development Special Interest Group[4] and the Open Web Application Security Project (OWASP).[5]

Vendors such as Microsoft have aimed to address the issue of software vulnerabilities internally through the Secure Windows Initiative (SWI)[6], and have publicly released the Security Development Lifecycle (SDL) methodology.[7]

The Open Source community have also recently benefited from a contract between the U.S. Department of Homeland Security and Coverity[8], a developer of commercial source code analysis tools. Coverity has employed its automated code auditing tools to reveal security vulnerabilities in 11 popular open source software projects.[9]

---

[4] http://www.ktn.qinetiq-tim.net/groups.php?page=gr_securesoft
[5] http://www.owasp.org/index.php/Main_Page
[6] http://www.microsoft.com/technet/archive/security/bestprac/secwinin.mspx
[7] http://msdn.microsoft.com/en-us/security/cc448177.aspx
[8] http://www.coverity.com/index.html
[9] http://www.coverity.com/html/press_story54_01_08_08.html

1.1.5 The Influence of End-User Testing

In *How Security Companies Sucker Us With Lemons* [38], Schneier considers whether an economic model proposed by George Akerlof in a paper titled *The Market for Lemons* [5] can be applied to the information security technology market.

If users are not able to obtain reliable information about the quality of products, information asymmetry occurs where sellers have more information than buyers and the criteria for a 'Lemons market' are satisfied. Here, vendors producing high-quality solutions will be out-priced by vendors producing poor quality solutions until the only marketable solution will be substandard - i.e. a 'lemon' [5].

By the same token, an objective quality metric that can be used to compare products can also influence a market such that products of higher quality command a higher market value [38].

The quality of a product may be brought to the attention of users via standards such as the Kite Mark[10], for example. The Common Criteria is an internationally recognised standard for the evaluation of security functionality of a product. International acceptance of the Common Criteria has meant that:

> *"Products can be evaluated by competent and independent licensed laboratories so as to determine the fulfilment of particular security properties, to a certain extent or assurance."* [11]

However, a Common Criteria evaluation cannot guarantee that a system will be free from security vulnerabilities, because it does not evaluate code quality, but the performance of security-related features [21]. Howard and Lipner set out the limitations of the Common Criteria with regard to software security as follows:

> *"What CC does provide is evidence that security-related features perform as expected. For example, if a product provides an access control mechanism to objects under its control, a CC evaluation would provide assurance that the monitor satisfies the documented claims describing the protections to the protected objects. The monitor might include some implementation security bugs, however, that could lead to a compromised system. No goal within CC ensures that the monitor is free of all implementation security*

[10] http://www.bsi-global.com/en/ProductServices/About-Kitemark/

[11] http://www.commoncriteriaportal.org/

> *bugs. And that's a problem because code quality does matter when it comes to the security of a system."* [21, p. 22]

Evaluation under the Common Criteria can help to ensure that higher quality products can justify their higher cost and compete against lower quality products. However, Common Criteria evaluations can be prohibitively expensive, and do not usually extend to the detection of implementation defects.[12]

Fuzz testing is one method that can be used to reveal software programming errors that lead to software security vulnerabilities. It is relatively cheap, requires minimal expertise, can be largely automated, and can be performed without access to the source code, or knowledge of the system under test. Fuzzing may represent an excellent method for end-users and purchasers to determine if an application has software implementation vulnerabilities.

1.2 Objectives for this Project

This project will explore fuzz testing: a specific form of fault injection testing aimed at inducing software failures by the means of manipulating input. The author devised this project as an opportunity to gain practical experience of fuzz testing and also to develop his understanding of software security testing, software vulnerabilities and exploitation techniques.

My objectives at the outset were to:

- examine the use of fuzzing tools for discovering vulnerabilities in applications;
- examine how the output of a fuzzing tool might be used to develop software security exploits (case study);
- describe the nature, types and associated methodologies of the various different classes of fuzzers;

[12]Implementation defects occur as a result of poor software programming practices and are a primary cause of software vulnerabilities. Software vulnerabilities are discussed in detail in Chapter 2, *Software Vulnerabilities*.

- briefly explain where fuzzers fit within the field of application security testing: i.e. who might use them, why they are used, and what value they offer the Information Security industry, software developers, end-users, and attackers;
- identify some of the limitations of, and problems with, fuzzing;
- compare some of the available fuzzing tools and approaches available possibly using two or more types of fuzzer against a single target application with known vulnerabilities;
- examine the evolution of fuzzing tools, comment on the state of the art and the outlook for fuzzing;
- examine what metrics may be used to compare fuzzers;
- comment on the influence of fuzzing on the information security and software development communities
- compare fuzzing with other forms of software security assurance - i.e. Common Criteria evaluations

CHAPTER

TWO

SOFTWARE VULNERABILITIES

It is impossible to produce complex software applications that do not contain defects. The number of defects per thousand lines of code (referred to as $KLOC$) vary between products, but even where development includes rigorous testing, software products may contain as many as five defects per KLOC [20, p. 14]. Considering that the Windows XP operating system comprises approximately 40 million lines of code, using this very loose rule-of-thumb, it might potentially contain 40,000 software defects [20, p. 15].

Some software defects result in inconsequential 'glitches' that have minimal impact. Other defects have the potential to impact on the security of the application or the data it processes. These security-related defects are termed vulnerabilities, since they represent a weakness, a 'chink in the armour' of the application.

2.1 Software Vulnerability Classes

Vulnerabilities can be grouped in many different ways. Dowd et al. specify three core vulnerability classes, based on software development phases [9, Chapter 1]:

- Design vulnerabilities
- Implementation vulnerabilities
- Operational vulnerabilities

2.1.1 Design Vulnerabilities

The software design phase is where user requirements are gathered and translated to a system specification which itself is translated into a high-level design. More commonly termed *flaws*; design vulnerabilities may occur when security requirements are not properly gathered or translated into the specification, or when threats are not properly identified [9]. Threat modelling is an accepted method for drawing out security requirements and identifying and mitigating threats at during the design phase [21, Chapter 9]. Perhaps the most significant source of vulnerabilities from the design phase occur because: *"Design specifications miss important security details that occur only in code."* [21, p. 23]

Because a design has to be a high-level view of the final product details must be abstracted out [14]. Yet, even the smallest of details can have great impact on the security of a product.

2.1.2 Implementation Vulnerabilities

The software implementation phase is where the design is implemented in code. It is important not to mistake *implementation* with *deployment*.[1]

Implementation errors usually arise due to differences between the perceived and actual behaviour of a software language, or a failure to properly understand the details of a language or programming environment. As Dowd, et al. put it:

> *"These problems can happen if the implementation deviates from the design to solve technical discrepancies. Mostly, however, exploitable situations are caused by technical artefacts and nuances of the platform and language environment in which the software is constructed."* [9, Chapter 1]

2.1.3 Operational Vulnerabilities

Operational vulnerabilities are not caused by coding errors at the implementation stage, but occur as a result of the deployment of software into a specific environment.

[1]Implementation is the software development phase; deployment is where the application is deployed for use in the live, operational environment.

Many factors can trigger operational vulnerabilities, including: configuration of the software or software and hardware it interacts with, user training and awareness, the physical operating environment and many others. Types of operational vulnerabilities include social engineering, theft, weak passwords, unmanaged changes, and many others.

Errors that occur at the design or operation phase are sometimes detectable using fuzzing, but the vast majority of defects that are revealed by fuzzing are attributable to the *implementation* phase, where concepts are implemented in software. From a software developer's perspective, fuzzing would be ideally performed during the implementation phase.

Having identified the implementation phase as being the primary source of the type of errors that are detectable via fuzzing, the rest of this chapter will focus on implementation errors and the potential security vulnerabilities that they may cause.

2.2 Implementation Errors

The bulk of implementation errors will be detected during the implementation phase by compiler errors or warnings[2], and activities such as Unit and other testing. However, it is highly unlikely that all implementation errors will be detected, since there is usually finite resource for testing, and most of the focus of software testing is on testing that a product will satisfy functional requirements, not security requirements.

Incidentally, Attack Surface Analysis (ASA) and Attack Surface Reduction (ASR) are aspects of the Secure Development Lifecycle that account for the inevitable presence of defects in production code by minimising risk where possible [21, p. 78]. The approach here is to accept that defects will inevitably propagate through the development, and to apply the principles espoused by Saltzer and Schroeder such as *least privilege* and *economy of mechanism* by identifying any areas of exposure and minimising these where possible such that the impact of a vulnerability can be reduced [21, p. 78].

Of the errors that propagate through the various phases of development, some

[2] However, most compilers are, by default, unable to test the logic of dynamic operations involving variables at compile time, meaning that vulnerabilities are not detected [13, p. 204].

will represent a significant risk to the application and the data it processes. Of these, a subset will be accessible to, and potentially triggered by, input.

Implementation errors that satisfy all of following criteria may be considered *implementation vulnerabilities*:

- They must allow an attacker to modify the functioning of the application in such a way as to impact upon the confidentiality, availability or integrity of the data it processes or undermine any security requirements that have been specified.
- This modification must be achievable by passing input[3] to the application, since a vulnerability that is not reachable is not exploitable.

Sutton et. al provide the following example that demonstrates the importance of reachability in determining ‘exploitability’ [46, p. 4].

```
#include<string.h>

int main (int argc, char **argv)

{

        char buffer[10];

        strcpy(buffer, "test");

}
```

Figure 2.1: A non-vulnerable routine that calls the `strcpy` function [46, p. 4].

Figure 2.1 shows a simple routine where a character array called `buffer` is declared and the characters “test” are copied into is using the `strcpy` function. `strcpy` is, of course, infamous for its insecurity: if the source data is larger than the destination array, `strcpy` will write the source data beyond the destination array boundary and

[3]Note that the term input is used here in the widest possible sense, to extend to the entire application attack surface.

into adjacent memory locations: a well understood security vulnerability termed a buffer overflow. However, the routine shown in figure 2.1 is not vulnerable because the argument is passed to the vulnerable `strcpy` from within the routine, and nowhere else: the vulnerable function is not reachable from input therefore it is not exploitable [46, p. 5].

Consider the routine shown in figure 2.2. Here, the pointer `argv` collects data from the command line passing it as an argument to `strcpy`. `strcpy` will copy whatever data is passed to it from the command line into the `buffer` array. If the argument passed from the command line is longer than 10 characters, it will exceed the memory space allocated for the `buffer` array on the stack and overwrite adjacent data objects. The routine is exploitable because the vulnerable function `strcpy` is reachable via input [46, p. 5].

```
#include<string.h>

int main(int argc, char **argv)

{

        char buffer[10];

        strcpy(buffer, argv[1]);

}
```

Figure 2.2: A vulnerable routine that calls the `strcpy` function [46, p. 4].

Non-exploitable vulnerabilities are of considerably less concern than exploitable vulnerabilities, hence reachability is a critical factor. Two key points that stem from the importance of reachability are:

1. Fuzzing, unlike other methods for vulnerability discovery, will usually only trigger reachable defects as it is based on supplying malformed input to a target application.

2. A software vulnerability that is not reachable via input is effectively not exploitable.[4]

Given that it is highly likely that implementation vulnerabilities *will* be present in an application, the ability to block access to such vulnerabilities by differentiating between valid and invalid input, termed *input validation*, has considerable influence on the security of an application.

2.3 The Need for Input Validation

An Analogy

> *A restaurant. A customer orders a meal and passes the waiter a note for the chef. The note contains malicious instructions. The waiter passes the order and the note to the chef, who promptly sets fire to the kitchen.*
>
> *The waiter learns from the experience and refuses to pass notes to the chef any more. His approach is that no notes will be accepted. The next day, a customer orders a club sandwich and passes the waiter a sealed letter for the chef. The letter doesn't have the same characteristics as a note, so the waiter passes it to the chef. The chef sets fire to the kitchen again.*
>
> *The waiter updates his approach: no notes, no letters. The next day a customer says he has a telegram for the chef, and the waiter takes it to the chef. The chef says, "I'm sick of this. From now on, customers can only select items from the menu. If it's not on the menu don't bring it to me."*

Here, the customer is the user; the waiter represents the application interface: the component of an application that is responsible for receiving input from users, and the chef represents the back-end processing part of the application.

Two key points that labour the above analogy are:

[4]This approach depends on the effectiveness of the mechanism that prevents access. Furthermore, the absence of a risk is always preferable to a mitigated risk.

1. The waiter is unable to differentiate data (i.e. "soup de jour"), from instructions ("set fire to the kitchen").

2. The chef trusts the waiter completely and will carry out whatever instructions the waiter passes to him.

The above points are laboured because of differences between humans and computers. Humans are able to differentiate between instructions and data, and humans generally have more 'fine grained' and adaptable trust relationships than computers.

2.4 Differentiation Between Instructions and Data

Computers, by default, are unable to differentiate data from instructions. As a result, a class of *command* or *code injection* vulnerabilities exist that permit an attacker to pass instructions to an application that 'expects' to receive data, causing the application to execute attacker-supplied instructions within the security context of the application. These are both explored in detail later in this chapter.

2.5 Escalation of Privilege

If the application is running at a higher privilege than the attacker, then a successful code or command injection attack results in an *escalation of privilege*, where the attacker is able to run instructions of their choosing at a privilege level greater than their own.[5]

2.6 Remote Code Execution

If an injection vulnerability is remotely exploitable (via a network connection), then *remote code execution* may be possible, and the potential for an internet worm such as the SQL 'slammer'[6], code red[7] or nimda[8] worms arises.

[5] Note that the attacker has effectively extended the functionality of the application beyond that intended by the designers or specified by the users.

[6] http://www.cert.org/advisories/CA-2003-04.html

[7] http://www.cert.org/advisories/CA-2001-19.html

[8] http://www.cert.org/advisories/CA-2001-26.html

2.7 Trust Relationships

The second laboured point in our analogy arises from another difference between humans and computers: humans (usually) have free will, while computers are bound to execute instructions. Humans will usually reject instructions that might threaten their well-being in all but the most extreme situations. In contrast, components of an application often 'trust' each other absolutely. Input validation may be performed at a trust boundary at the point of input to the application, but once past that check, it may not necessarily be performed when data is passed between component parts of an application.

Returning to the inability of computers to differentiate between instructions and data, an attacker merely needs to satisfy grammar requirements of the execution environment in order to submit instructions and control the flow of execution (discounting for a moment the effect of input validation). Peter Winter-Smith and Chris Anley made this point in a presentation given as part of a Security Seminar entitled *An overview of vulnerability research and exploitation* for the Security Group at Cambridge University.

> *"From a certain perspective, [buffer overruns, SQL injection, command injection] are all the same bug. Data in grammar A is interpreted in grammar B, e.g. a username becomes SQL, some string data becomes stack or heap. [...] Much of what we do relies on our understanding of these underlying grammars and subsequent ability to create valid phrases in B that work in A."* [49]

In other words, a deep understanding of the underlying programming language and its syntactical and lexical rules may be used to craft input that may modify the functionality of an application in a manner that a developer or designer without the same level of understanding of the underlying technology may not foresee. However, such modification is only possible if the application permits users to pass input to it that satisfies the grammar rules of the underlying technology [49].

Two common approaches to employing underlying grammar rules to inject instructions in the place of data are *command* and *code injection.*

2.8 Command Injection

Command injection involves the use of special characters called *command delimiters* to subvert software that generates requests based on user input. Hogland and McGraw provide an example of command injection, reproduced here in its entirety, where a routine intended to display a log file executes a command string that is dynamically generated at run time by inserting a user supplied value (represented by the "`FILENAME`" place holder) into a string [20, p. 172]. The string is shown prior to insertion of user data below:

```
exec( "cat data_log_FILENAME.dat");
```

If the user-supplied data comprises of a command delimiter followed by one or more commands, such as[9]:

```
; rm -rf /; cat temp
```

then the dynamically created request becomes:

```
exec( "cat data_log_; rm -rf /; cat temp.dat");
```

The request now consists of three commands, none of which were envisioned by the system designer. The attacker has realised malicious functionality: the commands will execute within the security context of the vulnerable process, attempting to display a file called `data_log_`, deleting all of the files that the process is permitted to delete, and attempting to display the contents of `temp.dat`.

In order to trigger software errors, a fuzzer may employ a library of 'known bad' strings. For each of the vulnerability types discussed in this chapter I will provide a brief example of how a fuzzer heuristic might trigger that vulnerability. Detailed examples of fuzzer heuristics can be found in Appendix 2, *The Sulley Fuzzing Framework Library of Fuzz Strings*.

A fuzzer is able to trigger *command injection* defects by inserting commonly used command delimiters such as ';' and '\n'.

[9]It is unlikely that any commercial operating system would permit users to assign a name to a file that includes delimiters such as ';' and '/'. However, the target routine in this example is part of a vulnerable Common Gateway Interface (CGI) program, not the operating system, and the request is passed to it in the form of a modified URL.

2.9 Code Injection

Code injection is similar to command injection, but works at a lower level: the object or machine code level. Code injection is usually a two-stage process where instructions (in the form of byte code) are injected into the target process memory space, and then execution flow is redirected to cause the injected instructions to be executed. Injected byte code (sometimes termed *shell code*, since it often comprises the instructions required to launch an interactive command shell), must conform to grammar requirements of the interface[10], as well as satisfying the programming rules of the target platform.[11] This makes shell code development non-trivial since it must satisfy many different constraints.

Redirection of execution is usually achieved via *pointer tampering* where a pointer to a memory location holding the next instruction to be executed is overwritten by an attacker supplied value. Overwriting the instruction pointer is, of course, not normally permitted but can be achieved as the result of some form of memory corruption such as buffer overruns, heap corruption, format string defects and integer overflows.

Fuzzing is able to trigger code injection vulnerabilities by causing illegal (or at least, unforeseen) memory read and write operations (termed access violations) via buffer overruns, heap corruption, format string defects and integer overflows.

2.10 Buffer Overflows

Regions of memory assigned to hold input data are often statically allocated on the stack. If too much data is passed into one of these regions (termed *buffers*), then adjacent regions may be overwritten. Attackers have made use of this to overwrite memory values that are not normally accessible, such as the Instruction Pointer, which points to the memory location holding the next instruction to be executed. Overwriting the instruction pointer is one method for redirecting program execution flow.

[10]For example, no null bytes can be included since these indicate the end of a character string and cause the shell code to be prematurely terminated. As a result, where a register must be set to zero (a common requirement), instructions to cause the register to be Exclusive Or-ed with itself are used instead of using the standard approach of moving the value zero into the register.

[11]For example, injected instructions must executable on the processor running the target application.

Fuzzers employ many techniques to trigger buffer overflows, of which the most obvious are long strings. However, there are other techniques which may also be employed such as integer overflows, signedness issues, and string expansion.

2.11 Integer Overflows

Integers are data types assigned to represent numerical values. Integers are assigned memory statically at the point of declaration. The amount of memory assigned to hold an integer depends upon the host hardware architecture and Operating System. 32-bit systems are currently common, though we are moving towards widespread adoption of 64-bit systems. If we assume a 32-bit system, we would expect to see 32-bit integers, which could hold a range of 2^{32}, or 4,294,967,296 values. Hexadecimal numbering systems are often used to describe large binary values in order to reduce their printable size. A hexadecimal value can be used to describe any value held in four bits; eight hexadecimal values can describe a 32-bit value. When binary values are represented using hexadecimal values, the convention is to use the prefix '0x', in order to differentiate hexadecimal from decimal or other values. For example, 0x22 must be differentiated from 22 to avoid confusion.

An integer can hold a bounded range of values. Once a numerical value reaches the upper bound value of an integer, if it is incremented, the integer will 'wrap around' resetting the register value, a phenomenon termed *integer overflow*. This manifests as a difference between normal numbers and integer-represented values: normal numbers can increment infinitely, but integer-represented values will increment until they reach the integer bound value and will then reset to the integer base value, usually zero.

Vulnerabilities may occur when integers are errantly trusted to determine memory allocation values. Integers are also commonly used to perform bounds checking when copying data from one buffer to another so as to prevent buffer overflows. When a compare condition on two integers is used to determine whether a copy operation is performed or not, the wrapping behaviour of integers may be abused. For example, consider the following bounds checking routine pseudo code, based on a code sample offered by Sutton et al. [45, p. 175]:

IF x+1 is greater than y, THEN don't copy x into y
ELSE copy x into y
ENDIF

Here, an attacker could set x to the value 0xffffffff. Since 0xffffffff + 1 will wrap around to 0, the conditional check will allow x to be copied into y regardless of the size of y in the specific case that x = 0xffffffff, leading to a potential buffer overflow [45, p. 175].

Fuzzers often employ boundary values such as 0xffffffff in order to trigger integer overflows. The Spike fuzzer creation kit employs the following integer values (amongst others), probably because these have been found to be problematic in the past.

0x7f000000, 0x7effffff, 65535, 65534, 65536, 536870912 [3, Lines 2,079-2,084], and also: 0xfffffff, f0xffffff, 268435455, 1, 0, -1, -268435455, 4294967295, -4294967295, 4294967294, -20, 536870912 [3, Lines 2,217-2,229].

2.12 Signedness Issues

Integers may be signed or unsigned. The former data type can hold positive and negative numerical values, while the latter holds only positive values. Signed integers use the 'twos complement' format to represent positive and negative values that can be simply summed. One of the features of twos compliment values is that small decimal values are represented by large binary values, for example decimal '-1' is represented by the signed integer '0xffffffff' in a 32 bit integer environment.

In order to trigger signedness issues a fuzzer could employ 'fencepost' values such as 0xffffffff, 0xffffffff/2, 0xffffffff/3, 0xffffffff/4 and so on. The divided values might be multiplied to trigger an overflow. More importantly, perhaps, are near border cases such as 0x1, 0x2, 0x3, and 0xffffffff-1, 0xffffffff-2, 0xffffffff-3, since these are likely to trigger integer wrapping. Combining the two, we might include (0xffffffff/2)-1, (0xffffffff/2)-2, (0xffffffff/2)-3, (0xffffffff/2)+1, (0xffffffff/2)+2, (0xffffffff/2)+3, and so on, in order to trigger more integer signedness issues.

2.13 String Expansion

The term *string* is used to describe an array of *char* data types which are used to hold characters. Strings are a very common input data type, and improper string handling has led to many security vulnerabilities. The encoding and decoding or

translation of characters within a string can be problematic when some characters are treated differently to others. An example are the characters 0xfe and 0xff, which are expanded to four characters under the UTF-16 encoding protocol [45, p. 85].[12] If anomalous behaviour such as this is not accounted for, string size calculations may fall out of line with actual string sizes resulting in overflows.

Since delimiter characters may be treated differently to non-delimiters, fuzzers may use long strings of known delimiter characters in addition to characters known to be subject to expansion.

2.14 Format Strings

The `printf` family of functions are part of the standard C library and are able to dynamically and flexibly create strings at runtime based on a format string which consists of some text, a format specifier (such as %d, %u, %s, %x or %n) and one or more arguments [13, p. 206]. In normal operation, the arguments are retrieved (POPed) from the stack, the format specifier defines how the arguments are to be formatted, and the formatted arguments are appended to the (optional) text, to construct an output string which may be output to the screen, to a file, or some other output.

Anyone who has coded in C will be familiar with the following:

```
printf ("Result = %d\n", answer);
```

Here, `"Result = %d\n",` is the format string (which consists of some text `Result =` and a format specifier `%d\n`), and `answer` is the argument. The value of the `answer` argument will have been pushed onto the stack within a stack frame prior to the `printf` function being called. When called, `printf` would POP the binary value held in the argument on the stack, format it as a decimal due to the `%d` format specifier, construct a string containing the characters `Result =`, and then append the decimal formatted value of the `answer` argument, say, `23`, to the end of the string.

[12] According to RFC 2781, UTF-16 *"is one of the standard ways of encoding Unicode character data"*. Put simply, UTF-16 can describe a large number of commonly used characters using two octets, and a very large number of less common 'special' characters using four octets.

Format string vulnerabilities occur when additional, spurious format specifiers are allowed to pass from input to one of the `printf` family of functions and influence the manner in which the affected function behaves.

`printf` is one of a number of C and C++ functions that does not have a fixed number of arguments, but determines the number of format specifiers in the format string and POPs enough arguments from the stack to satisfy each format specifier [13, p. 203]. Via the insertion of additional format specifiers, a format string attack can modify an existing format string to cause more arguments to be POPed from the stack passed into the output string than were defined when the function was called [13, p. 203]. The effect of this is akin to a buffer overflow in that it makes it possible to access memory locations outside of the called function's stack frame.

Where a vulnerable instance of the `printf` family of functions both receives input and creates accessible output, an attacker may be able to circumvent memory access controls using format string attacks to read from, and write to, the process stack and memory arbitrarily.

By inserting %x format specifiers into a vulnerable instance of the `printf` family of functions, an attacker can cause one or many values to be read from the stack and output to, say, the screen, or worse, may be able to write to memory [13, p. 202].

The %s format specifier acts as a pointer to a character array or *string.* By inserting a %s format specifier and providing no corresponding pointer address argument, an attacker may be able to trigger a failure causing a denial of service. By inserting a %s format specifier and providing a corresponding pointer address argument, an attacker may be able to read the value of the memory location at the provided address [13, p. 215].

Most concerning of all is the %n format specifier. This was created to determine and output the length of a formatted output string, and write this value (in the form of an integer) to an address location pointer provided in the form of an argument [13, p. 218]. Foster and Liu describe the nature of the %n format specifier as follows:

> *"When the %n token is encountered during printf processing, the number (as an integer data type) of characters that make up the formatted output string up to this point is written to the address argument corresponding to that format specifier."* [13, p. 218]

The capability to write arbitrary values to arbitrary memory locations within a process memory space by supplying malicious input represents a significant risk to an application, since execution could be redirected by overwriting the Instruction Pointer to attacker-supplied machine code, or to local library functions in order to achieve arbitrary code execution [13, p. 207].

The best way to prevent format string attacks is to sanitize data input by employing an input validation routine that prevents any format specifiers from being passed to the application for processing. Failing that, or preferably in addition, it is wise to explicitly specify the expected data format, as shown in an example taken from Foster and Liu [13, p. 212].

The below call to `printf` specifies that data received from the `buf` argument should be formatted as a character string. This will cause any inserted format specifiers to be harmlessly printed out as characters [13, p. 212].

```
printf ("%s" buf);
```

The below call to `printf` will process format specifiers included in the `buf` argument, allowing an attacker to modify the format string [13, p. 212].

```
printf (buf);
```

Fuzzer heuristics should certainly include repeated strings of some or all of the possible format specifiers (%l, %d, %u, %x, %s, %p, %n). However, Sutton et al. state that %n specifier is *"the key to exploiting format string vulnerabilities"* [46, p. 85], since it is most likely to trigger a detectable failure due to the fact it can cause illegal memory *write* operations, while other format specifiers may trigger illegal *read* operations.

2.15 Heap Corruption

Each process has its own heap, just as it has its own stack area of memory; both are subject to buffer overflows. Unlike the stack, heap memory is persistent between functions, and memory allocated must be explicitly freed when no longer needed. Like stack buffer overflows, heap overflows can be used to overwrite adjacent memory

locations; unfortunately, allocated regions of memory are usually located adjacently to one another. However, a heap overflow may not be noticed until the overwritten region is accessed by the application [13, p. 162].

The key difference between heap and stack memory storage is that stack memory structures, once allocated, are static. In contrast, heap memory structures can be dynamically resized, by manipulation of pointers which define the boundaries of heap memory structures [13, p. 163]. As with the stack, if a vulnerable function is used to copy data from a source buffer into a static heap-based buffer, and the length of the source data is greater than the buffer bounds, the vulnerable string function will overwrite the destination buffer and may overwrite an adjacent memory block [13, p. 164].

Wherever data within a process memory space is overwritable, one only has to locate a pointer to executable code to take control of execution. Since the heap contains many such pointers, [13, p. 167] and stack overflows are commonly prevented now by methods such as non-executable stack compiler switches and stack overflow detection, heap exploits may now be more common that stack overflows [13, p. 162].

Triggering heap overflows is not simply a matter of inserting special characters into input. Heap overflows can be triggered by malformed input that causes heap memory allocation errors, particularly in arithmetic operations used to determine required buffer lengths. The following is a description of a heap-based vulnerability (MS08-021) discovered by iDefence in the Microsoft Windows Graphics Rendering Engine:

> *"The vulnerability occurs when parsing a header structure in an EMF file that describes a bitmap contained in the file. Several values from this header are used in an arithmetic operation that calculates the number of bytes to allocate for a heap buffer. This calculation can overflow, which results in an undersized heap buffer being allocated. This buffer is then overflowed with data from the file."* [13]

It appears that the above vulnerability is the product of an integer overflow which leads to a heap overflow, so it might be triggerable by the use of random signed integers, random unsigned integers, and fencepost values. However, this blind approach might have very low odds of succeeding. We will discuss how fuzzing can be used to

[13] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=681

intelligently trigger such vulnerabilities by manipulation of common transfer syntaxes in Chapter 10, *Protocol Analysis Fuzzing*.

2.16 Chapter Summary

We have seen how vulnerabilities may stem from software defects, and that they can be grouped based on the different phases of the development/deployment life cycle, and we have seen that fuzz testing is mainly concerned with defects occurring at the implementation stage.

We have examined some of the causes of vulnerabilities, and explored each of the main classes of vulnerability that may be discovered via fuzzing. In each case we have briefly touched on the actual means that a fuzzer might use to trigger such defects. In order to ensure that this chapter focussed on vulnerabilities rather than fuzzer heuristics, more detailed examples of fuzzer heuristics, (specifically those aimed at fuzzing string data types) have been moved to Appendix B, *The Sulley fuzzing framework library of fuzz strings* for examples of fuzzer malicious string generation.

In the next chapter, we will examine the various types of security testing methodologies and place fuzzing in the wider context of security testing.

CHAPTER

THREE

SOFTWARE SECURITY TESTING

3.1 Software Testing

> *"The greatest of faults, I should say, is to be conscious of none."*

Thomas Carlyle, 1840

In the *Certified Tester Foundation Level Syllabus*, The International Software Testing Qualifications Board have defined a number of general software testing principles, all seven of which can be applied to software security testing. The first three principles are of particular relevance.

> *"Principle 1: Testing shows presence of defects*
> *Testing can show that defects are present, but cannot prove that there are no defects. Testing reduces the probability of undiscovered defects remaining in the software but, even if no defects are found, it is not a proof of correctness."* [31, p. 14]

Security testing can never prove the absence of security vulnerabilities, it can only reduce the number of undiscovered defects. There are many forms of security testing: none can offer anything more than a 'snapshot'; a security assessment of the application at the time of testing, based on, and limited by, the tools and knowledge available at the time.

> *"Principle 2: Exhaustive testing is impossible*
> *Testing everything (all combinations of inputs and preconditions) is not feasible except for trivial cases. Instead of exhaustive testing, risk analysis and priorities should be used to focus testing efforts."* [31, p. 14]

We will see that exhaustive fuzz testing is largely infeasible for all but the most trivial of applications.

> *"Principle 3: Early testing*
> *Testing activities should start as early as possible in the software or system development life cycle, and should be focused on defined objectives."* [31, p. 14]

There are strong economic and security arguments for testing for, and rectifying, defects as early as possible in the development life cycle.

3.2 Software Security Testing

At the highest level, software functional testing involves determining whether an application satisfies the requirements specified in the functional specification by testing positive hypothesis such as *"the product can export files in .pdf format"*, or use cases such as *"the user is able to alter the booking date after a booking is made"*.

In contrast, Software *security* testing is concerned with determining the total functionality of an application, as shown in figure 3.1, which includes any functionality that may be realised by a malicious attack, causing the application to function in a manner not specified by the system designer.[1]

Since security requirements tend to be negative [24, p. 35], software security testing generally involves testing negative hypothesis - i.e. *"the product does not allow unauthorised users to access the administration settings"*. This has led security testers to search for exceptions to security requirements. Such exceptions are, of course, *vulnerabilities*, opportunities to realise malicious functionality.

The range of methodologies for identifying vulnerabilities can be separated into one of two main classes: *white box*, also known as *structural* testing, or *black box* also known as *functional* testing.

[1] An example is SQL injection, where meta characters are employed in order to modify a database query dynamically generated partly based on input data.

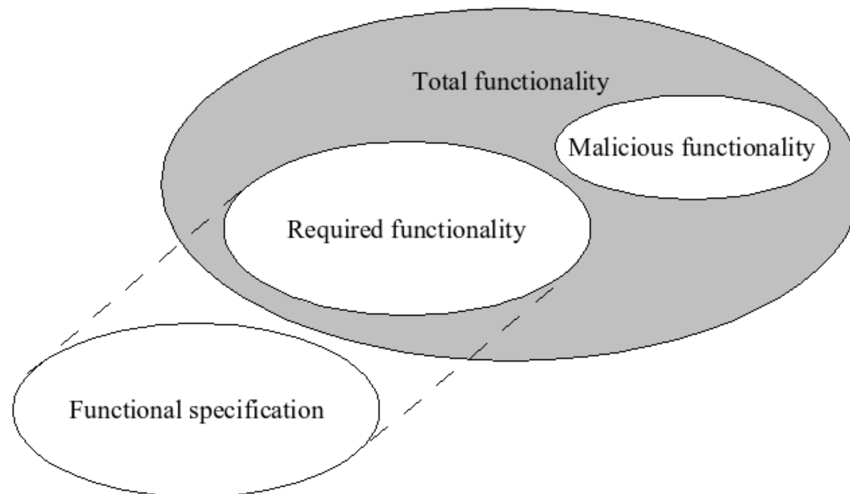


Figure 3.1: The total functionality of a software application may be greater than that specified in requirements.

3.3 Structural, 'White Box' Testing

White box testing is performed when the tester obtains and employs knowledge of the internals of the target application. Typically this means that the tester has access to the source code[2], and possibly also the design documentation and members of the application development team.

White box testing can be divided into two approaches: (static) structural *analysis* and (dynamic) structural *testing*.

3.3.1 Static Structural Analysis

Static analysis may be performed on the source code or the binary (object) code, and involves searching for vulnerabilities via analysis of the code itself, not the executed application. This is usually achieved by pattern matching against a library of 'known bad' code sections.

[2]Structural methods can be applied to the object code of a component [7].

It is invariably easier to analyse the source code than the object code, since higher-level languages are closer to human languages than the object code they are compiled and assembled into. Additionally, source code should feature comments that can assist in understanding functionality. If the source code is available this will usually be preferred for analysis. However, the absence of source code does not preclude analysis; it merely makes it more difficult.

Source Code Analysis

Source code auditing has been an effective method of vulnerability discovery for many years. As the complexity and scale of software products has increased, manual analysis of source code has been replaced with automated tools.

The earliest source code analysis tools were little more than pattern matching utilities combined with a library of 'known bad' strings, which tested for calls to vulnerable functions such as *strcpy* and *sprintf*. These early tools generated many false positives, since they were intended to identify any potential areas of concern, so that these could be reviewed by a human auditor.

Access to the source code is an obvious requirement for source code analysis, which precludes source code analysis for many parties including end users, corporate clients, professional vulnerability researchers, developers who rely on the application as middleware, and of course, hackers. Access to the source code may be limited to those prepared to sign Non-Disclosure Agreements; this may deter security researchers and hackers alike.

Static source code analysis often lacks the contextual information provided by dynamic testing. For example, though static source code analysis may reveal that a value is disclosed to users, without contextual information about what that value actually *is*, it may be hard to determine if there is an associated security risk.

There may be differences between the source code that the tester is given and the final shipped executable: the software development process may not necessarily halt while testing is performed; last-minute changes may be made to the source code; mass distribution processes may alter the codebase, and so on. This is a form of 'TOCTOU' (Time Of Check, Time Of Use) issue, where what is tested is not what is used. This could lead to both false positives (i.e. a bug is found in the tested code

that is fixed in the distributed code) and false negatives (i.e. no bugs are found in the tested code, but there are bugs present in the distributed code).

Access to source code does not guarantee that vulnerabilities will be found. In February 2004, significant sections of the source code of Windows NT 4.0 and Windows 2000 operating systems were obtained by, and distributed between, a number of private parties. At the time there were fears that numerous vulnerabilities would result, yet only a handful have since been attributed to this leak [46, p. 5].

The level of complexity and sheer scale of software products mean that source code analysis usually means a reliance on the automated tools: a human simply cannot read through the source code of most applications. Modern source code auditing tools, whilst undoubtedly powerful, require tuning to the environment in order to get the most out of them [48].

After presenting a number of (pre-disclosed) vulnerabilities discovered using a fuzzer, Thiel went on to state that:

> *"At least one of these vendors was actually using a commercial static analysis tool. It missed all of the bugs found with Fuzzbox [a fuzzer created by Thiel]"* [47, Slide 32]

While the above indicates that fuzzing may discover defects not found via source code auditing, I believe there are also many cases where source code auditing could find defects which would not be discoverable via fuzzing. All forms of software analysis are of value, each provides a different insight into an application. The best approach might be to apply them all, where possible.

The output of source code auditing may mean that further effort is required to develop *proof of concept* code to convince developers and vendors of the existence of a security vulnerability. Vulnerabilities discovered via fuzzing generally consist of a specific test case instance coupled with an associated crash report, a crude yet convincing form of proof of concept code [48].

Binary Analysis

In *binary analysis* (also termed *binary auditing*, or, *Reverse Code Engineering* (RCE)) the binary executable file is not executed but is 'disassembled' for human interpretation or analysed by an automated scanning application.

Disassembly describes the process where compiled object code is converted back to assembly operation codes and displayed on a computer screen. This information can be interpreted by a human auditor in order to understand the inner functionality of an application and identify vulnerabilities by revealing programming flaws.

Since the source code is not required, anyone who has access to the binary executable file can disassemble it for analysis, (the exception being cases where obfuscation or anti-disassembly techniques have been applied; such techniques are common in the areas of malware development and intellectual property protection applications). However, disassembly analysis is considerably harder than source code analysis since assembly is a lower-level programming language, increasing the level of complexity.

A number of automated scanning tools (some are stand-alone, some are plug-ins for a commercial reverse code engineering tool called *IDA Pro*[3]) have been developed to assist with the process of Reverse Code Engineering. These include: *Logiscan*[4], *BugScam*[5], *Inspector*[6], *SecurityReview*[7] and *BinAudit*[8].

Note that many of the above products are not publicly available. This may be due to the risk that they represent in that they could be used by malicious parties to identify software vulnerabilities, and it may also be due to the fact that some are commercial products, or components of commercial products.

Binary auditing, whether automated or manual is an extremely powerful method for identifying vulnerabilities, since it offers all of the benefits of source code analysis, yet the source code is not required. However, it requires the highest level of expertise of all software security testing methodologies.

Binary analysis may be illegal in some circumstances, and has been associated with software ‘cracking’, where criminals circumvent software protection controls. Many software product End User Licence Agreements expressly forbid any form of disassembly.

---

[3] http://www.hex-rays.com/idapro/
[4] http://www.logiclibrary.com/about_us/
[5] http://sourceforge.net/projects/bugscam
[6] http://www.hbgary.com/
[7] http://www.veracode.com/solutions
[8] http://www.zynamics.com/products.html

3.3.2 Dynamic Structural Testing

Dynamic structural testing involves 'looking into the box', i.e. analysis of the target internals in order to, in the case of security testing, discover vulnerabilities.

An example of dynamic structural testing is API[9] call hooking, where API calls made by the application are intercepted and recorded. This could reveal whether an application calls a vulnerable function such as strcpy.

Another example of dynamic structural testing (there are many), termed *red pointing* is described by Hogland and McGraw. Here, the source code is analysed and obvious areas of vulnerability (again, usually vulnerable API calls such as strcpy) are identified, and their location is recorded. The application is launched and manipulated with the aim of reaching the vulnerable area. If the tester can reach the target location (usually detected via attaching a debugger and setting a breakpoint on the target location), via manual manipulation of input, and input data is processed by the vulnerable function, then a vulnerability may have been identified [20, p. 243].

3.4 Functional, 'Black Box' Testing

Black box testing means not using any information about the inner workings of the target application. This means that access to the source code, the design documents and the development team are not required.

Conventional black box or functional testing focuses on testing whether the functionality specified is present in the application, by executing the application, feeding it the required input and determining if the output satisfies the functional specification [24, 24].

Software security black box testing involves testing for the presence of security vulnerabilities that might be used to realise malicious functionality, (see figure 3.1) without any knowledge or understanding of the internals of the target application. This may also be termed *fault injection*, since the objective is to induce fault states in the target application or its host by injecting malformed input.

Fault injection testing, also known a negative testing, involves passing unexpected input to the executed application and monitoring system and application health, such

[9] API stands for Application Programming Interface.

that specific instances of input that cause application or system failure are identified. Fault injection is closely related to performance testing: both are less interested in pure functional specification testing, and relate more to the quality of the software implementation.

Consider a sofa. A designer may check that the finished product satisfies her original design (i.e. by testing against functional requirements), but the sofa cannot be sold legally without safety testing (i.e. negative testing) to ensure it is, for example, flame retardant.

One of the difficulties of conventional functional testing is that in order to test whether required functionality is present, a test case must include a clear definition of the expected output, based on the defined input. This means a tester can clearly determine whether an application satisfies that particular test case. For fault injection testing, the scope is limited to identifying input that causes the application to fail; we do not need to define expected output, we merely need to monitor the application (and host operating system) health [24, p. 35]. This approach greatly simplifies testing, but also prevents the detection of defects that do not result in an application failure state.

There are test tools that can detect potentially vulnerable states that do not result in application failure. Application Verifier [21, p. 159] is an example of a test tool that can detect a range of indicators of a vulnerable state. Such tools may monitor CPU usage, memory allocation, thread processing and other aspects of an application and it's host operating system, and would be better placed in the field of dynamic structural testing, where analysis of the internals of the system under test are employed. There are no hard boundary points between test methodologies. However, the further we look 'into the box', the further we move away from black box testing.

Fuzzing is a particular form of injection testing where the emphasis is on automation. Other methods of injection testing include the construction of *malicious clients*, facilitating manual parameter manipulation [4, p. 1], or manual malicious data entry such as entering random or known bad characters into an application [46, p. 10].

3.5 Chapter Summary

In this chapter, we started out with the basics of software testing, we then focussed upon software security testing, examining the various software security methodologies. It is important that these methodologies are seen as a 'grab bag' of non-exclusive tools and approaches. A good software security tester will employ whatever methods suit the task at hand, and some approaches (sometimes the best approaches) defy methodology boundaries. As in the last chapter, we again placed fuzzing in context, contrasting its characteristics against alternative approaches.

This concludes the 'wider view' section of the report, and from the next chapter onward we will focus exclusively on fuzzing, beginning with an examination of its origin and a brief overview of the subject.

CHAPTER

FOUR

FUZZING – ORIGINS AND OVERVIEW

4.1 The Origins of Fuzzing

The 'discovery' of fuzzing as a means to test software reliability is captured in a paper produced in 1989 by Miller, Fredriksen and So [34].

It is unlikely that Miller et al were the first to employ random generation testing in the field of software testing. However, Miller et al. appear to have produced the first documented example of a working fuzzer in the form of a number of scripts and two tools called and *fuzz* and *ptyjig*.

Fuzzing was 'discovered' almost accidentally, when one of the authors of the above paper experienced electro-magnetic interference when using a computer terminal during a heavy storm. This caused random characters to be inserted onto the command line as the user typed, which caused a number of applications to crash. The failure of many applications to robustly handle this randomly corrupted input led Miller, et al to devise a formal testing regime, and they developed two tools *fuzz* and *ptyjig*, specifically to test application robustness to random input.

The results of the testing were that of 88 different UNIX applications tested, 25 to 33% crashed when fed input from their fuzzing tools. Of the two tools produced by Miller et al, *fuzz* is of greater significance, since *ptyjig* is merely used to interface the output of *fuzz* with applications that required input to be in the form of a terminal device.

fuzz is, essentially, a random character string generator. It allows users to define an upper limit to the amount of random characters to be generated. The output of *fuzz* can be limited to printable characters only, or extended to include control characters and printable characters. A seed can be specified for the random number generator, allowing tests to be repeated. Finally, *fuzz* can write its output to a file as well as to the target application, acting as a record of generated output.

Miller et al employed scripting to automate testing as much as possible. After an application terminates, a check is performed to see if a core dump has been generated. If so, the core dump and the input that caused it are saved.

By creating *fuzz* to satisfy their testing requirements, Miller et al also inadvertently defined a general model of a practical fuzzer: i.e. the elements it should comprise, the functionality it should offer, etc. While fuzzing has undoubtedly moved forward in the last twenty years, the basic model of a fuzzer has remained the same.

4.2 A Basic Model of a Fuzzer

The term *fuzzer* may be used to describe a multitude of tools, scripts, applications, and frameworks. From dedicated, one-off Perl scripts, to all-encompassing modular frameworks, the range of fuzzers can be bewildering to the uninitiated.

However, all fuzzers share a similar set of features, namely:

- data generation (creating data to be passed to the target);
- data transmission (getting the data to the target);
- target monitoring and logging (observing and recording the reaction of the target), and;
- automation (reducing, as much as possible, the amount of direct user-interaction required to carry out the testing regime).

In fact, the last two features could be considered optional or might be implemented externally to the fuzzer; a stand-alone debugger application might be employed to monitor the target.

By treating the above features as high-level requirements, we can outline a basic model of a fuzzer in figure 4.1.

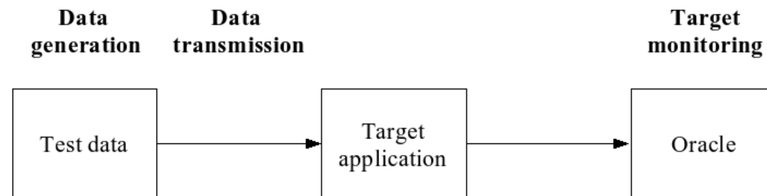


Figure 4.1: A basic model of a fuzzer.

4.3 Fuzzing Stages

However, there is more to fuzzing than the fuzzer itself - our basic model of a fuzzer fails to capture the fuzzing life cycle. Sutton has listed the stages of fuzzing as being [46, p. 27]:

1. Identify target
2. Identify inputs
3. Generate fuzzed data
4. Execute fuzzed data
5. Monitor for exceptions
6. Determine exploitability

What follows is a brief description of each of the stages listed above.

4.3.1 Target Identification

This stage is optional since the target may have already been selected. Attackers typically get to choose their targets while testers may not. Risk, impact and user base are the primary factors that influences target selection for those who get to choose, and resource deployment for those who don't.

Targets that present significant risk are:

1. Applications that receive input over a network - these have the potential to be remotely compromised, facilitating remote code execution, which creates the potential for an internet worm.
2. Applications that run at a higher privilege level than a user - these have the potential to allow an attacker to execute code at a privilege level higher than their own, known as privilege escalation.
3. Applications that process information of value - an attacker could circumvent controls and violate integrity, confidentiality or availability of valuable data
4. Applications that process personal information - an attacker could circumvent controls and violate integrity, confidentiality or availability of private data

Targets that combine two or more of the above are at particular risk. A service that runs with Windows SYSTEM-level privileges and also receives input from a network is a juicy target for an attacker. A large user base, i.e. a widely deployed application, or a default component of a commercial operating system represents an increased risk since the impact of a successful attack is increased by a large user population.

The Microsoft Security Response Centre Security Bulletin Severity Rating System defines four levels of threat that can be used to evaluate a vulnerability. The below ratings and their definitions are taken from Microsoft's website[1].

> Critical: A vulnerability whose exploitation could allow the propagation of an Internet worm without user action.
>
> Important: A vulnerability whose exploitation could result in compromise of the confidentiality, integrity, or availability of users' data, or of the integrity or availability of processing resources.
>
> Moderate: Exploitability is mitigated to a significant degree by factors such as default configuration, auditing, or difficulty of exploitation.
>
> Low: A vulnerability whose exploitation is extremely difficult, or whose impact is minimal.

[1] `http://www.microsoft.com/technet/security/bulletin/rating.mspx`

4.3.2 Input Identification

Input identification involves enumerating the *attack surface* of the target. Howard and Lipner define the attack surface of a software product as:

> *"the union of code, interfaces, services, and protocols available to all users, especially what is accessible by unauthenticated or remote users."* [21, p. 78]

This stage is important since a failure to exhaustively *enumerate* the attack surface will result in a failure to exhaustively *test* the attack surface, which in turn could result in deployment of an application with exposed vulnerabilities.[2]

Application input may take many forms, some remote (network traffic), some local (files, registry keys, environment variables, command line arguments, to name but a few). A range of fuzzer classes have evolved to cater for the range of input types. Sutton sets out input classes (and provides some example fuzzers) as follows [45, p. 9]:

1. Command line arguments
2. Environment variables (ShareFuzz)
3. Web applications (WebFuzz)
4. File formats (FileFuzz)
5. Network protocols (SPIKE)
6. Memory
7. COM objects (COMRaider)
8. Inter Process Communication

Network protocol, web application and COM object fuzzing are suited to the discovery of remote code execution vulnerabilities, while the rest generally lead to the discovery of local vulnerabilities[3].

[2]However, it may not be practical to fuzz all of the identified forms of input. It may be that there are no fuzzers already developed to fuzz that input type and the development of such a fuzzer would not be worth the required investment. While it is acceptable not to fuzz a given input type it is wise to identify any untested forms of input and ensure that alternative testing or mitigation strategies are applied to input vectors that fall out of scope of fuzz testing.

[3]Web browser fuzzing is an exception: it is a particular form of file fuzzing that can reveal code execution vulnerabilities in browsers [46, p. 41].

Some application inputs are obvious (a web server will likely receive network input in the form of HTTP via TCP over port 80), or are easily determined using tools provided with the host Operating System such as ipconfig, netstat, task manager in Windows systems. Others require specialist tools such as filemon[4], which reports every file access request made by an application.

4.3.3 Fuzz Test Data Generation

This is perhaps the most critical aspect of fuzz testing, and this area has developed considerably since Miller et al produced their early fuzzing tools.

The purpose of a fuzzer is to test for the existence of vulnerabilities that are accessible via input in software applications. Hence, a fuzzer must generate test data which should, to some degree, enumerate the target input space which can then be passed to the target application input.

Test data can either be generated in its entirety prior to testing[5], or more commonly, iteratively generated on demand at the commencement of each of a series of tests.

The entire range of test data generated for fuzzing a given target (referred to hereafter as the *test data*) comprises of multiple individual specific instances (referred to hereafter as *test case instances*). The general approach to fuzz testing is to iteratively supply test instances to the target and monitor the response.

Where, during testing, a test case is found to cause an application failure, the combination of a particular test case and information about the nature of the failure it caused represents a defect report. Defect reports may be thought of as the distilled output of fuzz testing and could be passed to developers to facilitate the process of addressing failures.

In order to determine how a fuzzer will generate test data, a set of rules is usually defined by the user. This is shown in figure 4.2.

There are a multitude of different approaches for generating test data, all of which fall into one of two categories: zero knowledge testing (comprising *random*, *brute force*

[4] `http://technet.microsoft.com/en-us/sysinternals/bb896642.aspx`

[5] This approach is often seen in file format fuzzing.

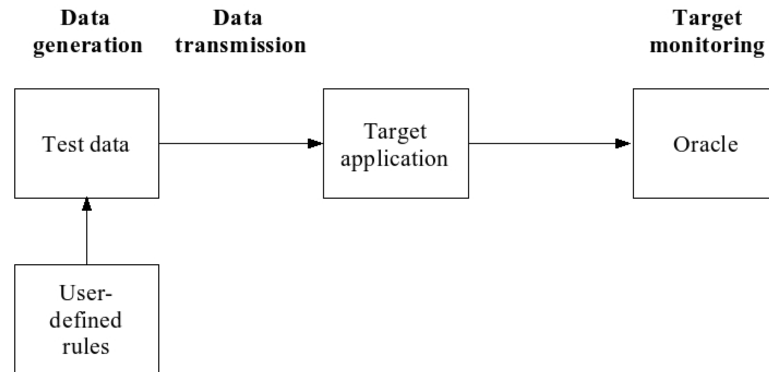


Figure 4.2: A basic model of a fuzzer including user-defined rules for data generation.

and *'blind' mutation* fuzzing) or analysis-based testing (termed *protocol* or *protocol implementation* testing).

Test data generation differs greatly across various fuzzers and its importance means that it will be covered in detail over three chapters: Chapter 5, *Random and Brute Force Fuzzing*, Chapter 6, *Data Mutation Fuzzing*, and Chapter 10, *Protocol Analysis Fuzzing.*

4.3.4 Fuzzed Data Execution

This stage also differs between fuzzers but is largely a function of the particular approach to automating the test process. This will not be covered any further than two of the Case Studies: Chapter 8, *Case Study 1 Blind Data Mutation File Fuzzing* and Chapter 11, *Case Study 3 Protocol Fuzzing a Vulnerable Web Server.*

4.3.5 Exception Monitoring

It is not sufficient to simply generate test data that triggers the manifestation of software defects: in order to discover vulnerabilities via fuzzing, one must have a

means for detecting them. This is achieved via an *oracle*, a generic term for a software component that monitors the target and reports a failure or exception. An oracle may take the form of a simple liveness check, which merely pings the target to ensure it is responsive, or it may be a debugger running on the target that can monitor for, and intercept, exceptions and collect detailed logging information.

This area will be explored in more detail in Chapter 7, *Exception Monitoring*.

4.3.6 Determining Exploitability

Once one or a number of software defects have been identified, there may be no further work to do other than to submit a list of these defects to a development team, in order that they can correct them. However, it may be that the tester is required to determine the risk that such bugs represent, and this usually requires an examination of whether defects are exploitable or not, and if so, what impact exploitation may mean for users. This is discussed further in Chapter 7, *Exception Monitoring*, and also in Chapter 9 *Case Study 2 – Using Fuzzer Output to Exploit a Software Fault*.

4.4 Who Might Use Fuzzing

Anyone who has access to an application can fuzz it. Access to the source code is not required. Compared to other vulnerability discovery methodologies, very little expertise is required (at least to identify basic defects). Additionally, implementation is comparatively fast - an experienced user of fuzzers can, in some cases, initiate fuzzing an application in a matter of minutes.

As a result of the comparatively low barrier to entry in terms of investment of time, understanding of the application and software in general, and access to the source code, a number of different parties may benefit from fuzzing.

Developers may employ fuzzing as part of a wider vulnerability discovery and resolution program throughout the development life-cycle. Software vendors such as Cisco, Microsoft, Juniper, AT&T, and Symantec all employ fuzzing as a matter of course [11, Slide 8].

End-users, Small to Medium sized Enterprises, and corporations might also employ fuzzing as a form of software quality assurance. For these parties, the inability to access source code and the efficient use of time and resources may be an attraction.

> *"One of the surprises of selling fuzzing products at Beyond Security, is who actually wants them. Banks, Telcos, large corporations."* [11, Slide 16]

For these customers, Evron states one of the reasons for fuzzing an application prior to purchasing licences is:

> *"Being able to better decide on the security and stability of products than look at their vulnerability history."* [11, Slide 17]

This highlights the fact that many corporate customers lack reliable sources of information regarding the security of a potential product. This suggests information asymmetry exists as mentioned in Chapter 1, *The Case for Fuzzing*, and that corporate customers are employing fuzzing to remedy this situation.

Attackers may also make use of fuzzing. Fuzzing offers many benefits to malicious parties, particularly those who are skilled in exploit development and interested in identifying injection vectors for malicious payloads. Such parties often do not have access to the target application's source code, and are interested in identifying vulnerabilities for the purpose of developing un-patched, undisclosed exploits. Note that fuzzing itself does not produce exploits, but can be used to reveal software defects which may be exploitable.

Exploit development consists of two primary activities: vulnerability discovery and payload generation. Payload generation is widely researched and information is generally shared openly on numerous websites, of which Milw0rm[6] and Metasploit[7] are two high-profile examples. Many payloads (such as shell code which will launch an interactive command shell and bind it to a listening network port) are interchangeable, and can be tweaked to suit the target application and the objectives of the attacker.

Specific information about implementation vulnerabilities is not generally shared, since this information is precious and may be passed solely to the vendor in order to

[6] `http://www.milw0rm.com`
[7] `http://www.metasploit.com`

provide them with an opportunity to address the vulnerability, or may be sold on the black market or to a reputable vulnerability intelligence group such as the iDefence Vulnerability Contribution Program (VCP)[8] [44].

4.5 The Legality of Fuzz Testing

In general, black box security testing is not illegal, since most anti-reverse engineering law is based on forbidding unwarranted examination of intellectual property, usually achieved via disassembly and reverse engineering of internal functioning. Since black box testing is merely concerned with input/output analysis, it might be argued that it does not break user licensing agreements, or intellectual property law since there is no attempt to understand the business logic of the application.

That said, since the objective of black box testing is to discover application failure states, some of which may be exploitable, it could also be argued that the legality of such testing depends on the motivation of the tester and the actions taken after vulnerabilities are discovered. In this interpretation, the action the tester takes after discovery of a vulnerability is critical to determining their legal position. There is a moral and legal imperative to act responsibly with information regarding vulnerabilities, and anyone undertaking any form of software security testing should be prepared to justify their actions or risk prosecution.

4.6 Chapter Summary

In this chapter we examined the origin of fuzzing, presented a basic model of a fuzzer identified some of the parties likely to employ fuzzing, and covered the legality of fuzz testing. The next chapter will focus upon the most basic forms of fuzz testing: random and brute force fuzzing.

---

[8]labs.idefense.com/vcp/

CHAPTER

FIVE

RANDOM AND BRUTE FORCE FUZZING

There are a three different zero knowledge approaches for generating test data:

- Random data generation;
- Data generated in a 'brute-force'[1] manner;
- Data mutation, i.e. capturing 'valid' input data that is passed to application in normal use, and mutating it, either in a random or 'brute-force' manner;

In order to understand and compare the various data generation methods, we will employ a trivial example (inspired by [36, p. 2]) to compare test data generation methods and illustrate the concept of *application input space*.

5.1 Application Input Space

The range of all of the possible permutations of input that an application can receive may be termed its *input space*. Consider a trivially simple application that receives input from four binary switches.

Figure 5.1 allows us to visualise the input space of our trivial application as a two dimensional grid.

[1] The term brute-force refers to the sequential generation of all possible combinations of a number of values.

Switch 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Switch 2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Switch 3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Switch 4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Figure 5.1: A visual representation of the input space of four binary switches [36, p. 2].

Imagine that we have been asked to test the trivial application's robustness. Our objective then, is to enumerate the input space with the aim of uncovering any test instances that cause application failure. Due to a software error, the trivial application will fail if the switches are set to the combined value of 1011, as shown in figure 5.2.

Switch 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Switch 2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Switch 3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Switch 4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Figure 5.2: A visual representation of the input space of four binary switches with an error state (1011) [36, p. 2].

Fuzzing aims to automatically generate and submit test data that will trigger input-related software defects. The challenge for the data generation aspect of a fuzzer is to generate test data that includes test instances that will trigger vulnerabilities present in the target application; in this case, the combination 1011. An analogy for this task is the game Battleships [36, p. 1], where two players first place ships

in a grid, and then take turns in trying to 'hit' their opponents ships by guessing co-ordinates that match the ships locations.

5.2 Random Data Generation

Fuzz testing with randomly generated data has been referred to as 'blind fuzzing' [4, p. 2], since neither knowledge of the target nor the data it is designed to process are required. This 'zero-knowledge' approach has two desirable attributes: minimal effort is required to commence testing, and assumptions are not allowed to restrict the scope of testing. The success of *fuzz* illustrates that random testing is a viable means to induce fault states in applications.

There are, however, significant disadvantages to the random approach to generating test data. Consider our trivial testing scenario as outlined in figure 5.2: there is no guarantee that random generation will ever produce the combination 1011 required to trigger the application to fail. There is, of course, a one-in-16 probability of randomly generating the test instance required, but this is a trivial application. Real life applications typically have a very large input space, and the chances of randomly generating a test instance that will trigger a failure are considerably reduced.

5.2.1 Code Coverage and Fuzzer Tracking

A key question in fuzzing, in fact in any form of testing, is: when do you stop? Setting criteria that will determine test completion is important in order to deploy resources efficiently. Yet, setting and measuring such criteria when fuzzing is very difficult, mainly due to a lack of measurable parameters that describe fuzz test completeness. One possible metric for tracking fuzz test completion is *code coverage*, which Sutton et al. define as *"the amount of process state a fuzzer induces a target's process to reach and execute"* [46, p. 66].

A program may be thought of as a collection of branching conditional execution paths. This is true at the source code level and at the binary (i.e. object code) level. Data input to the program determine the path that execution takes via conditional statements. Different paths result in different sections of the program being executed. Imagine that a vulnerability exists in a specific section of a program. In order to trigger that vulnerability we will need to achieve two goals:

1. ensure that the vulnerable region of code is executed;
2. ensure that suitable input is passed to the vulnerable section, such that the vulnerability is triggered.

Both of the above are achieved through input; therefore input comprises two basic components:

1. data to navigate through conditional code paths, in order to establish a specific application state;
2. data to be passed into the application for processing once a specific application state has been reached.

Ideally, we should aim to execute all code regions in order to satisfy ourselves that we have tested the whole application. However, DeMott makes the point that, from a security perspective, we are only interested in coverage of the attack surface: code that is reachable via input:

> *"Some code is internal and cannot be influenced by external data. This code should be tested, but cannot be externally fuzzed and since it cannot be influenced by external data it is of little interest when finding vulnerabilities. Thus, our interests lie in coverage of the attack surface."* [8, p. 8]

This means that effective coverage could mean a result of less than 100% in terms of code paths executed.

Dynamic testing can be used to determine what control paths are executed during fuzzing, via *path tracing*. Static analysis can be used to map all of the possible code paths of a defined region of code or an entire application. As Sparks, et al. put it:

> *"A control flow graph for an executable program is a directed graph with nodes corresponding to blocks of sequential instructions and edges corresponding to non-sequential instructions that join basic blocks (i.e. conditional branch instructions.)"* [42, p. 2]

We have already mentioned structural testing (see Chapter Three, *Software Security Testing*). A disassembler such as IDA Pro[2] can be used to generate a control flow

[2] `http://www.hex-rays.com/idapro/`

graph from a binary executable file [42, p. 2]. This graph could then be used to as a basis to determine which regions of an application are executed and which are not during a test run. This would require runtime analysis of the target internals. The term *white box fuzzing* is used to describe methods where internal, structural analysis techniques are used to guide fuzzing [**?**, p. 1].

Code coverage is relevant to fuzz testing since it measures how much of the application code has been tested. If you found no bugs but had covered only 10% of the application state, you would not be likely to claim that the application was free of software defects.

Code coverage may be used to track fuzzer progress during fuzzing and to determine if fuzzing is 'complete'. However, while code coverage is a very useful metric, and may be the only useful metric for measuring the performance of a fuzzer, it is important to note that code coverage only tells you *what percentage of (reachable) code was executed*. It does not tell you what range of input values were passed to the application once each of the different application states were established [46, p. 467].

If you completed a fuzz test of an application, found no bugs, and determined that the code path coverage was 100%, you could not argue that the application was devoid of defects. This is because (as we shall see later in this chapter), for all but the simplest applications, it is infeasible to pass every possible input value into the application for every possible application state. Hence, code coverage is important and desirable, but cannot guarantee that testing is complete or exhaustive.

Microsoft applies a pragmatic solution to the problem of measuring the range of values passed to the application:

> *"[The Security Development Lifecycle] requires that you test 100,000 malformed files for each file format your application supports. If you find a bug, the count resets to zero, and you must run 100,000 more iterations by using a different random seed so that you create different malformed files."* [21, p. 134]

It might be possible to apply this approach at a finer degree by, for example, specifying that a either specific range of intelligently selected values, or a specific number of random values must be passed to each identified data element (such as a byte or a string).

Regardless of the limitations of code coverage as a metric for fuzzing, achieving high code coverage is a significant problem for random testing. Patrice Godefroid provides an example illustrating the issue.

> *"... the THEN branch of the conditional statement IF (x==10) has only one in 2^{32} chances of being exercised if x is a randomly chosen 32-bit input value. This intuitively explains why random testing usually provides low code coverage."* [16, p. 1]

That is not to say that the random data generation approach does not work, but that the benefits offered by random generation should be qualified by its limitations. Random testing should never be discounted, but should be always be used with an awareness of the problems that it is not able to solve.

5.2.2 Static Values

Static values (also referred to as 'magic numbers') in binary data formats and network protocols are problematic for random generation methods. The presence of these values at specific locations is often tested in network protocols or binary file formats in order to detect data corruption, or simply as a means to identify the data format and differentiate it from other formats. The probability of randomly generating a valid static value is a function of the size of the value, but to do so at a specific position within a sequence of values is very small indeed.

Consider the static value used in Java class files. Unless the first four bytes of a Java class file are set to the value 'CAFEBABE', the file will be rejected by the class file loader [29, p. 141].

The probability of randomly generating the value 'CAFEBABE' is, once again, 2^{32} (assuming 4-byte character representation). The probability of randomly generating the value 'CAFEBABE' *at a specific location in a test instance* is a function of the length of the sequence and is vanishingly small; hence a large ratio of test instances that will yield no useful information[3] will be generated. This may be termed low *efficiency* of test data.

[3] Beyond the fact that the application is robustly rejecting test instances where the static value is incorrect.

5.2.3 Data Structures

In addition to magic numbers, application input is invariably subject to a high degree of structure. Perhaps the most common structural features arise from the need for input to compartmentalise and label separate regions in the form of *headers*.

Take, for example, the Portable Executable (PE) header, used to identify files that conform to the Portable Executable structure. Among files that conform to the PE structure are .exe files, .dll files and .cpl files. The PE header allows the Windows operating system loader to (amongst other things) map PE files into memory. Since the PE header is located at the beginning of all PE files, the first 15 bytes of a .cpl file viewed using a binary editing application in figure 5.3 show the beginning of the PE header.

```
Offset     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
```

Figure 5.3: The first 15 bytes of `Access.cpl`.

If the PE header is altered, the windows loader may reject the file, and the data within the file may not be loaded into memory.

This real-world example raises an important consideration: that data often passes through various levels of validation before being passed to the core of an application for processing. Where networking is employed, validation may occur during transition before the data has reached the application.

Encapsulation

Many common data protocols are themselves contained within other protocols. Networking protocols that encapsulate higher-layer protocols are an example of this. If input does not conform to structures defined at the lowest of levels, it will be rejected and not passed up to the level above.

The high degree of structure that is typically present in application input data adds significant redundancy to the application input space. This has the effect of

reducing the effective input space, but in a complex manner. Charting this effective input space requires a detailed understanding of the application and/or the format of the data it consumes. However, as we shall see in Chapter 6, *Data mutation fuzzing*, by sampling an input instance (or better still a range of instances) when the application is in use, one could obtain a valid 'subordinate' image of the effective input space, with very little effort.

Let us amend our trivial scenario to reflect the fact that application input data typically has structure. Let us say that the application checks all input and rejects any where the fourth switch is set to zero, representing a trivial static value check.

	Rejected input								Accepted input							
Switch 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Switch 2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
Switch 3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Switch 4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Figure 5.4: A visual representation of the input space of four binary switches, an error state and a static check.

Figure 5.4 illustrates the effect (via diagonal hatching) of rejecting input where the fourth switch is set to zero: the effective input space is reduced (in this case halved) as a result of a static value check, though the absolute input space remains the same.

Since random generation fails to account for any structure in input data, a large proportion of randomly generated test data will be rejected at every point where a check occurs, and there may be many such checks. Again, the small scale of our trivial example input space fails to indicate the significance of this problem. As the scale of the input space increases, the ratio of rejected to accepted test instances will increase significantly.

This raises another important consideration: that each iteration of a test instance

takes a finite amount of time to process. The test instance must be passed to the application, the application needs to process the data and the oracle[4] needs time to determine the health of the application/host. Hence, test data *efficiency*, (which we shall define to be the ratio of test instances that yield valuable information to test instances that do not[5]) is a valuable commodity.

5.3 Brute Force Generation

Brute force generation involves programmatically generating every possible permutation of the input space. This approach requires no knowledge of the target, with the exception that input space dimensions should be known so as to limit data generation. Since brute force generation requires that the input space dimensions are bounded, there will be a finite amount of test data and a hence a clear indication of the completion of the test is possible.

Brute force generation could potentially provide a high level of assurance by testing the application's response to all possible potential input values and traverse all possible combinations of input.

However, like random generation, brute force generation is significantly impacted by the large absolute input space presented by most applications and the high degree of structure found in application input data, which means that the effective input space is vastly smaller than the absolute input space.

Since brute force generation is a zero-knowledge approach, it cannot account for the vastly reduced effective application input space, and must generate test data that will enumerate the absolute input space. This results in extremely poor test data efficiency.

For example, consider the Hypertext Transfer Protocol as defined in RFC 2616. There are a limited number of methods such as GET, HEAD, TRACE, OPTIONS and so on. Unless a Hyper Text Transfer Protocol (HTTP) request is prefixed by one of the required methods, the request will be rejected. Though it would certainly be

[4]Defined in Chapter 4, *Fuzzing – Origins and Overview.*

[5]This is a rather murky definition in that every test instance yields *some* information. However, it should be obvious that repeatedly proving that an application robustly rejects input with malformed structure does not yield much value.

possible to generate all of the required methods by brute force generation, the poor efficiency of this approach would mean that many millions of useless test instances would be generated.

It is important to emphasise how infeasible brute force data generation is. We have already seen that a finite time is required to process each test instance. In order to brute force fuzz all values of a 32 bit integer, a total of 4,294,967,295 test instances would be required. Disregarding the time and space required to generate and store this test data, it would take 500 days to process each of the required test instances assuming it would take one hundredth of a second to process each one.[6]

Brute force data generation has been applied successfully in the field of cryptography for enumerating a key space, but there are many differences between key space enumeration and application input space enumeration. Even the largest of (feasibly brute force-able) key spaces are considerably smaller than the smallest of input spaces.[7] Moreover, application input is usually highly structured with large amounts of redundancy, compared with a key space which should be highly random with very low levels of redundancy.

Ultimately brute force generation has never been applied as a fuzz test data generation method due to the very poor efficiency of the test data.

5.4 Chapter Summary

Random generation is a valid method for generating test data, and has been used to identify security vulnerabilities in software applications, not only in the form of the original fuzzing tool in 1989, but also in recent, enterprise level applications.[8]

The key benefits of random generation are that it requires little or no analysis of the target application or the data it processes, and this means fuzzing can be rapidly

[6]This is a conservative estimation. In the authors (limited) experience, file fuzzing required approx 200 mS per test instance on an Intel Pentium P4 processor, 2GB RAM, 800 MHz, Win XP SP2, and network fuzzing required approx 1 second per test instance.

[7]It is generally agreed that brute forcing approx 70 bit symmetric keys is at the edge of feasibility, compare this with the input space of a web server, browser or word-processing application

[8]Sutton et al. provide an example of a vulnerability that could easily be found using random fuzzing in Computer Associates BrightStor ARCserve data backup application [46, p.367]. All that is required to trigger the vulnerability is to send more than 3,168 bytes to a specific port number.

initiated. Additionally, the fact that knowledge of the application is not required means that testing is not influenced by assumptions that may arise from possession of knowledge of the target.

The key disadvantages of random generation are:

- poor efficiency: the potential for a very large proportion of test instances to yield no valuable information[9]
- the resultant poor code coverage and inability to penetrate into 'deeper' application states.
- the lack of assurance that the input space will be completely enumerated;
- the infinite test data and hence no clear indication of when fuzzing is complete;

Brute force data generation is theoretically interesting, but is simply infeasible for all but the simplest of applications, rendering it useless as a method for fuzz test data generation.

The next chapter will examine 'blind' data mutation fuzzing, where sample data is collected and mutated, solving some, but not all, of the problems encountered by fuzzers.

[9]This is due to the fact that random testing cannot account for the difference between the *actual* input space (the range of data that can be passed to the application) and the *effective* input space (the range of data that will not be rejected by the application)

CHAPTER

SIX

DATA MUTATION FUZZING

In order to address the high degree of structure found in application input data, a different approach to purely random or brute force test data generation is *data mutation*. Data mutation involves capturing application input data while the application is in use, then selectively mutating the captured data using random and brute force techniques. By using captured data as the basis for generating test data, conformance to the effective input space is considerably higher than that seen in purely random or brute force test data generation approaches. A high level of conformance to the effective application input space means that static numbers and structure such as headers all remain intact, unless, of course, they are subject to mutation.

Data mutation fuzzing, like random and brute force fuzzing can be performed at many different levels: it can be applied at the application layer, usually in the form of mutated files, and it can be applied at the network layer, in the form of mutated network protocols.

Because the data capture phase is generally simple and fast[1], data mutation need not require significantly greater effort or time to commence fuzzing in comparison to zero-knowledge approaches such as purely random or brute force generation. Yet, due to the similarity to valid input, the test data will have a much higher efficiency. Hence, the benefits of random or brute force mutation (i.e. minimal effort or knowledge required) can be achieved, while disadvantages (poor code coverage, poor efficiency) are avoided.

[1]Typically, a file is copied or an exchange of network traffic is captured.

6.1 Data Location and Data Value

An explanation of two key concepts is required: data *location* and data *value*. A data location is a unique address that may be used to identify a single data item (character, byte, bit, etc) in a sequential array of such items. A data value is simply the value stored at a specific data location.

```
Offset     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
```

Figure 6.1: The first 15 bytes of `Access.cpl` with data location ‘6’ highlighted.

In figure 6.1 we see the same first 15 bytes of `Access.cpl` as seen in figure 5.3. However, in figure 6.1, data location 6 has been highlighted in red. The value at this data location is hexadecimal 00.

Mutating data (for the purposes of generating test data for fuzzing) may be defined as: altering data values at specific data locations. Randomness and brute force techniques may be applied to both the selection of locations and the manner in which the values held at the selected locations are modified.

It is important to note that mutation involves *selective* alteration: only a subset of locations should be selected for modification in any one instance. In this way much of the structure of the source data is maintained.

6.2 Brute Force Data Mutation

Within data mutation, brute force techniques can be applied to either location selection or value modification, or both.

6.2.1 Brute Force Location Selection

This approach offers the opportunity to programmatically move through the source data, determining the effect on the application of changing the value of each location of the source data. Hence, a list of vulnerable locations of the source data can be

identified without any knowledge of the format or the content of the data. This technique is successfully applied in Chapter 8, *Case Study 1 Blind Data Mutation File Fuzzing*.

This approach is entirely valid, but there is a risk of a 'combinatorial explosion' if multiple value modifications are to be applied to every data location.[2] As a result, brute force techniques are of practical use to testers, but often only when combined with limited scope in terms of:

- the range of modification values used
- the number of locations to be modified

The former means restricting the values overwritten to selected locations. In Chapter 8, *Case Study 1 Blind Data Mutation File Fuzzing*, selected locations are overwritten by only one value: zero. The latter approach involves selecting a finite region of locations within the source data, limiting the range of locations to be enumerated.

What follows is an example of what the author terms a *brute force compromise*, since the generated test data size was at the edge of acceptability, the author was forced to compromise and reduce the scope of testing.

In Chapter 8, *Case Study 1 Blind Data Mutation File Fuzzing*, brute force location selection is used to fuzz `Rundll32.exe` by mutating `Access.cpl`, a file that is passed to `Rundll32.exe` as input. `Access.cpl` is 68 kB long, which means that 68,000 test cases had to be generated in order to modify each byte (in file fuzzing it is common to generate all of the test data at the start of the test). This was feasible, but was at the edge of acceptability. This meant that options for value modification had to be severely limited; in this case, modification was limited to setting the selected location to zero. As a result, it took 2 hours to create all of the test cases, 4 gigabytes to store them and about 4 hours to test them all.

Had the author wanted to modify each location through all 255 possible values (each location represented a byte in this case), then the number of test cases would have been multiplied by 255.

[2] where the combinations of mutations result in a prohibitively large number of test cases

Brute force location selection fuzzing can be used to identify interesting regions of the source data. Once identified, such regions can be manipulated manually, outside of the test framework i.e. using a hex editor, or supplying a range of values via the command line. Alternatively, the region could be manipulated within the testing framework via brute force value modification.

6.2.2 Brute Force Value Modification

Another approach to brute forcing is to select a single data location (or range of locations) and enumerate every possible value that that location could be. Of course, the effort required by this approach will be determined by the nature of the data type at the location in question, since the data type will determine the number of values that will need to be enumerated - i.e. if the location data type is a byte, then 255 test instances will be required. If it is a 32 bit double word, then considerably more test instances would be required.

6.3 Random Data Mutation

For many people, random mutation is the definitive mode for fuzzing: random mutation is applied to well-formed source data to produce "semi-valid" test data with a high degree of structure, coupled with specific regions of randomly generated data. Random mutation can reproduce valid static values and input structure in order to penetrate into the application, whilst also exercising specific regions of the target application.

However, random mutation has many limitations: one only has to recall Godefroid's conditional statement to realise that random mutation is severely limited in terms of code coverage. Further, random mutation, like all other forms of zero knowledge data generation, cannot produce valid self-referring checks such as checksums or Cyclical Redundancy Checks (CRCs).

6.4 Data Mutation Limitations

Considering data mutation as a whole, two key limitations are:

1. The source data is not a representation of the effective input space.
2. Self-referring checks are extremely unlikely to be satisfied.

Both limitations result in reduced code coverage.

6.4.1 Source Data Inadequacy

It is unlikely that a single example of a given protocol or file format will exercise all possible functionality. Consider an application that handles .jpg image files: it will probably be capable of processing all possible aspects of the .jpg standard in order to be considered compatible with it. Now consider a single randomly chosen .jpg image file: it is unlikely to make use of every aspect of the .jpg standard. Indeed, the standard has aspects which may be mutually exclusive in particular instances (e.g. an image can be in either portrait or landscape orientation, but not both simultaneously), but both must be supported by applications processing such media.

The difference between requirements on source data (format data in such a way as to describe something specific, such as a visual image) and application input processing (parse and process data in a manner that satisfies one or a number of protocols or standards) mean that the usage of source data as a means to enumerate the effective input space is limited, unless the source data gathered exercises all possible aspects of the protocols or standards that the application is compliant with.

Hence, data mutation is not guaranteed to enumerate the input space, since the source data will be a subset of the input space. Of course, mutation will increase the size of the source data space with respect to the effective input space, but mutation without intelligence is unlikely to make up the difference between the two.

In order to increase the size of the source data relative to the effective input space, it is possible, and desirable to use more than one source file for data mutation fuzzing. As Howard and Lipner put it:

> *"You should gather as many valid files from as many trusted sources as possible, and the files should represent a broad spectrum of content. Aim for at least 100 files of each supported file type to get reasonable coverage. For example, if you manufacture digital photography equipment, you need representative files from every camera and scanner you build. Another*

way to look at this is to select a broad range of files likely to give you good code coverage." [21, 155]

6.4.2 Self-Referring Checks

We have seen that data mutation can overcome problems such as static values and structure in data formats and protocols by leveraging source data to penetrate into the target application. However, self-referential checks are a problem that data mutation cannot overcome.

Self-referential checks measure an aspect of input data and store the result with the data. The application independently measures the same aspect and compares the result generated with the value stored in the data. For example, consider the use of checksums and Cyclical Redundancy Checks (CRCs); unless a fuzzer is aware of, and can account for such checks, a very high proportion of test data will be rejected by active checks such as these, and this will have a considerable impact on efficiency.

Furthermore, these checks are commonly deployed, and may occur at multiple levels. For example, there is a Content Length field in the HTTP protocol, but there are also checksums at Internet Protocol level: the IP header value is 'protected' by a checksum, which, if found to be invalid, will result in the IP packet being rejected.

If one is blindly mutating data, it is likely that the first point at which checks are performed will reject the input and the test data will not penetrate up to the higher layers of the protocol stack or the application itself.

In his seminal work *The Advantages of Block-Based Protocol Analysis for Security Testing*, Dave Aitel describes a requirement when fuzz testing network protocols to "flatten" the IP stack, removing the inter-relationships between higher and lower layers [4, p. 2]. By creating a framework that can dynamically re-calculate protocol Meta data (such as data length values or CRCs), self-referential checks can be maintained as selected values are altered.

> *"Any protocol can be decomposed into length fields and data fields, but as the tester attempts to construct their test script, they find that knowing the length of all higher layer protocols is necessary before constructing the lower layer data packets. Failing to do so will result in lower layer protocols rejecting the test attempt."* [4, p. 2]

We shall explore this 'intelligent', analysis-based approach to fuzzing in Chapter 8, *Protocol Analysis Fuzzing*.

6.5 Chapter Summary

We have explored the range of zero knowledge test data approaches, encompassing random, brute force and data mutation.

We have seen that data mutation can solve some of the problems faced by random and brute force testing; namely maintaining magic numbers and basic input data structure. We have also seen that data mutation cannot solve all the problems: it cannot overcome self-referential checks, resulting in poor test data efficiency and low code coverage as lower layer checks fail and test data is not passed further into the application; and, the selection of source data to mutate can limit the potential to generate test data that will enumerate the input space.

Data mutation has a higher chance of finding vulnerabilities than random testing, and brute force testing is simply infeasible. Data mutation also requires minimal effort to initialise testing, compared to zero effort required for random or brute force testing. These factors together mean that the author would always err toward data mutation when selecting between any of the zero knowledge approaches.

Table 6.1 shows a comparison of the various zero knowledge test data generation approaches and protocol analysis-based generation, which will be covered in Chapter 10, *Protocol Analysis Fuzzing*.

This concludes our two-chapter examination of the theory of zero knowledge test data generation for fuzz testing. The next chapter examines the role and importance of exception monitoring in fuzz testing.

Data Generation Method	Finite test data	Requires analysis of application	Likely to produce valid static numbers	Likely to maintain data structure	Likely to produce valid CRCs
Random generation	N	N	N	N	N
Brute force generation	Y	N	N	N	N
Data mutation (random)	N	N	Y	Y	N
Data mutation (brute force)	Y	N	Y	Y	N
Analysis-based data generation	Y	Y	Y	Y	Y

Table 6.1: A comparison of various zero-knowledge test data generation approaches.

CHAPTER SEVEN

EXCEPTION MONITORING

> *What do you get when you blindly fire off 50,000 malformed IMAP authentication requests, one of which causes the IMAP server to crash and never check on the status of the IMAP server until the last case? You get nothing. [...] A fuzzer with no sense of the health of its target borders on being entirely useless.* [46, p. 472]

In a presentation entitled *How I learned to stop fuzzing and find more bugs* given at Defcon in August 2007 [48], Jacob West, the Head of Security Research at Fortify software, described an experience he had that provides insight about the importance of target monitoring when fuzzing. West had been fuzzing a target and had found no defects. The target host operating system was restarted, whereupon it failed to boot because a critical system file had been overwritten as a result of fuzz testing. However, since the target application had not raised an exception, the oracle used had not been triggered and the defect, though triggered, had not been detected. The defect would never have been identified were it not for the fact that a file critical for the boot process was overwritten [48].

This example demonstrates that simply monitoring for target application exceptions (the standard method of monitoring employed by most fuzzers) is flawed in that many defects may not be detected. While most of the focus within fuzzer development has been upon the generation of test data that will trigger the manifestation of defects, *target monitoring* for detection of defects may have been neglected. However, given the choice between devoting time to developing new ways of triggering significant flaws, or developing better methods for detecting subtler flaws, the author would

err toward the former. Significant flaws (i.e. defects that result in application failure) often represent significant risk to users and until it is common for production code to ship without significant flaws, this is where the focus of security testing should remain.

7.1 Sources of Monitoring Information

Most modern operating systems provide, by default, a range of sources of relevant information including application and operating system logs, error messages and alerts. There are also a wealth of tools which can be used to actively monitor and report on the state of a target application and its host operating system.

An idealistic approach to target monitoring might be to capture a snapshot of the entire system state before and after each test instance is passed to the application. The two states could then be compared to produce a 'difference map' which would describe the effect of each test instance not only upon the target, but on the host system.

Unfortunately, this approach is simply infeasible. To take a complete snapshot of a nominal system might require between 4 and 8 GB of storage[1]. Since it is not unusual to run thousands individual tests within a test session, this could quickly equate to a requirement for an infeasible amount of storage. Due to limitations in storage space and processing capacity, compromises have to be made, just as they have to be made in other aspects of fuzz testing such as employing intelligent integer values rather than brute force enumeration.

As a result of compromises made during test data generation, it is reasonable to state that of the total set of all vulnerabilities v, only a subset, w of v will be triggered by the test data, since it is generally not feasible to enumerate the entire application input space. Unfortunately, of the subset of triggered vulnerabilities u, only a further subset x will be detected. This is illustrated in figure 7.1.

[1]Based on capturing the operating system, the application and system RAM.

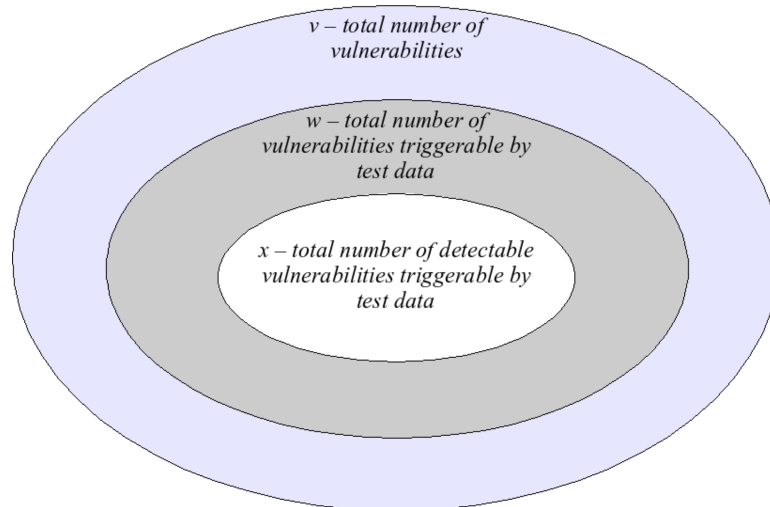


Figure 7.1: Illustrating the inter-relationships between limitations in test data generation and vulnerability detection.

7.2 Liveness Detection

At the opposite end of the spectrum from storing the entire system state for every test case instance, let us consider the minimum requirements for error detection. It would be useful to stop testing when the target stops responding. Feedback from a simple 'liveness' check on the target application could be used to halt automated testing. This would prevent test data from being passed to a non-responsive application, potentially resulting in false negatives where test data capable of causing a defect to manifest are errantly classed as harmless.

By testing liveness at the commencement of each test instance, we can identify input that induces application failure. However, we will only detect failures that cause the application (or host operating system) to become unresponsive.

7.3 Remote Liveness Detection

One way to determine the liveness of a remote application is to send a 'known good' test case instance and monitor for a valid response. Using the 'ping' command is an alternative, but this would only prove that the target application's host operating system was responsive to pings. A response from a valid test case instance would prove that the application itself was active. The PROTOS testing suite supports this approach [24, p. 94], which could be termed *valid-case instrumentation* [26].

The output of a fuzzer employing this form of target monitoring would usually be a log of test case instances that triggered a fault. The tester could then submit offending test case instances to the development team. Any form of fault analysis would need to be conducted manually, probably by attaching a debugger to the target, manually submitting problematic test case instances to it, and observing the outcome.

7.4 Target Recovery Methods

Once a fault state has been induced and detected, the target needs to be returned to a functional state. This provides assurance that the target application is functional when test input is passed to it, and facilitates unsupervised, automated testing where the fuzzer is able to continue testing once a fault state has been induced in the target. This can be achieved via one of two methods:

1. restart the target at the commencement of each test case instance; or,
2. monitor the target, detect fault states, and restart the target when a fault state occurs.

The former approach is simpler but means that time is wasted unnecessarily restarting the target. This is best suited to local file fuzzing, where the application and the fuzzer reside on the same host operating system, since monitoring the application for exceptions and restarting it can be performed relatively quickly. This approach is only possible when the application can be simply launched with each test case instance and there is no need to establish a specific target state prior to submitting input.

FileFuzz is an example of a fuzzer that employs this method (see Chapter 8, *Case Study 1 – 'Blind' Data Mutation File Fuzzing*). The application is launched and a

test case instance is passed to it. The application is monitored for exceptions, and after a set duration it is terminated, re-launched with the next test case instance, and the cycle continues.

Problems with this approach are:

- in some cases, restarting the target is insufficient: preconditions have to be satisfied in order to invoke a required target system state prior to testing,
- it may be time-consuming to restart the target for each test instance,
- concurrent errors which occur as a result of a sequence of test case instances will not be detected as the target state is reset at the commencement of each test,
- if set too small, the duration setting may be insufficient to detect some errors, if set too long, it will extend the overall test duration unnecessarily,
- if the host operating system is affected by testing, restarting the application will not restore the default operating system state.

The second approach involves either a simple liveness check, or a more elegant solution, where a debugger is employed to monitor the target application so that exceptions can be intercepted, at which point information about the exception is gathered and recorded in the form of a 'crash report'.

This approach, termed *debugger-assisted monitoring* is the approach taken by Sulley fuzzing framework and the FileFuzz file fuzzer, and many others. Such fuzzers can usually be left to complete a fuzz testing run without human intervention, and their output will consist of a list of crash reports (assuming software defects were detected) of varying detail. An example of a crash report generated by Sulley can be found in figure 11.8, Chapter 11, *Case Study 2 - Protocol Fuzzing a Vulnerable Web Server*, and an example crash report generated by the FileFuzz can be found in figure 8.4, Chapter 8, *Case Study 1 - Data Mutation File Fuzzing*.

Other examples of fuzzers that employ this method are the Breaking Point Systems BPS-1000 and the Mu Security Mu-4000 hardware appliance fuzzers. Both appliances are aimed at testing network appliances, and both are able to supply power to a target device, so that the power supply can be interrupted in order to reset the device when a fault state is detected.

Problems with this approach mainly arise when fault states, also termed *exceptions* are not detected.

7.5 Exception Detection and Crash Reporting

Software defects actually manifest at the hardware, object code, assembly level [46, p. 474], where object code running on the Central Processing Unit triggers an exception.

> *"Exceptions are classified as faults, traps, or aborts depending on the way they are reported and whether restart of the instruction which caused the exception is supported."* [23]

As a result, exceptions and crash reports refer to events occurring at the assembly level. In order to interpret this information and get the most form it, a tester needs to cultivate an understanding of the operation of processors at the hardware level.

Yet, as we have seen, the sources of exceptions are usually software defects: errors made by programmers usually working at the source code level. For example, the use of a vulnerable function such as strcpy might result in a vulnerability where long input strings cause an Access Violation exception being raised due to a read access violation on EIP. An analyst that wishes to do more than simply present defect reports that list input value / exception types pairs, will need to have an understanding of both the source code and assembly level programming. We will explore the interpretation of fuzzer output later in this chapter.

7.6 Automatic Event Classification

Target monitoring should be shaped to inform and assist the fuzzer output analysis phase as much as possible. Once fuzzing has been completed, the first task of the tester is to triage the crash reports. As part of the triage process, it is useful to group exceptions (and the test cases that caused them) into classes.

If performed, automatic collection of crash reports offers many benefits. One of these is that events may be automatically grouped into classes, or *'buckets'*. This can aid the process of fuzzer output analysis, particularly if a large number of defects

are identified. Lambert describes how the automatic 'bucketization' of exceptions has been implemented at Microsoft:

> *"This was accomplished by creating unique bucket ids calculated from the stack trace using both symbols and offset when the information is available. The bucket id was used to name a folder that was created in the file system to refer to a unique application exception. When an exception occurred, we calculated a hash (bucket id) of the stack trace and determined if we had already seen this exception. If so, we logged the associated details in a sub-directory under the bucket id folder to which the exception belonged. The sub-directory name was created from the name of the fuzzed file that caused the exception."* [27]

The technique of automatically identifying and grouping similar exceptions (in this case by comparing stack trace symbols and offset values) means that the number of exceptions that the tester has to examine is drastically reduced. This is particularly useful if, as is not uncommon, large numbers of similar exceptions occur that are attributable to a single 'noisy' defect.

7.7 Analysis of Fuzzer Output

There is a range of defect types which may be induced via malformed input. Some of these are trivial to exploit, some are only exploitable in certain conditions and some are very unlikely to be exploitable.

The safest approach to take to assessing the exploitability of defects is to treat *any* form of application failure as exploitable. This is because determining whether a defect could be exploited is usually non-trivial, and may require more effort than rectifying the defect. An attacker may have more time, motivation, information, money, support and skill than the person who must decide if a defect is exploitable. Furthermore, information security is a rapidly evolving field. New exploitation techniques emerge changing the way certain defects are viewed. This means that a seemingly harmless defect today can become a critical security vulnerability tomorrow.

It is unlikely that all of the currently possible exploitation techniques are in the public realm. The highly valuable nature of undisclosed vulnerabilities means that

there may be undisclosed exploitation techniques which could render a seemingly innocuous defect a significant threat.

Unfortunately, where there are many defects to be managed, a pragmatic approach must be adopted, where defects are ranked in terms of their potential severity in order to determine where scarce resources can be deployed.

Howard and Lipner present a table which outlines the approach taken by Microsoft specifically to ranking errors detected via fuzzing [21]. Analysis of fuzzer output is conducted at the assembly level.

Category	***Errors***
Must Fix	Write Access Violation
	Read Access Violation on extended instruction pointer (EIP) register
Must Investigate (Fix is probably needed.)	Large memory allocations
	Integer Overflow
	Custom Exceptions
	Other system-level exceptions that could lead to an exploitable crash
	Read Access Violation using a REP (repeat string operation) instruction where ECX is large (on Intel CPUs)
	Read Access Violation using a MOV (Move) where ESI, EDI, and ECX registers are later used in a REP instruction (on Intel CPUs)
Security issues unlikely (Investigate and resolve as a potential reliability issue according to your own triage process.)	Other Read Access Violations not covered by other code areas
	Stack Overflow exception (This is stack-space exhaustion, not a stack-based buffer overrun.)
	Divide By Zero
	Null dereference

Table 7.1: Ranking errors discovered using fuzz testing [21].

Table 7.1 shows the approach taken by Microsoft to rank errors discovered via fuzzing. It is clear from this table that the bulk of faults that are discovered via fuzzing are read or write access violations. Let us briefly examine the two most significant errors.

7.8 Write Access Violations

Write access violations exceptions are raised when an application attempts to write to a memory location that it is not permitted to (usually outside the process memory space). If a defect can be exploited to raise a write access violation exception, and the attacker exploits the defect to write to a memory location the target is permitted to write to (usually *inside* the process memory space), no exception will be raised and the attacker-controlled write operation will execute. Hence, write access violations indicate the potential for attackers to modify data held in memory addresses that the vulnerable application can access. Unless the destination pointer is null, it is trivial to, for example, redirect execution flow and hence execute arbitrary code with the security context of the application.

7.9 Read Access Violations on EIP

A read access violation on the EIP could be used to read attacker supplied data into the Extended Instruction Pointer, redirecting execution to an attacker supplied memory location. Exploitation of this type of vulnerability is demonstrated in Chapter 9 *Case Study 2 – Using Fuzzer Output to Exploit a Software Fault.*

7.10 Chapter Summary

We have seen that target monitoring is critical for detecting faults triggered by test data and also for ensuring test data is passed to a responsive target. Failures or poor performance in either area can lead to false negatives. Target monitoring is a critical aspect of fuzzing, yet there has not seen a great deal of development in this area. Sutton et al. suggest that, in the future, technologies such as Dynamic Binary Instrumentation (DBI) could be implemented to advance this area of fuzzing [46, p. 492]. We briefly explore DBI in the Outlook section of Chapter 12, *Conclusions*.

We have seen that 'crash reports' for fault reporting require understanding of processor operation to be fully understood, and can be integrated into automatic event classification schemes such as that applied by Microsoft. We have also shown the approach taken by Microsoft to ranking errors discovered via fuzzing.

The next chapter examines an advanced approach to fuzzing developed in the late nineties, termed protocol analysis fuzzing, or, *intelligent* fuzzing.

CHAPTER

EIGHT

CASE STUDY 1 – 'BLIND' DATA MUTATION FILE FUZZING

This chapter documents a practical 'blind' data mutation fuzzing exercise. The overall objective was to test whether 'blind' data mutation fuzzing could be used to discover failures in a software component without knowledge or analysis of the target or the data it receives as input. Specifically, we tested how `Rundll32.exe` (a component of the Windows XP Service Pack 2 operating system) responded to mutated forms of a valid `.cpl` file. A total of 3321 different test case instances resulted in target application failure. At least 28 of these test instances resulted in a Read Access violation on the Extended Instruction Pointer register, which could allow an attacker to control the flow of execution, creating the potential for local arbitrary code execution. The nature of these high severity failures is explored in detail in Chapter 9, *Case Study 2*.

The target selected was `Rundll32.exe`, a component of the Windows XP operating system. `Rundll32.exe` is a command line utility program that *"allows[s] you to invoke a function exported from a DLL."* [33]. DLL is an abbreviation of the term Dynamic Link Library. Software libraries contain *code* (instructions), *data* (values) and/or *resources* (icons, cursors, fonts, e.t.c.), and are usually integrated with an application (a process termed *linking*) after software compilation. DLLs are different from standard software libraries in that they are linked dynamically at run-time.

Rather than provide stand-alone executable files, some components of the Windows XP operating system are provided as DLLs. These special-case DLLs rely upon `Rundll32.exe` to load and run them when they are invoked. The operating system achieves this by associating certain file types with `Rundll32.exe`, such that when

files of a certain type are invoked, it will launch Rundll32.exe and pass the file to it as a command line argument. As a result these special-case DLLs must have a different file extension from the normal DLL file extension (.dll).

An example of these special DLLs are .cpl files, and an example .cpl file is Access.cpl, a component of the Windows XP operating system. Access.cpl is launched when a user clicks on the Accessibility Options icon within Control Panel. When invoked, via Rundll32.exe, Access.cpl presents the user with a Graphical User Interface (GUI) for configuring input and output settings, as shown in figure 8.1.

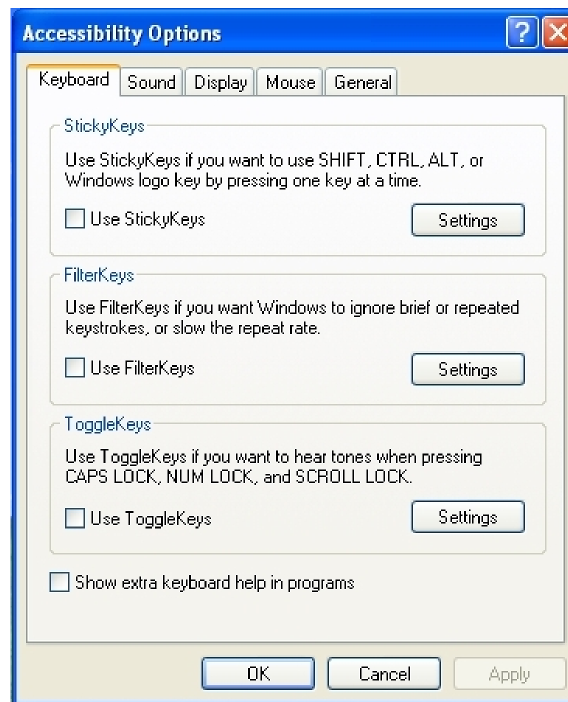


Figure 8.1: The Accessibility Options Graphical User Interface.

Access.cpl was used as the basis for data mutation for a number of reasons, namely: at 68kB, it is a reasonably small file - we shall see later that there is a linear relationship between file length and fuzz test time (and test data storage requirements) when brute force location fuzzing is performed, and; it was recommended as an interesting target by the creators of FileFuzz, the fuzzer used for this exercise.

The following equipment was used in this exercise: a personal Computer running Windows XP Service Pack 2 Operating System Software, *FileFuzz*, a self-contained 'dumb' file mutation fuzzing application which was chosen as it is simple to use, is automated, and contains a debugger that generates easy to understand output logs[1], and *HxD*, a hexadecimal file editor (hex editor), which was used to visually analyse binary files.[2]

8.1 Methodology

In general, the process of data mutation file fuzzing involves taking a valid file and altering that file in a variety of ways in order to produce a malformed file. The target application is then used to open the malformed file and is monitored to determine if opening the malformed file has had any effect on the target application.

Using the FileFuzz application is a two phased operation. First, the FileFuzz *Create* module must be configured to generate the test data, after which it generates and stores the test data. Once the test data creation phase is complete, the execution phase involves first configuring and then launching the *Execute* module. Once this has completed executing each of the test instances in turn, a report may be extracted listing all of the instances where the target application failed generating an exception.

8.2 FileFuzz

FileFuzz was created in 2004 as a stand-alone application for data mutation file fuzzing. It comprises two modules: *Create*, a module that creates multiple test files, by programmatically mutating a single source file and *Execute*, a module which executes mutated files and logs any exceptions. FileFuzz has functionality for fuzzing binary or plain text files. We will focus upon binary file fuzzing, and will make no use of the capability to fuzz text files.

Note that FileFuzz is a deterministic fuzzer: there are no options to employ randomness for fuzzing. FileFuzz is also a 'dumb' fuzzer in that no awareness of the sample file format or the target host application is required.

[1] `http://labs.idefense.com/software/fuzzing.php`
[2] `http://www.mh-nexus.de/hxd/`

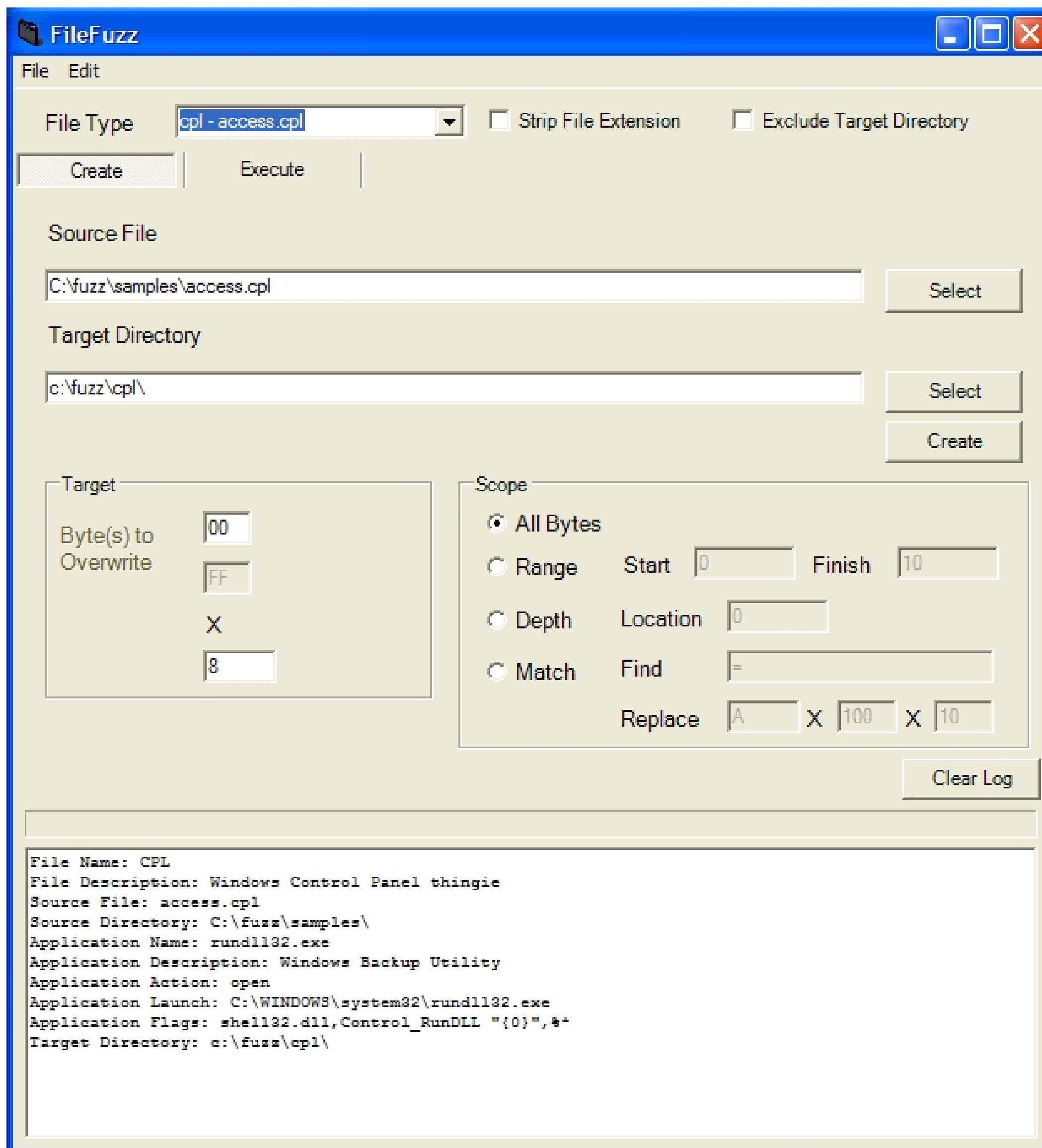


Figure 8.2: The FileFuzz Create module.

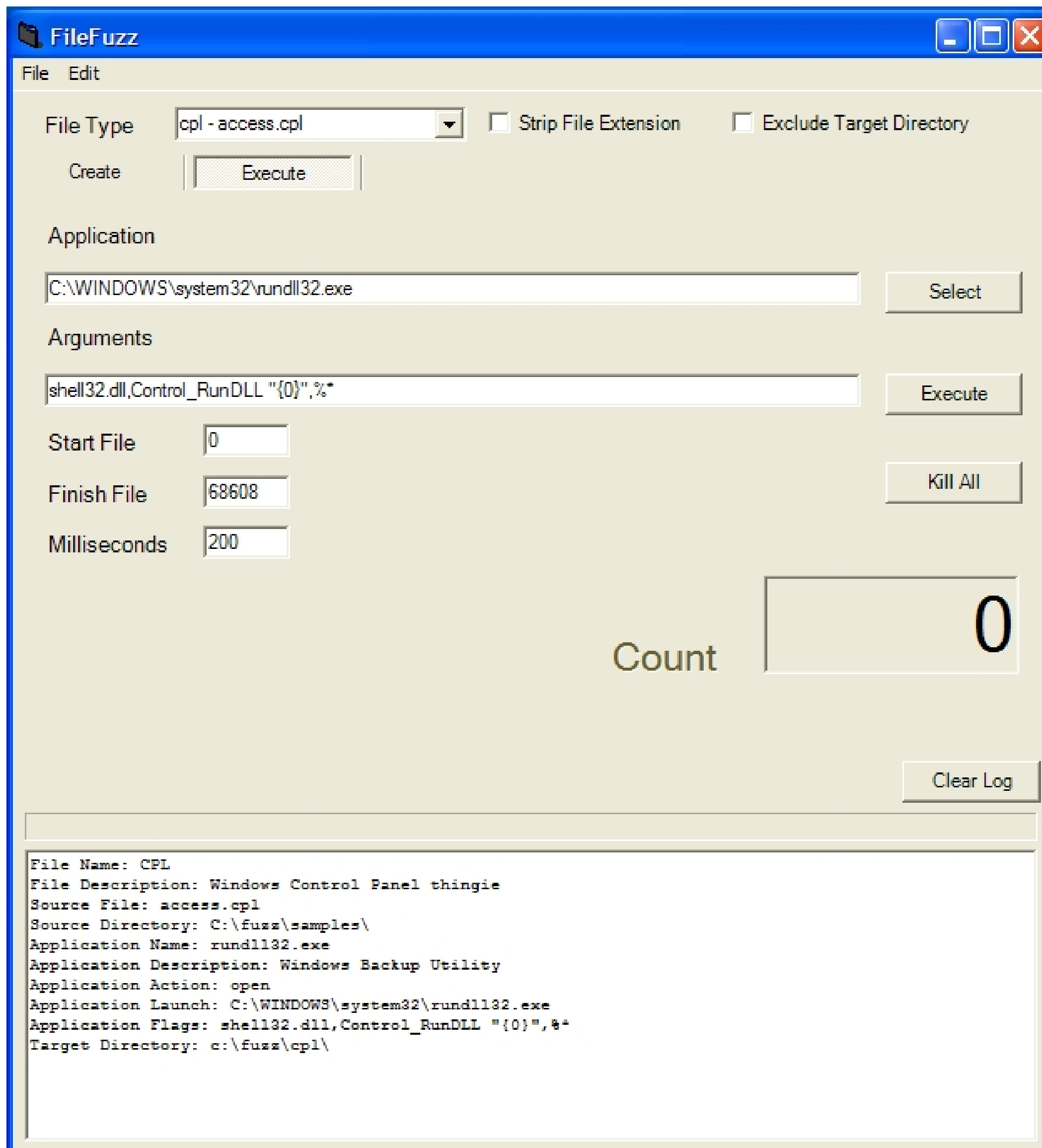


Figure 8.3: The FileFuzz Execute module.

The FileFuzz Create Module

The inputs required by the FileFuzz Create module are:

- the path to a sample file which is to be modified;
- the path to a directory in which to save the modified files;
- the *Scope* and *Target* settings (*Scope* determines the location the modification will take place.)

Ignoring FileFuzz's text file fuzzing capabilities (which are activated by setting the *Match* radio button), the Create module offers three exclusive options for binary data fuzzing *All Bytes*, *Range* or *Depth*. If the *All Bytes* radio button is set, FileFuzz will determine the total number of bytes in the sample file. It will then take the settings from the *Target* section and apply these sequentially to every single byte in the sample file, creating a test case file each time. This was the approach taken and this is discussed in detail later on.

If the *Range* radio button is set, FileFuzz will apply the settings set in the *Target* section to the range of locations set by the user, creating a test case file each time a new value is generated. If the *Depth* radio button is set, FileFuzz will create 255 test case files, where the value at the location set by the user is set to a value between 0x00 and 0xff. For example, the first test case file will have the specified location value set to 0x00; the next will have the same location set to 0x01; the next 0x02 and so on until the value reaches 0xff.

The FileFuzz Execute Module

The inputs required by the FileFuzz Execute module are:

- the path to a directory containing the test case files,
- the path to a target application which is to be launched with the test case files,
- any arguments which the application may be supplied with in order to launch the test cases,

- the *Start File* and *Finish File* values (these decide which will be the first and last test case files to be launched and thus sets the range of test cases to be launched),
- the *Milliseconds* setting sets the number of milliseconds that the target application will be given to launch each test case before it is closed by FileFuzz.

Once launched, the Execute module will work sequentially through the specified range of test case files, launching the target application with each file in turn, allowing it to run for the duration set in the *milliseconds* field, before shutting that instance down and launching another with the next test case.

FileFuzz includes an application called crash.exe which monitors for any exceptions and captures information about critical system registers whenever the target application crashes. The output is a list of exceptions (assuming there are any) and the values of a number of critical system registers at the time of the crash. Figure 8.4 shows an example of a crash report.

```
[*] "crash.exe" C:\WINDOWS\system32\rundll32.exe 2000 shell32.dll,Control_RunDLL
"c:\fuzz\cpl\1.cpl",%*.
[*] Access Violation
[*] Exception caught at 00009c40 add [eax],al
[*] EAX:0007f238 EBX:80000001 ECX:77dd6a51 EDX:00000000
[*] ESI:58ae12e0 EDI:00000001 ESP:0007f208 EBP:0007f228
```

Figure 8.4: An example of a crash report

8.3 FileFuzz Configuration

8.3.1 FileFuzz Create Module Configuration

The path to a sample file which is to be modified was given as:

```
C:\fuzz\samples\access.cpl
```

This was because the source file had been copied into a directory created to hold source files for many audits. The path to a directory in which to save the modified files was set to:

```
C:\fuzz\cpl\
```

A directory entitled 'fuzz' had already been created within the root directory of the workstation. The subdirectory 'cpl' was created within the 'fuzz' directory, so that other audits could also place their test data in other subdirectories in the 'fuzz' directory.

The *Scope* was set to *All Bytes*, since the intention was to work through every single location of the source file, creating a new test instance each time. The *Target* was set to *00* because I wanted to overwrite selected bytes with zeros. The multiplier value 'X' was set to 8 because I wanted to set 8 bytes at a time to zero.

8.3.2 The Rationale for Overwriting Multiple Bytes at a Time

The rationale for overwriting multiple bytes at a time stems from the form of fuzzing that was undertaken: *data* mutation, where code is mutated at the object level, where values are taken from the object code, passed into memory, and then passed into registers on the target platform. In order to induce failure states, multiple values were overwritten in order to affect multiple register locations as they were processed. If we managed to affect a register by mutating the source file, the mutation would have a greater effect on the register if it altered more than one byte. Ideally we would like to alter all of the bytes of the register, since this would indicate that we have been able to completely control that register.

By adopting this approach the data generated would fail to induce or detect other potential bugs that would only be triggered when a single byte of a register is altered. However, we would be more likely to detect faults using our chosen rationale, since it would have a bigger impact on a vulnerable register and the faults detected would be of greater significance since they represent instances where an entire register is controllable.

The controllability of all four bytes of an entire register rather than, say, a single byte is relevant to an attacker since controlling a single byte offers limited scope for

exploitation. Consider that, as the result of a vulnerability, you are able to control a register that reads data from memory. If you control only one byte of the register, you may be able to read a small range of memory locations adjacent to the unmodified location. If you control the entire register, you may read data from any location in memory (dependant on any other memory access limitations).

For this reason, the multiplier value 'X' of the *Target* section of the create module of FileFuzz defaults to a setting of four, since 32 bit registers comprise of 4 bytes. Note that the author errantly set this value to eight in the mistaken belief that registers on the I-86 platform comprised eight bytes. Fortunately, this mistake did not prevent the test from identifying numerous bugs.

8.3.3 The Rationale for Overwriting Bytes with the Value Zero

With hindsight, the author would suggest that this was a flawed decision, born of a lack of experience. As discussed in Chapter 6, the scope of brute force testing usually has to be limited in order to be feasible. The author decided to reduce the scope of testing by employing brute force location selection and overwriting selected locations to a single value. However, the default value of zero was used, which was a poor choice: a better value would have been 0xffffffff. This is discussed further in the *Lessons Learned* section at the end of this chapter.

8.3.4 FileFuzz Execute Module Configuration

The path to a directory containing the test case files was:

```
C:\fuzz\cpl\
```

The path to a target application which is to be launched with the test case files was:

```
C:\fuzz\cpl\
```

In order to determine any arguments which the application must be supplied with in order to launch the test cases, Windows Explorer was used as follows.

1. By selecting `Tools`, then `Folder options` from the Windows Explorer menu, the `Folder Options` window is launched.
2. Within this, by clicking on the `File Types` tab, a list of file extensions are shown, mapped to file types.
3. By double-clicking on a specific extension (we were interested in the CPL extension) an `Edit File Type` window is launched.
4. Finally, clicking the `Edit` button opens an `Editing action for type:` window is launched. This window shows (amongst other things) both the application that is launched by default when a user double-clicks on the file type, and any arguments that are passed to the application at launch.

The full name and path of the application to be executed was entered as:

```
C:\WINDOWS\System32\Rundll32.exe
```

The following arguments, taken from the `Editing action for type:` window, were entered:

```
shell32.dll,Control_RunDLL "{0}",%*
```

The *Start File* value was set to 0 and the *Finish File* value was set to 68608 in order that all of the generated test instances would be tested.

The *Milliseconds* setting was set to 200, one tenth of the recommended value. This was done in order to process the test cases in a reasonable time. Trial and error with brief test runs had shown that errors were still detectable at this setting. Note that there may have been any number of errors that were not detected.

8.4 FileFuzz Creation Phase

The above configuration was employed to systematically alter eight bytes of the sample file at every possible location. Since the file was 68 kB, this meant the creation of 68,608 test case files. In figure 8.6, we see the first 15 bytes of the sample file `Access.cpl`, when viewed using a hex editor. It is common for binary files to be represented in hexadecimal format for analysis since this permits visual interpretation

```
<test>
     <name>cpl - access.cpl</name>
     <file>
       <fileName>CPL</fileName>
         <fileDescription>Windows Control Panel</fileDescription>
     </file>
     <source>
         <sourceFile>access.cpl</sourceFile>
         <sourceDir>C:\fuzz\samples\</sourceDir>
     </source>
     <app>
         <appName>rundll32.exe</appName>
         <appDescription></appDescription>
             <appAction>open</appAction>
         <appLaunch>C:\WINDOWS\system32\rundll32.exe</appLaunch>
         <appFlags>shell32.dll,Control_RunDLL "{0}",%* </appFlags>
     </app>
     <target>
         <targetDir>c:\fuzz\cpl\</targetDir>
     </target>
</test>
```

Figure 8.5: The XML configuration file used to configure FileFuzz to fuzz `Access.cpl`

of the data within the file. A binary file represented as hex values may be referred to as a 'hex dump'.

For brevity, we have only shown the first of many lines of hexadecimal values - the file is 68 kB. Figure 8.6 shows a hex dump of the first file created by the fuzzer (`0.cpl`). The first eight bytes (0 - 7) of the original file have been set to zero. Note that the rest of `0.cpl` (bytes 8 onwards) is an exact match of `Access.cpl`.

Figure 8.8 shows a hex dump of `1.cpl`, the second file created by the fuzzer. Here, bytes one to eight of the original file have been set to zero.

Finally, 8.9 shows a hex dump of `2.cpl`, the third file created by the fuzzer. Here, bytes two to nine of the original file have been set to zero.

```
 Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
```

Figure 8.6: The first 15 bytes of `Access.cpl`.

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  00 00 00 00 00 00 00 00 04 00 00 00 FF FF 00 00  ...........ÿÿ..
```

Figure 8.7: The first 15 bytes of `0.cpl`.

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  4D 00 00 00 00 00 00 00 00 00 00 00 FF FF 00 00  M...........ÿÿ..
```

Figure 8.8: The first 15 bytes of `1.cpl`.

We have shown how the first three test cases were created. FileFuzz produced 68,6068 test cases, each time advancing the 8 bytes that were overwritten to zero. After approximately 3 hours, all of the test data had been generated.

8.5 FileFuzz Execution Phase

The settings were entered and the machine was left to complete the testing which took approximately 4 hours.

The output of the FileFuzz Execute module was a log file listing all of the exceptions in the format shown in Figure 8.12.

8.6 Results Analysis

Testing was initially compromised since the fuzzing application was itself subject to a bug described in Appendix 1.

There were a total of 3321 crash reports, all of which were of the type Access Violation.

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000  4D 5A 00 00 00 00 00 00 00 00 00 00 FF FF 00 00  MZ..........ÿÿ..
```

Figure 8.9: The first 15 bytes of `2.cpl`.

```
[*] "crash.exe" C:\WINDOWS\system32\rundll32.exe 200 shell32.dll,Control_RunDLL
"c:\fuzz\cpl\1068.cpl",%*
[*] Access Violation
[*] Exception caught at 00000000 add [eax],al
[*] EAX:0007f234 EBX:80000001 ECX:00000000 EDX:7f6f06de
[*] ESI:58ae12e0 EDI:00000001 ESP:0007f20c EBP:0007f228
```

Figure 8.10: An example crash report from the log file.

Figure 8.11 shows locations where setting values to zero caused a crash, plotted against crash report numbers (a linearly incrementing value). Crash report numbers are represented along the x axis, and the location in `Access.cpl` that was modified, causing a crash is represented along the y axis. This graph illustrates which locations of `Access.cpl` (when set to zero) caused `Rundll32.exe` to crash. Vertical plot angles indicate regions of `Access.cpl` where mutation did not cause `Rundll32.exe` to crash, while horizontal or inclined angles indicate regions where mutation caused `Rundll32.exe` to crash.

There were a total of 28 crash reports where the value of the Extended Instruction Pointer appeared to have been set to the value overwritten by the fuzzer, including the one shown in figure 8.12.

These represent the most significant of the faults generated since user supplied values (even if they take the form of the binary values of a loaded file) should never be allowed to influence the value of EIP. This is because the EIP holds the value of the next instruction to be executed. If this value can be controlled, then program execution can be influenced.

The nature and exploitability of the bug is explored in detail in Chapter 9, *Case Study 2 – Using Fuzzer Output to Exploit a Software Fault.*

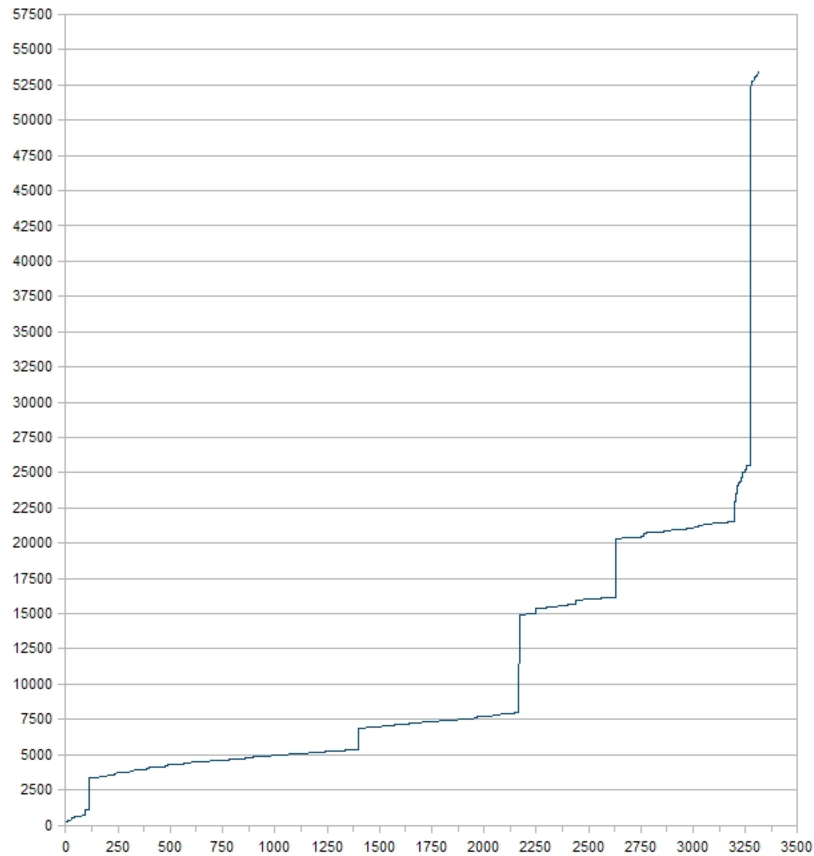


Figure 8.11: A graph based on crash reports from fuzz testing `Rundll32.exe`.

8.7 Lessons Learned

The bug discovered in the fuzzer (described in Appendix 1) raised the author's awareness of the need for independent confirmation of fuzzer output. There is always potential for human error in configuring the fuzzer, just as there is always potential for fuzzers themselves to be subject to software defects. Either way, the author would strongly recommend analysis of fuzzer output via pre-test test runs to confirm the fuzzer is functioning as it should.

Regarding the test configuration, the decision to set the value to overwrite bytes to zero was a poor choice, which hampered fault detection during the results analysis phase. Since this form of fuzzing focuses on the effect of the test data on the registers

```
[*] "crash.exe" C:\WINDOWS\system32\rundll32.exe 200 shell32.dll,Control_RunDLL
"c:\fuzz\cpl\1068.cpl",%*
[*] Access Violation
[*] Exception caught at 00000000 add [eax],al
[*] EAX:0007f234 EBX:80000001 ECX:00000000 EDX:7f6f06de
[*] ESI:58ae12e0 EDI:00000001 ESP:0007f20c EBP:0007f228
```

Figure 8.12: An example crash report from the FileFuzz log file

of the host platform, a very simplistic approach to analysis of the output data (which takes the form of none or more crash reports) is to simply search for instances where the value of the overwritten bytes in the mutated test instances are present in registers at the point when the application crashes.

The reason for this focus is better understood when one sees an example crash report.

From the crash report shown in figure 8.12 we can infer that the Extended Instruction Pointer (EIP) register had been set to 0x00000000. We can say this because the text 'Exception caught at' is followed by the address of the instruction that the processor was attempting to execute at the time of the crash. Indeed, we may infer that the action of attempting to execute an instruction located at memory address 0x00000000 *caused the crash to occur*, since:

1. It is highly unlikely that the processor would try to run an instruction located at the address 0x00000000.

2. It is also unlikely that the process memory space would extend to memory address 0x00000000. If a process attempts to access (i.e. to read from, or write to) memory outside of the bounds of its virtual memory space, this will trigger an access violation fault, causing the process to crash.

3. We also know that the mutated file contained eight bytes that had been overwritten to the value zero.

Hence, it is not unreasonable to propose that overwriting eight consecutive bytes to zero at the location specific to this test instance, and then feeding the test instance to

the target application caused the target application to attempt to execute instructions located at memory address 0x00000000.

Setting the overwritten bytes to the value zero allowed the author to determine that execution redirection was possible by searching for presence of the overwritten value in the EIP register in the crash reports. This relied upon the fact that the EIP very rarely has the value 0x00000000.

Setting an input to a conspicuous value and searching for that value as it propagates through an application is known as *tainting* data.

However, it is common for other registers to have the value 0x00000000; in fact, in the example crash report, the Extended Counter register (ECX) is set to zero. Because this is not unusual, there is no (simple) way to determine whether this is because the test instance has eight bytes written to zero, or if the ECX would be set to zero anyway.

Hence, a better value to overwrite selected locations to would have been something like 0xffffffff, since this is just as likely to cause a crash, whichever register it is passed to, and is not particularly common in any of the registers, so would have allowed detection of control over registers other than the EIP.

CHAPTER

NINE

CASE STUDY 2 – USING FUZZER OUTPUT TO EXPLOIT A SOFTWARE FAULT

The overall objective of this exercise was to test the following hypothesis: by injecting program instructions into a file, and then redirecting the flow of execution (by employing a bug found via fuzzing), a process could be subverted to execute injected code.

In order to test this hypothesis, proof-of-concept software was developed to exploit a previously identified fault in a component of the Windows XP operating system by redirecting program execution to run shell code injected into a file (`Access.cpl`) loaded by the vulnerable component (`Rundll32.exe`).

The original shell code used performed a non-malicious function: the motherboard 'beeper' was activated, and the non-malicious proof-of-concept code was found to successfully exploit the vulnerability in a host running an un-patched Windows XP Professional Service Pack 2 operating system. However, when the host operating system was fully patched (as of February 2008) via Windows Update, the non-malicious proof-of-concept code failed to work.

It was not clear if patching the host operating system had (a) caused the non-malicious shell code to fail (i.e. by relocating the memory location of a function relied upon by the non-malicious shell code), or (b) had addressed the vulnerability by patching `Rundll32.exe`, for example. In order to determine whether the vulnerability had been patched, a second version of the proof-of-concept code was produced, where the non-malicious shell was replaced with an alternative shell code, which included functionality to test the shell code in isolation and determine if the target

was vulnerable (and also contained malicious functionality). Since the fully patched Windows XP host was found to be vulnerable to the malicious shell code in isolation, it could be used to determine if the vulnerability had been patched. The malicious proof-of-concept code was able to exploit the vulnerability in fully patched (as of February 2008) Windows XP Professional systems, indicating that the vulnerability had not been patched.

Though the second proof-of-concept code contained a malicious payload (it binds a command shell to a listening port, essentially opening the compromised host up to remote attacks), the risk to users is low since the host process that is compromised (`Rundll32.exe`) runs with the privileges of the entity that launches the modified file.

Furthermore, this is a wholly local attack, in that it requires that the modified file is placed on the local machine. Mitigating circumstances (in this case the absence of privilege escalation and an absence of network functionality) may mean that it is reasonable for vendors to accept bugs rather than address them. This may seem like a foolish or mercenary strategy to adopt, but it is important to consider that, in the face of a multitude of security vulnerabilities the assignment of resources must be based on the threat level of the vulnerability.

The author reported the bug to the Microsoft Security team[1] and after a period of review they agreed that the bug did not represent a threat to the user community. See appendix 3 for the author's communication with Microsoft.

In order to complete this case study a Personal Computer running Windows XP Service Pack 2 operating system software and the following applications were used:

- HxD, a hexadecimal file editor (hex editor) was used to visually analyse binary files.[2]
- OllyDbg 1.10: a free 32-bit assembler-level analysing debugger with a Graphical User Interface[3]
- Shell code: specially crafted byte code written specifically for the Windows XP Service Pack 2 Operating System with the function of activating the mother board 'beeper'. This was obtained from the milw0rm website.[4]

[1] https://www.microsoft.com/technet/security/bulletin/alertus.aspx
[2] `http://www.mh-nexus.de/hxd/`
[3] `http://home.t-online.de/home/Ollydbg`
[4] `http://www.milw0rm.com/shellcode/1675`

- Shell code: specially crafted byte code written specifically for the Windows XP Service Pack 2 Operating System with the function of binding a command shell to a listening port on the host machine. This was obtained from the Metasploit website which features a modular payload development environment that allows users to create customised shell codes based on requirements.[5] The payload development environment also includes a script that builds four versions of a payload, including a Portable Executable file that will simply launch the payload. This means that payloads can be tested in isolation from injection vectors in order to establish that they will function on a target system.

9.1 Methodology

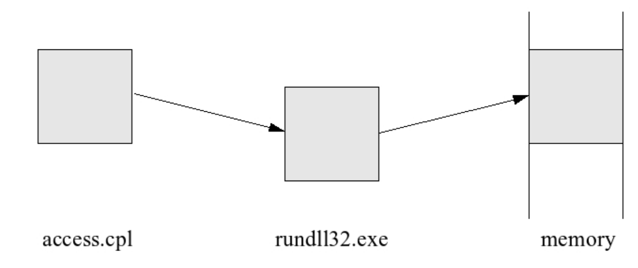
When a user launches Accessibility Services in the Control Panel, `Rundll32.exe` receives `Access.cpl` as input and loads it into memory. If a specific location of `Access.cpl` is modified, execution flow is redirected to the memory address of the value held at the modified location. The approach taken to exploiting `Rundll32.exe` was to insert shell code into the `Access.cpl` file, and redirect EIP to the shell code using the location identified by fuzzing. This approach is illustrated in figure 9.1.

The component tasks were:

1. Obtain suitable shell code.
2. Identify a suitable location within `Access.cpl` where shell code could be placed without causing the host application to crash, and without changing the length of the file.
3. Insert the shell code into the identified area of the .cpl file by overwriting values in the file.
4. Redirect the host application flow to the location where the shell code was to be placed, using the location identified earlier.

[5] `http://www.metasploit.com/shellcode/`

1. Normal operation: access.cpl is loaded into memory



2. Access.cpl is modified to expoit an error in rundll32.exe

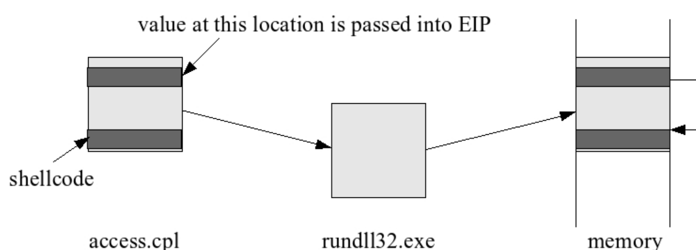


Figure 9.1: The approach taken to exploiting `Rundll32.exe`.

9.1.1 Obtaining the Shell Code

The first shell code was selected from a number of shell codes available at the milw0rm website.[6] Initially, a short (39 byte) non-malicious shell code written for the Windows XP Professional Service Pack 2 operating system that caused the motherboard beeper to sound was selected.

When this was found not to work on fully patched Windows XP Professional Service Pack 2 operating system, a second shell code was obtained this is discussed in more detail in the Results section below.

9.1.2 Identifying a Suitable Location for the Shell Code

After placing shell code into a what appeared to be suitable area (a large number of zeros) located at (decimal) 40,000, or 9c40 (hexadecimal) in `Access.cpl`, the author naively attempted to redirect program flow to 00009C40. This was done by

[6] www.milw0rm.com

overwriting four bytes commencing at memory location 1068 (decimal), or 0000042C (hexadecimal) with the target address of 00009C40. (Note that the actual value that was written was 40 9C 00 00 due to the 'little-endian' interpretation of values into memory addresses common to Windows operating systems).

It had already been identified that an arbitrary value overwritten at locations commencing 1068 (decimal) in `Access.cpl` would be copied into the EIP register (see Chapter 8, *Case Study 1 – 'Blind' Data Mutation File Fuzzing*). Thus, the hypothesis was that the program flow would be redirected to the address set by these values, and that code placed at this address would be executed by the host.

This approach failed in that the motherboard beeper did not sound, indicating that the inserted shell code had not been executed. The author was able to make use of the debugger within FileFuzz to determine that program flow was being directed to 00009C40. This led the author to consider whether and how the `Access.cpl` file was being mapped into memory by the host application (`Rundll32.exe`).

A valid version of `Access.cpl` was launched and OllyDbg was then attached to the process. The Memory Map window of OllyDbg was used to examine how the data held in the .cpl file was mapped into memory when loaded by the `Rundll32.exe` application as shown in figure 9.2, where `Access.cpl` (identified as access-t) is viewed using the OllyDbg debugger mapped into memory into five sections: PE header, text, data, resources and relocations segments.

OllyDbg - rundll32.exe - [Memory map]

M File View Debug Plugins Options Window Help

Address	Size (Decimal)	Owner	Section	Contains
009F0000	00010000 (65536.)	009F0000 (itself)		
00A3E000	00001000 (4096.)	00A30000		
00A3F000	00001000 (4096.)	00A30000		stack of thread 00000E30
00A50000	00002000 (8192.)	00A50000 (itself)		
00A60000	00007000 (28672.)	00A60000 (itself)		
01000000	00001000 (4096.)	rundll32 01000000 (itself)		PE header
01001000	00002000 (8192.)	rundll32 01000000	.text	code, imports
01003000	00001000 (4096.)	rundll32 01000000	.data	data
01004000	00007000 (28672.)	rundll32 01000000	.rsrc	resources
58AE0000	00001000 (4096.)	access-t 58AE0000 (itself)		PE header
58AE1000	00006000 (24576.)	access-t 58AE0000	.text	code, imports, exports
58AE7000	00002000 (8192.)	access-t 58AE0000	.data	data
58AE9000	0000B000 (45056.)	access-t 58AE0000	.rsrc	resources
58AF4000	00001000 (4096.)	access-t 58AE0000	.reloc	relocations
5AD70000	00001000 (4096.)	UxTheme 5AD70000 (itself)		PE header
5AD71000	00030000 (196608.)	UxTheme 5AD70000	.text	code, imports, exports

Figure 9.2: OllyDbg reveals how `Access.cpl` is mapped into memory.

Not only did this make it clear that `Access.cpl` had been mapped into five separate sections (Portable Executable header, text, data, resources and relocations sections), but the addresses range that `Access.cpl` gets mapped into is shown to commence at 58AE0000. Further, by clicking upon any of the sections, the entire section as mapped into memory was viewable (see figure 9.5).

The author chose to insert shell code within the code section on the basis that this area appeared to contain the most op-codes, and it contained a large section at the end of the segment that appeared to be unused, as is indicated by the series of zero valued op-codes commencing at memory address 58AE6625 in figure 9.5.

The next objective was to identify where this area appeared (and if it appeared) in the `Access.cpl` file. If this area could be identified in `Access.cpl`, then the shell code could be inserted here, and would end up mapped into memory, where it could be executed by redirecting EIP to it.

Mapping the relationship between object code address locations in the static `Access.cpl` file and the regions of memory that the `Access.cpl` object code occupied was not trivial. This was not a simple linear offset relationship, since the object code as contained in `Access.cpl` was mapped into five distinct regions as shown in figure 9.2. Fortunately, there were some favourable conditions: memory mapping appeared to be static - i.e. the `Access.cpl` object code was always mapped into the same regions of memory. Additionally, the location mapping relationship only had to be determined between the object code and the chosen code section (the text section) as mapped into memory.

Determining the mapping relationship was achieved by opening `Access.cpl` in a hex editor, and identifying any regions that appeared to contain a large number of zeros, since one of these might be the region of zeros seen in the text section in figure 9.5.

A number of such regions were observed, and the author tainted each of these individually by overwriting zero-ed regions with ASCII A's and B's in differing patterns to aid their identification. In figure 9.4 `Access.cpl` is viewed using a hex editor, showing a zero-ed region that has been over-written with a single ASCII 'A', followed by a string of ASCII 'B's commencing at 00005A40.

The modified version of `Access.cpl` (hereafter referred to as `access-tainted.cpl`) was then launched to see if overwriting the tainted values prevented the .cpl file

```
OllyDbg - rundll32.exe - [Dump - access-t:.text 58AE1000..58AE6FFF]
D File View Debug Plugins Options Window Help
L E M T W H C / K B R ...
58AE65E1  6363 65          ARPL WORD PTR DS:[EBX+65],SP
58AE65E4 v73 73            JNB SHORT access-t.58AE6659
58AE65E6  2E:              PREFIX CS:                              Superfluous prefix
58AE65E7  64:6C            INS BYTE PTR ES:[EDI],DX                I/O command
58AE65E9  6C               INS BYTE PTR ES:[EDI],DX                I/O command
58AE65EA  0043 50          ADD BYTE PTR DS:[EBX+50],AL
58AE65ED  6C               INS BYTE PTR ES:[EDI],DX                I/O command
58AE65EE  41               INC ECX
58AE65EF v70 70            JO SHORT access-t.58AE6661
58AE65F1  6C               INS BYTE PTR ES:[EDI],DX                I/O command
58AE65F2  65:74 00         JE SHORT access-t.58AE65F5              Superfluous prefix
58AE65F5  44               INC ESP
58AE65F6  65:6275 67       BOUND ESI,QWORD PTR GS:[EBP+67]         Superfluous prefix
58AE65FA  4D               DEC EBP
58AE65FB  61               POPAD
58AE65FC  696E 00 446C6C5  IMUL EBP,DWORD PTR DS:[ESI],526C6C44
58AE6603  65:67:6973 74 6  IMUL ESI,DWORD PTR GS:[BP+DI+74],655372
58AE660C v72 76            JB SHORT access-t.58AE6684
58AE660E  65:72 00         JB SHORT access-t.58AE6611              Superfluous prefix
58AE6611  44               INC ESP
58AE6612  6C               INS BYTE PTR ES:[EDI],DX                I/O command
58AE6613  6C               INS BYTE PTR ES:[EDI],DX                I/O command
58AE6614  55               PUSH EBP
58AE6615  6E               OUTS DX,BYTE PTR ES:[EDI]               I/O command
58AE6616 v72 65            JB SHORT access-t.58AE667D
58AE6618  67:6973 74 6572  IMUL ESI,DWORD PTR SS:[BP+DI+74],655372
58AE6620 v72 76            JB SHORT access-t.58AE6698
58AE6622  65:72 00         JB SHORT access-t.58AE6625              Superfluous prefix
58AE6625  0000             ADD BYTE PTR DS:[EAX],AL
58AE6627  0000             ADD BYTE PTR DS:[EAX],AL
58AE6629  0000             ADD BYTE PTR DS:[EAX],AL
58AE662B  0000             ADD BYTE PTR DS:[EAX],AL
58AE662D  0000             ADD BYTE PTR DS:[EAX],AL
58AE662F  0000             ADD BYTE PTR DS:[EAX],AL
58AE6631  0000             ADD BYTE PTR DS:[EAX],AL
58AE6633  0000             ADD BYTE PTR DS:[EAX],AL
58AE6635  0000             ADD BYTE PTR DS:[EAX],AL
58AE6637  0000             ADD BYTE PTR DS:[EAX],AL
58AE6639  0000             ADD BYTE PTR DS:[EAX],AL
58AE663B  0000             ADD BYTE PTR DS:[EAX],AL
58AE663D  0000             ADD BYTE PTR DS:[EAX],AL
58AE663F  0000             ADD BYTE PTR DS:[EAX],AL
58AE6641  0000             ADD BYTE PTR DS:[EAX],AL
58AE6643  0000             ADD BYTE PTR DS:[EAX],AL
58AE6645  0000             ADD BYTE PTR DS:[EAX],AL
58AE6647  0000             ADD BYTE PTR DS:[EAX],AL
58AE6649  0000             ADD BYTE PTR DS:[EAX],AL
58AE664B  0000             ADD BYTE PTR DS:[EAX],AL
58AE664D  0000             ADD BYTE PTR DS:[EAX],AL
```

Figure 9.3: The text section of `Access.cpl` is viewed using OllyDbg.

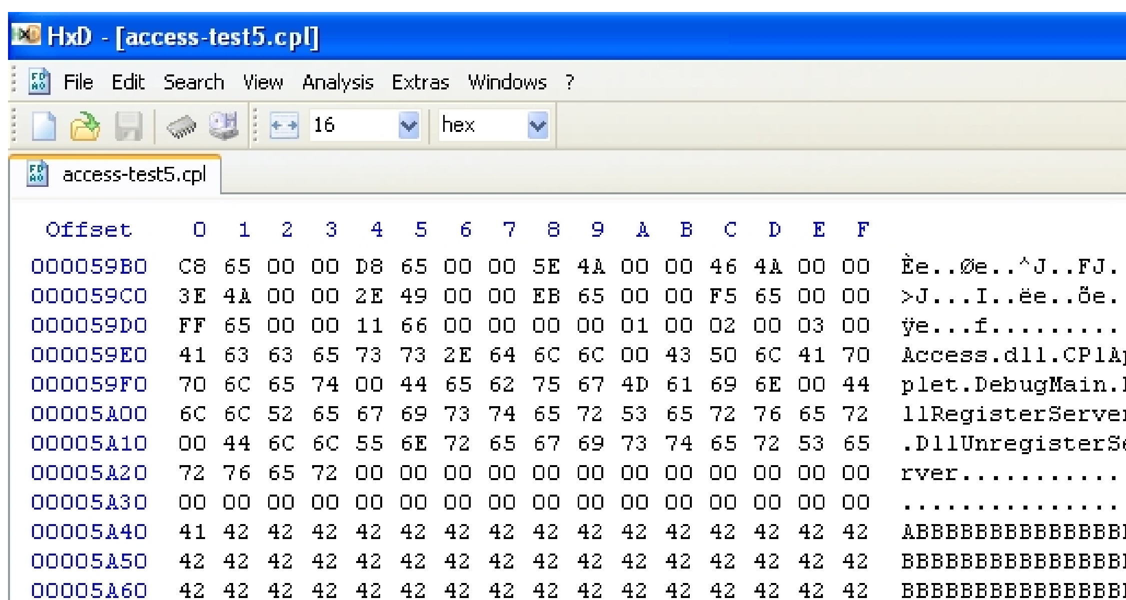


Figure 9.4: A region of `Access.cpl` is 'tainted'.

from launching properly. The tainted file launched, meaning that the tainted regions of `access-tainted.cpl` did not (at least obviously) affect the host program (`Rundll32.exe`) operation. OllyDbg was then launched and attached to the `access-tainted.cpl` file, and the code section was observed to contain a single A (hex 41) followed by a number of B's (hex 42) (see figure 9.5) this meant that the corresponding tainted region in `access-tainted.cpl` could be identified by searching for the same pattern in the `Access.cpl` file and determining the memory location of the fist ASCII 'A' value.

In figure 9.5 the text section of `Access.cpl` is again viewed using Ollydbg. The tainted region of the `Access.cpl` file seen in figure 9.4 has now been mapped into memory. Note the single op code of the value 41 (ASCII 'A'), followed by a string of op codes of the value 42 (ASCII 'B'), commencing at memory address 58AE6642, just as seen in figure 9.4.

By identifying the address to which a tainted region of `access-tainted.cpl` had been mapped to memory, the author had identified an address to place the shell code into `Access.cpl`: (hex) 00005A41, and that by redirecting EIP to the corresponding memory-mapped address (58AE6642), the shell code could be executed.

```
OllyDbg - rundll32.exe - [Dump - access-t:.text 58AE1000..58AE6FFF]
D File View Debug Plugins Options Window Help
L E M T W H C / K B R ...
58AE65E9   6C              INS BYTE PTR ES:[EDI],DX                  I/O command
58AE65EA   0043 50         ADD BYTE PTR DS:[EBX+50],AL
58AE65ED   6C              INS BYTE PTR ES:[EDI],DX                  I/O command
58AE65EE   41              INC ECX
58AE65EF  v70 70           JO SHORT access-t.58AE6661
58AE65F1   6C              INS BYTE PTR ES:[EDI],DX                  I/O command
58AE65F2   65:74 00        JE SHORT access-t.58AE65F5                Superfluous prefix
58AE65F5   44              INC ESP
58AE65F6   65:6275 67      BOUND ESI,QWORD PTR GS:[EBP+67]           Superfluous prefix
58AE65FA   4D              DEC EBP
58AE65FB   61              POPAD
58AE65FC   696E 00 446C6C5 IMUL EBP,DWORD PTR DS:[ESI],526C6C44
58AE6603   65:67:6973 74 6 IMUL ESI,DWORD PTR GS:[BP+DI+74],655372
58AE660C  v72 76           JB SHORT access-t.58AE6684
58AE660E   65:72 00        JB SHORT access-t.58AE6611                Superfluous prefix
58AE6611   44              INC ESP
58AE6612   6C              INS BYTE PTR ES:[EDI],DX                  I/O command
58AE6613   6C              INS BYTE PTR ES:[EDI],DX                  I/O command
58AE6614   55              PUSH EBP
58AE6615   6E              OUTS DX,BYTE PTR ES:[EDI]                 I/O command
58AE6616  v72 65           JB SHORT access-t.58AE667D
58AE6618   67:6973 74 6572 IMUL ESI,DWORD PTR SS:[BP+DI+74],655372
58AE6620  v72 76           JB SHORT access-t.58AE6698
58AE6622   65:72 00        JB SHORT access-t.58AE6625                Superfluous prefix
58AE6625   0000            ADD BYTE PTR DS:[EAX],AL
58AE6627   0000            ADD BYTE PTR DS:[EAX],AL
58AE6629   0000            ADD BYTE PTR DS:[EAX],AL
58AE662B   0000            ADD BYTE PTR DS:[EAX],AL
58AE662D   0000            ADD BYTE PTR DS:[EAX],AL
58AE662F   0000            ADD BYTE PTR DS:[EAX],AL
58AE6631   0000            ADD BYTE PTR DS:[EAX],AL
58AE6633   0000            ADD BYTE PTR DS:[EAX],AL
58AE6635   0000            ADD BYTE PTR DS:[EAX],AL
58AE6637   0000            ADD BYTE PTR DS:[EAX],AL
58AE6639   0000            ADD BYTE PTR DS:[EAX],AL
58AE663B   0000            ADD BYTE PTR DS:[EAX],AL
58AE663D   0000            ADD BYTE PTR DS:[EAX],AL
58AE663F   0041 42         ADD BYTE PTR DS:[ECX+42],AL
58AE6642   42              INC EDX
58AE6643   42              INC EDX
58AE6644   42              INC EDX
58AE6645   42              INC EDX
58AE6646   42              INC EDX
58AE6647   42              INC EDX
58AE6648   42              INC EDX
58AE6649   42              INC EDX
58AE664A   42              INC EDX
58AE664B   42              INC EDX
58AE664C   42              INC EDX
58AE664D   42              INC EDX
58AE664E   42              INC EDX
```

Figure 9.5: A tainted region of `Access.cpl` has been mapped into memory.

9.1.3 Inserting Shell Code Into `Access.cpl`

Using a hex editor, the shell code was written over the identified region of a copy of `access-tainted.cpl`, starting at (hex) address 00005A41.

Having pasted the shell code into `access-tainted.cpl`, the file (hereafter referred to as `access-shellcode.cpl`) was launched to ensure that the shellcode insertion did not disrupt the host program. Figure 9.6 illustrates that the shellcode could be observed to be resident in memory commencing at the expected location: 58AE6642.

9.1.4 Redirecting Execution Flow to Execute the Shellcode

The final stage was to redirect the host application flow to the location where the shellcode was to be placed, using the location identified earlier. By fuzzing the host application, we learned that an arbitrary value could be placed into EIP by setting four consecutive bytes at a specific location within `Access.cpl` namely (decimal) 1086.

By over-writing the values at (decimal) 1086 with the address of the first byte of where the shellcode would be located *when mapped into memory*, it was intended that the flow of execution would be diverted from the application to the shellcode.

If launching the application caused the shellcode to be run, this would prove that the discovered vulnerability was exploitable.

Using a hex editor a copy of `access-shellcode.cpl` (hereafter known as access-exploit.cpl) was modified such that starting at location 1086, four consecutive bytes were set to 42, 66, AE, 58. The file was saved.

9.2 Results

The altered version of `Access.cpl` was launched. A beep sounded, (proving the hypothesis that the injected shellcode could be run) and the program then crashed.

The author then set out to determine if a fully patched version of Windows XP service pack 2 was similarly vulnerable.

```
OllyDbg - access-test6.cpl - [Dump - access-t:.text 58AE1000..58AE6FFF]
D File View Debug Plugins Options Window Help

58AE65E6  2E:              PREFIX CS:                                Superfluous prefix
58AE65E7  64:6C            INS BYTE PTR ES:[EDI],DX                  I/O command
58AE65E9  6C               INS BYTE PTR ES:[EDI],DX                  I/O command
58AE65EA  0043 50          ADD BYTE PTR DS:[EBX+50],AL
58AE65ED  6C               INS BYTE PTR ES:[EDI],DX                  I/O command
58AE65EE  41               INC ECX
58AE65EF v70 70            JO SHORT access-t.58AE6661
58AE65F1  6C               INS BYTE PTR ES:[EDI],DX                  I/O command
58AE65F2  65:74 00         JE SHORT access-t.58AE65F5                Superfluous prefix
58AE65F5  44               INC ESP
58AE65F6  65:6275 67       BOUND ESI,QWORD PTR GS:[EBP+67]           Superfluous prefix
58AE65FA  4D               DEC EBP
58AE65FB  61               POPAD
58AE65FC  696E 00 446C6C5  IMUL EBP,DWORD PTR DS:[ESI],526C6C44
58AE6603  65:67:6973 74 6  IMUL ESI,DWORD PTR GS:[BP+DI+74],655372
58AE660C v72 76            JB SHORT access-t.58AE6684
58AE660E  65:72 00         JB SHORT access-t.58AE6611                Superfluous prefix
58AE6611  44               INC ESP
58AE6612  6C               INS BYTE PTR ES:[EDI],DX                  I/O command
58AE6613  6C               INS BYTE PTR ES:[EDI],DX                  I/O command
58AE6614  55               PUSH EBP
58AE6615  6E               OUTS DX,BYTE PTR ES:[EDI]                 I/O command
58AE6616 v72 65            JB SHORT access-t.58AE667D
58AE6618  67:6973 74 6572  IMUL ESI,DWORD PTR SS:[BP+DI+74],655372
58AE6620 v72 76            JB SHORT access-t.58AE6698
58AE6622  65:72 00         JB SHORT access-t.58AE6625                Superfluous prefix
58AE6625  0000             ADD BYTE PTR DS:[EAX],AL
58AE6627  0000             ADD BYTE PTR DS:[EAX],AL
58AE6629  0000             ADD BYTE PTR DS:[EAX],AL
58AE662B  0000             ADD BYTE PTR DS:[EAX],AL
58AE662D  0000             ADD BYTE PTR DS:[EAX],AL
58AE662F  0000             ADD BYTE PTR DS:[EAX],AL
58AE6631  0000             ADD BYTE PTR DS:[EAX],AL
58AE6633  0000             ADD BYTE PTR DS:[EAX],AL
58AE6635  0000             ADD BYTE PTR DS:[EAX],AL
58AE6637  0000             ADD BYTE PTR DS:[EAX],AL
58AE6639  0000             ADD BYTE PTR DS:[EAX],AL
58AE663B  0000             ADD BYTE PTR DS:[EAX],AL
58AE663D  0000             ADD BYTE PTR DS:[EAX],AL
58AE663F  0041 55          ADD BYTE PTR DS:[ECX+55],AL
58AE6642  89E5             MOV EBP,ESP
58AE6644  83EC 18          SUB ESP,18
58AE6647  C745 FC 538A837  MOV DWORD PTR SS:[EBP-4],kernel32.Beep
58AE664E  C74424 04 D0030  MOV DWORD PTR SS:[ESP+4],3D0
58AE6656  C70424 010E0000  MOV DWORD PTR SS:[ESP],0E01
58AE665D  8B45 FC          MOV EAX,DWORD PTR SS:[EBP-4]
58AE6660  FFD0             CALL EAX
58AE6662  C9               LEAVE
58AE6663  C3               RETN
58AE6664  0000             ADD BYTE PTR DS:[EAX],AL
58AE6666  0042 42          ADD BYTE PTR DS:[EDX+42],AL
58AE6669  42               INC EDX
58AE666A  42               INC EDX
58AE666B  42               INC EDX
58AE666C  42               INC EDX
```

Figure 9.6: Shell code is inserted into the text section of `Access.cpl`.

Using the Windows Update service (which can identify and apply all of the updates required to fully patch the Windows Operating System) the workstation was fully patched. Launching the `access-shellcode.cpl` an error was shown, but no beep was audible, suggesting that some aspect of the exploit had been 'broken' by applying the security patches.

There were at least two possible causes for the failure of the exploit to function: the operation of the injection vector had been disrupted, or the operation of the exploit payload had been disrupted. In order to establish whether the payload or the injection vector had been addressed by the security patches, a number of alternative shell codes were obtained from the Metasploit project website.

The shell codes obtained from the Metasploit website came in the form of a modular development kit, which meant that in addition to obtaining the byte code sequence of a shell code, the user is also able to generate a Portable Executable file which will launch the shellcode.

This meant that the shell codes could be tested in isolation to determine if the fully patched operating system was susceptible to any of them. A shell code was identified that the fully patched operating system was susceptible to, named "`win32_stage_boot_bind_shell`". Since this shellcode was proven to work, if it was injected via `Access.cpl` and successfully exploited the operating system, this would prove that the injection vector was valid for fully patched Windows XP Operating System. The new shellcode was inserted into `Access.cpl`, commencing at the same point as the previous shellcode.

The method used to determine if the shellcode had exploited the vulnerability was to use the `netstat -an` command which can reveal the status of network ports. The literature that came with the shell code stated that it would bind an interactive command shell to port number 8721.

Before running the modified version of `Access.cpl`, the `netstat -an` command was used to determine the status of any ports. Figure 9.7 shows the output of the `netstat -an` command prior to launching the modified version of `Access.cpl`: a number of ports are in a listening state. The unusual colour scheme is due to the author inverting the colours of the image to reduce printer ink usage.

The modified version of `Access.cpl`, was launched. There were no visible effects. Like the first modified version of `Access.cpl`, the Accessibility GUI that should

```
C:\WINDOWS\system32\cmd.exe

C:\Documents and Settings\tc>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:25             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:80             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:443            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1025           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1433           0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1026         0.0.0.0:0              LISTENING
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:500            *:*
  UDP    0.0.0.0:1434           *:*
  UDP    0.0.0.0:3456           *:*
  UDP    0.0.0.0:4500           *:*
  UDP    127.0.0.1:123          *:*
  UDP    127.0.0.1:1900         *:*

C:\Documents and Settings\tc>
```

Figure 9.7: The `netstat` command is used to determine the status of network ports.

appear when a normal version of `Access.cpl` is launched did not appear. Unlike the first modified version of `Access.cpl`, no error messages were generated. It was as though the application had not been launched.

The `netstat -an` command was used again. Figure 9.8 shows the output: Port 8721 was now listening.

Task Manager was invoked to determine if `Rundll32.exe` was running. Figure9.9 shows the result: it was. Additionally, it could be seen that `Rundll32.exe` was associated with the user name ‘tc’: the user account that was used to launch the modified version of `Access.cpl`.

In order to confirm that the shellcode was running, the account was switched from ‘tc’ (an administrator account), to ‘user99’: a restricted user account. Telnet was launched from the command window as follows `telnet 127.0.0.1 8721`. The result was a command prompt with the path of the folder from where the modified version of `Access.cpl` was launched. The author was able to browse to the root directory using the `cd /` command.

In order to test the privilege level of the command prompt, the author launched an explorer window, browsed to the C:/Windows/repair directory and attempted to copy

```
C:\WINDOWS\system32\cmd.exe
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\tc>netstat -an

Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    0.0.0.0:25             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:80             0.0.0.0:0              LISTENING
  TCP    0.0.0.0:135            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:443            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:445            0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1025           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:1433           0.0.0.0:0              LISTENING
  TCP    0.0.0.0:8721           0.0.0.0:0              LISTENING
  TCP    127.0.0.1:1026         0.0.0.0:0              LISTENING
  UDP    0.0.0.0:445            *:*
  UDP    0.0.0.0:500            *:*
  UDP    0.0.0.0:1434           *:*
  UDP    0.0.0.0:3456           *:*
  UDP    0.0.0.0:4500           *:*
  UDP    127.0.0.1:123          *:*
  UDP    127.0.0.1:1900         *:*

C:\Documents and Settings\tc>
```

Figure 9.8: The `netstat` command is used to reveal a new service is listening on port 8721.

the `security` file. This operation was denied with an Access Denied message. The author then browsed to the C:/Windows/repair directory using the command prompt and was able to successfully copy the `security` file, proving the command prompt presented by the shellcode inherited the privileges of the account that launched it.

9.3 Conclusions

The hypothesis, that the output of fuzz testing could be used as a means to develop an exploit by combining an ‘off the shelf’ payload with a discovered injection vector (in this case a Read Access Violation on the Extended Instruction Pointer) was confirmed.

Even though the injection vector used is of simplest type to exploit: one simply places a payload and directs EIP to it, the author did not expect to be successful in creating exploit code. The overall experience was disturbing: the author had no experience in exploit development, yet the discovery of the injection vector was trivial, and largely automated, and the ease of integration of the payload, and the potency of the malicious payload was chilling.

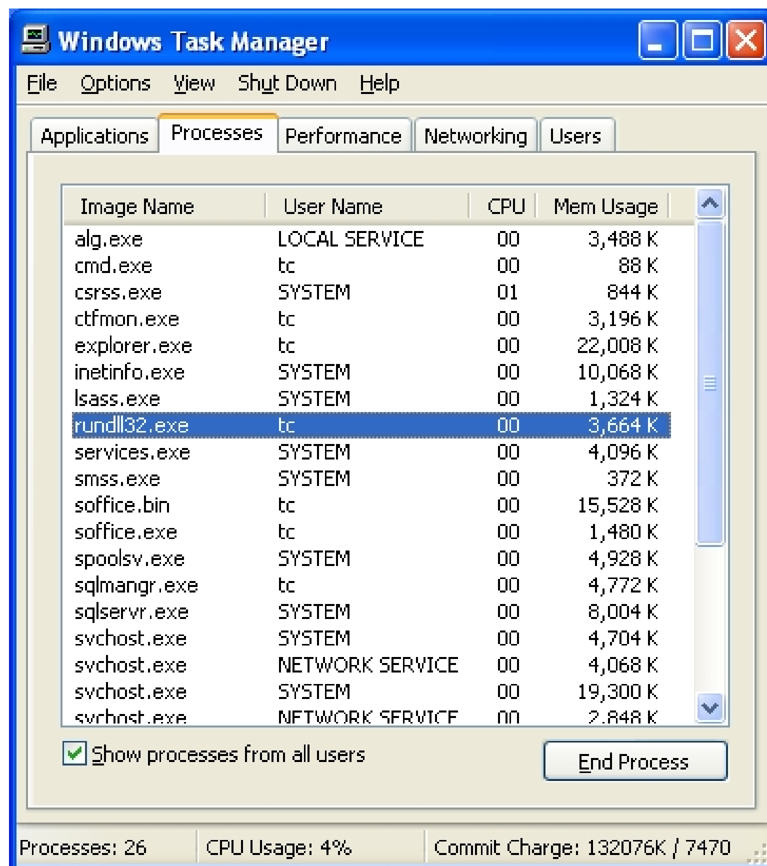


Figure 9.9: Task Manager is used to determine what processes are running.

The author has given some thought to whether this case study should be shared. It might be argued that this information could be used to assist malicious parties to generate malicious software. Regarding the fact that information is divulged; specifically, the presence of injection vectors and the actual memory locations of these vectors in `Access.cpl`, I would say the following: the free availability of malicious shellcode payloads, particularly those that come with a development environment that can generate a standalone Portable Executable file which will launch the shellcode when double-clicked, offer as much threat as the proof of concept code created as a part of this report. When launched this way, the security context of the shellcode is inherited from the user that launches it, and the same is true of any file that exploits `Rundll32.exe`.

It might seem concerning that both I and Microsoft Security Response team would play down the risks around this vulnerability, but, since resources are precious and vulnerabilities are common, we must place risks in context. Against the backdrop of a multitude of unpatched vulnerabilities that offer privilege escalation and/or remote code execution, this vulnerability is minor to the point of irrelevance.

CHAPTER

TEN

PROTOCOL ANALYSIS FUZZING

"If I had eight hours to chop down a tree, I'd spend six sharpening my axe."

Abraham Lincoln

We have seen the benefits and limitations of zero-knowledge testing. In this chapter we will examine protocol analysis fuzzing, an approach that solves many of the problems faced by zero-knowledge testing, but requires more intelligence and effort on the part of both the tester and the fuzzer. It also takes longer to initiate, but can result in more efficient test data, which means test runs may be shorter and more effective at discovering defects.

Protocol testing involves leveraging an understanding of the protocols and formats that define data received by the target application in order to 'intelligently' inform test data generation. In the authors opinion there are two key requirements for intelligent fuzzing:

1. Protocol structure and stateful message sequencing: the ability to define a grammar which describes legal message types and message sequencing rules.

2. Data element isolation, also termed *tokenisation*, the ability to decompose a message into individual data elements and an associated capacity to mutate and modify data elements in isolation and with reference to their type.

Together, these two features allow a fuzzer to generate test data based on the effective input space of the application. This chapter will cover these two requirements, and how they are realised, in detail.

Protocol analysis may still be thought of as 'black box' testing since no specific knowledge of the inner functioning of the *target* is required. This approach to testing has been termed *protocol implementation testing*, since the subject of such testing is really the mechanism by which an application implements a protocol in order to process received data: a combination of *demarshaling* (essentially unpacking the data stream) and *parsing* (separating the received data into individual components).

Protocol testing can result in the production of efficient test data with the potential to exercise a greater percentage of the target application code than that produced by zero knowledge methods. This is because it can be used to create test data that maps to the effective input space and will penetrate deeper into application states, past static numbers, self-referring checks, past structure requirements, and even through many levels of protocol state via grammar based message sequencing.

However, there is a price to be paid for the benefits of protocol testing: protocol analysis and the development of a protocol-aware fuzzer require considerably more effort than any of the zero-knowledge approaches. In order to understand how this works and why this approach is worth the additional effort required, we must explore the nature of protocols and how their implementation may impact on software security.

Two key contributors to the development of intelligent fuzzing frameworks are Dave Aitel, who developed the SPIKE fuzzing framework, and Rauli Kaksonen and his colleagues at the University of Oulu, where the PROTOS suite of protocol implementation testing tools were developed. Both SPIKE and the PROTOS suite offer intelligent, protocol aware fuzzer development environments which permit testers to create fuzzers that leverage understanding of a protocol and also permit amortisation of fuzzer development investment across multiple targets.

10.1 Protocols and Contextual Information

Protocols allow independent parties to agree on the format of information prior to its exchange. This is important since formatting can be used to provide context, without which, data is meaningless. Contextual information allows raw data to be interpreted as information. The process of extracting meaning (i.e. information) from symbols (i.e. data) is termed *semantics*, and a protocol may be defined by a collection of semantic rules.

The importance of protocols for application security testing is that the widespread adoption of common protocols for processes such as serialization for data exchange makes analysis of data formats possible, even when undocumented, proprietary formats are used.

10.2 Formal Grammars

A formal grammar defines a finite set and sequence of symbols which are valid for a given language, based on the symbols and their location alone. No meaning need be inferred in order to determine whether a phrase is grammatically correct.

> *"For each grammar, there are generally an infinite number of linear representations (sentences) that can be structured with it. That is, a finite-size grammar can supply structure to an infinite number of sentences. This is the main strength of the grammar paradigm and indeed the main source of the importance of grammars: they summarize succinctly the structure of an infinite number of objects of a certain class."* [18, p. 13]

The value of an awareness of a particular protocol grammar for testing is that it may be employed to identify a grammatically correct base data construct, which can then be mutated. This reduces the task of a fuzzer to the mutation of the (potentially infinite) absolute input space to the mutation of the effective input space. A high level of conformance of the test data to the target application input specification will lead to high code coverage rates as less test data is rejected at the unmarshalling stage, allowing it to reach and test the parser. Of course, complete conformance is not the objective, since the test data must deviate from that 'expected' by the application. Rather: a grammar capable of modelling valid data is used as the basis for mutation.

10.3 Protocol Structure and Stateful Message Sequencing

Kaksonen states: *"For creation of effective test cases for effective fault injection the semantics of individual messages and message exchanges should be preserved."* [25, p. 3]. The semantics of message exchanges can be thought of as protocol *state*.

Protocols may be stateful or stateless. When testing stateless protocols, each test case is simply injected into the process. In order to test stateful protocols, the fuzzer must account for protocol state in order to reach 'embedded' states and exercise all of the protocol functionality. An example would be Transport Control Protocol (TCP). In order to properly test a server implementation of TCP, the fuzzer would need to be capable of establishing the various TCP states.

Protocol state is best represented in a graphical form. Such a graph can be "walked" by the fuzzer so that all states are enumerated. Furthermore, messages can be fuzzed in isolation such that the validity of all other messages is maintained, accessing 'deep' states in the target application and resulting in high code coverage.

In figure 10.1 we see the message sequencing format of the Trivial File Transfer Protocol described using production rules of the Backus Nuar Form (BNF)[1] context-free grammar. The same message sequencing may also interpreted to produce a graph (which Kaksonen terms a *simulation tree*) describing the TFTP protocol state. Figure 10.2 shows a simulation tree of the Trivial File Transfer Protocol (TFTP).

Kaksonen describes many different formal grammars for describing protocol state including *Backus Nuar Form*, *Specification and Description Language*, *Message Sequence Chart*, and *Tree and Tabular Combined Notation* [24]. However, the method used to define protocol message sequencing is not important, as long as it can be done.

The Sulley fuzzing framework facilitates message sequence definition within test *sessions*. An example below, taken from [46] shows the how message sequencing of the Simple Mail Transfer Protocol (SMTP) can be defined within a test session, and figure 10.3 provides a graphical representation of the result.

```
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"))
```

(The above is taken from [46].)

[1] *"[BNF] is used to formally define the grammar of a language, so that there is no disagreement or ambiguity as to what is allowed and what is not"*[15]

```
# Request PDUs
<RRQ> ::= (0x00 0x01) <FILE-NAME> <MODE>
<WRQ> ::= (0x00 0x02) <FILE-NAME> <MODE>

# Subsequent PDUs
<BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 512 x <OCTET>
<LAST-BLOCK> ::= (0x00 0x03) <BLOCK-NUMBER> 0..511 { <OCTET> }
<ACK>  ::= (0x00 0x04) <BLOCK-NUMBER>
<ERROR> ::= (0x00 0x05) <ERROR-CODE> <ERROR-MESSAGE>

# Miscellaneous productions
<MODE> ::= "octet" 0x00 |"netascii" 0x00
<FILE-NAME> ::= { <CHARACTER> } 0x00
<BLOCK-NUMBER> ::= <OCTET> <OCTET>
<ERROR-CODE> ::= <OCTET> <OCTET>
<ERROR-MESSAGE> ::= { <CHARACTER> } 0x00
<CHARACTER> ::= 0x01 - 0x7f
<OCTET> ::= 0x00 - 0xff
```

Figure 10.1: The message formats of the Trivial File Transfer Protocol using Backus Nuar Form [24, p. 62].

We have seen how a fuzzer can be made ‘protocol-aware’ using formal grammars to describe message sequencing. This means that a fuzzer can ‘walk’ an implementation of a protocol through it’s various states. It also means that, once each state is reached, the fuzzer can then apply a range of input values to that particular state. However, we have not yet covered how we might define rules for what values to pass into the application. This is the second key requirement for intelligent fuzzing: identification and isolation of data elements, or *tokenisation*.

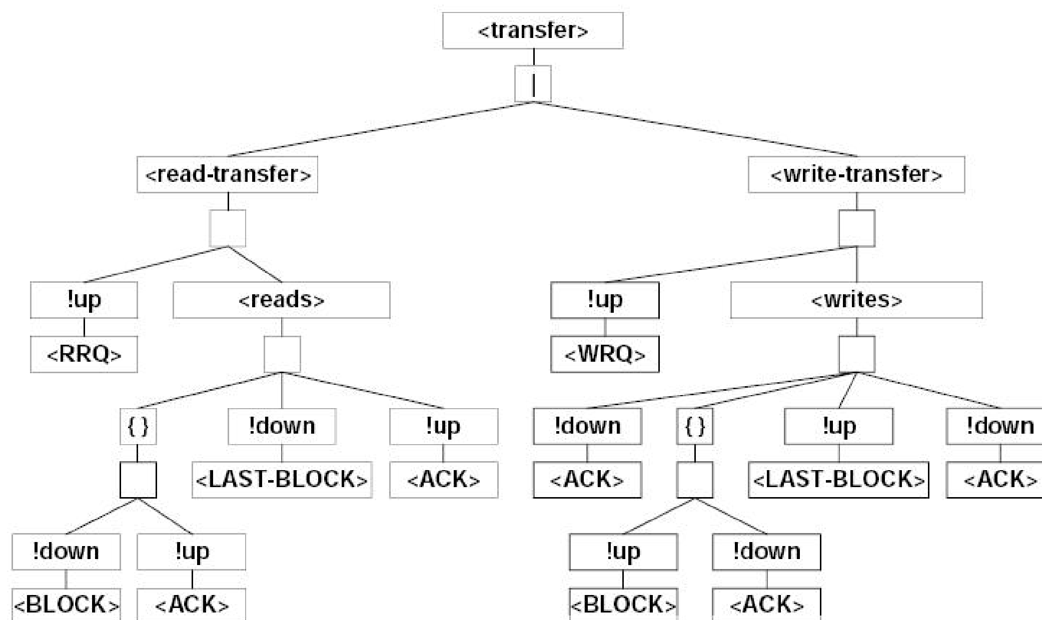


Figure 10.2: A PROTOS simulation tree for Trivial File Transfer Protocol without error handling [24, p. 61].

10.4 Tokenisation

Up to this point we have examined protocols at the message level. Yet, messages are composed of one or many data elements. *Tokenisation*, the process of breaking down a protocol into specific data element types and identifying those types, allows a protocol fuzzer to apply intelligent fuzz heuristics to individual data types: *string* elements may be fuzzed with a string library, while *byte* elements may be fuzzed with a different heuristics library. Furthermore, derived data types, once identified as such, can be dynamically recalculated to match fuzzed raw data values, or fuzzed using an appropriate fuzz heuristic library.

10.4.1 Meta Data and Derived Data Elements

The data that applications receive is often a combination of raw data, contextual or *Meta* data and derived data.

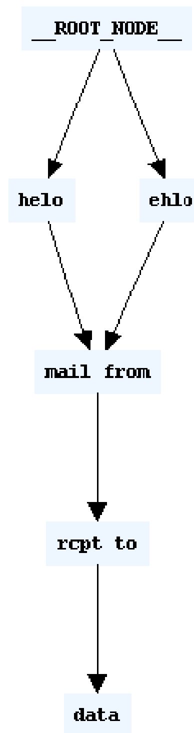


Figure 10.3: A graphical representation of the SMTP protocol message sequencing as defined in a Sulley test session [46].

An Analogy

> *A man walks up to a stranger and says "22 A Stanmore Place Bridge End QR7 62Y." The stranger walks away, baffled. The man approaches another stranger and says "Excuse me, could you give me directions to get to Stanmore Place?" This time the stranger responds: "Certainly..." and proceeds to provide directions.*

The man actually provides *less* factual information in the second scenario than in the first, yet achieves a far better result. This is due to the addition of *Meta data* (in this case "could you give me directions to get to...") which provides contextual information for the receiving party, allowing them to attach meaning to raw data (in this case "22 A Stanmore Place...").

Humans are very good at inferring context from very limited Meta data. In human communication contextual information can take many forms such as facial expression, body language and vocal tone. Each of these represents a separate channel for information to be conveyed. In contrast, computers have significantly fewer sources for inferring context, and often employ serial communications where raw data must be multiplexed with contextual Meta data. In order for machines to communicate, contextual information is usually pre-agreed in the form of standardised protocols. Protocols are not only required for network data exchanges (network protocols), but also for inter- and intra- process communication and data storage and retrieval (binary protocols).

It's worth noting that whenever data channels are combined, (i.e. control data is sent with raw data) there is an associated risk of 'channel issues', [39, p. 8] where an attacker can tamper with existing data in transit or creates malicious data that abuses privileges assigned to control data. Hence, recipients of such data should always sanitize or validate control data before acting on it.

If two parties agree on a protocol for data exchange, the Meta data that provides the required contextual information can be pre-agreed and does not need to be sent with the raw data. Alternatively, the protocol may specify some Meta data be sent along with the raw data. In order to implement either approach, each instance of raw data (which we will term an *element*), needs to be separated in some way from other elements, so that elements can be gathered together, transmitted, and then differentiated post-reception.

10.4.2 Separation of Data Elements

Separation of elements can be achieved by *positional information* (the serial position of the data may be used to infer pre-agreed contextual information), or by the use of positional markers, termed *delimiters*.

Positional separation of elements requires that fixed length fields are assigned to store specific data elements, while delimiters may be used to separate variable length fields. A positional, fixed field approach may result in data expansion unless data elements are of uniform length, since variation of element length must be accommodated in such systems by 'padding out' elements with null data to fit their specified fields.

Delimited, variable length methods mean that fields can precisely match element sizes, but add complexity (and risk due to channelling issues) since delimiter characters must be exclusive to avoid inadvertent termination of a field.

An example of a fixed length protocol is the Internet Protocol Version 4 (IPv4) protocol, which has many fixed length fields. The Hypertext Transfer Protocol (HTTP), on the other hand, contains many delimited, variable length fields.

There are many other factors that influence the nature of a protocol, for example: it is very common for protocols to be aligned along 32 bit binary words in order to optimise processing performance [46, p. 47].

Since many applications have a need to collect data (known as *marshalling*) and *serialize* data in order to transfer it, standards for collection and transfer have been widely adopted.

10.4.3 Serialization

The process of gathering, preparing and transmitting data objects over a serial interface such as a network socket, or in preparation for storage is termed *serialization*. Data objects are collapsed or *deflated* into their component fields [1]. *Data marshalling* is the process whereby objects that are to be transferred are collected, deflated and serialized into a buffer in preparation to be transferred across the application domain boundary and deserialized in another domain [1].

Due to the commonplace need for serialization, almost all development environments include resources to support it, such as formatter objects in C # 2005 which convert objects for serialisation [1]. Furthermore, since many applications have a requirement for serialisation, common approaches have been taken [4, p. 4]. Proprietary implementations often employ a common transfer syntax such as Type, Length, Value [22], in order to implement standards such as the Basic Encoding Rules (BER), and Distinguished Encoding Rules (DER) which themselves fall under Abstract Syntax Notation One (ASN.1).

For serialization to be interoperable across diverse operating system and processor architectures, a degree of abstraction must be implemented. For example, different processor architectures have different byte ordering systems. Hence, abstract transfer syntaxes have been developed that can be used to describe data elements in a platform-neutral manner [1].

10.4.4 Parsing

> *"A parser breaks data into smaller elements, according to a set of rules that describe its structure."* [2]

Parsers apply a grammatical rule set (termed production rules) to process input data, in order to identify individual data element values and types and create specific instances of data structures from general definitions, and populate those instances with specific values.

> *"Parsing is the process of matching grammar symbols to elements in the input data, according to the rules of the grammar. The resulting parse tree is a mapping of grammar symbols to data elements. Each node in the tree has a label, which is the name of a grammar symbol; and a value, which is an element from the input data."* [2]

Parsers derive contextual information and add meaning to data, and also to some extend validate data. Typically, information arriving at an application input boundary point will be demarshalled, and then directed to a parser for analysis.

10.4.5 Demarshalling and Parsing in Context

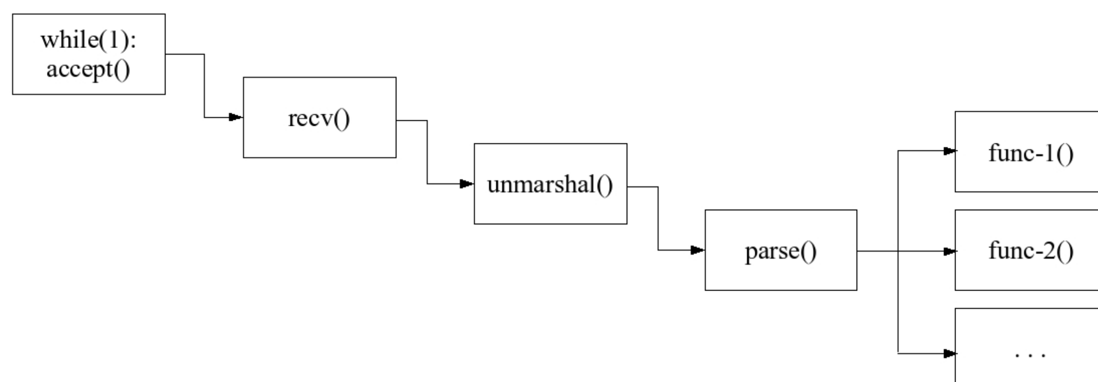


Figure 10.4: High-level control flow of a typical networked application [46, p. 307].

Figure 10.4 shows the high-level control flow of a typical networked application. Sutton et al.'s description (below) and the above diagram help to place unmarshalling and parsing in the context of a common application's operation.

> *"A loop within the main thread of our example target application is awaiting new client connections. On receiving a connection, a new thread is spawned to process the client request, which is received through one or more calls to* `recv()`*. The collected data is then passed through some form of unmarshalling or processing routine. The* `unmarshal()` *routine might be responsible for protocol decompression or decryption but does not actually parse individual fields within the data stream. The processed data is in turn passed to the main parsing routine* `parse()`*, which is built on top of other routines and library calls. The* `parse()` *routine processes the various individual fields within the data stream, taking the appropriate requested actions before finally looping back to receive further instructions from the client."* [46, p. 306]

Unmarshalling and parsing routines may be thought of as the point 'where the rubber meets the road' in terms of application input processing. If these routines are not designed or implemented correctly, they represent a significant risk to application security. Both unmarshalling and parsing rely upon standardised transfer syntax protocols such as Abstract Syntax Notation One.

10.4.6 Abstract Syntax Notation One

Abstract Syntax Notation One (ASN.1) may be defined as follows:

> *"[...] a formal notation used for describing data transmitted by telecommunications protocols, regardless of language implementation and physical representation of these data, whatever the application, whether complex or very simple."* [2]

ASN.1 encompasses many different approaches to encoding data for transfer, one of which is termed Basic Encoding Rules (BER).

[2] `http://asn1.elibel.tm.fr/en/introduction/index.htm`

10.4.7 Basic Encoding Rules

As already stated, it is possible to include some Meta data with raw data. An example of this is a very common binary format termed Type, Length, Value, which is part of the Basic Encoding Rules. Here, two Meta data elements *Type* and *Length* precede a raw data element: *Value*. The Meta data element *Type* is a numerical value which corresponds to a lookup table of acceptable data types, while the Meta data element *Length* is derived from the length of the raw data element, which is held in the *Value* element. The *Type* Meta data element provides contextual data, while the *Length* Meta data is derived from an attribute of the raw data element.

This example shows all three possible elements, a raw element, a contextual Meta data element and a derived Meta data element:

02 – tag indicating INTEGER (contextual Meta data)
01 – length in octets (derived Meta data)
05 – value (raw data element)

10.4.8 Fuzzing Data Elements in Isolation

If a fuzzer is capable of data element identification and isolation, it can mutate elements individually, in isolation, so ensuring a high degree of compliance with the base protocol, and it can also apply type-awareness to fuzz elements based on their type, using intelligently selected heuristics to reduce the test data range.

We have identified three different types of data element that an application might expect to receive. Kaksonen defines an additional element termed an *exception* element specifically for the purposes of fault injection. This term describes a malformed element that is used to replace one of the three expected elements in order to induce a failure state [25, p. 3]. This exception element could be a heuristic, a randomly selected value, or a brute force range.

Let us consider the effect of mutating data elements individually, using Type Length Value as an example:

1. Mutating Meta data *type* elements may cause the incorrect context to be applied to data: e.g. a DWORD (typically a 32 bit data type) could be treated as a

byte (an 8 bit data type), likely causing truncation. This could lead to under or over runs, where too much or too little memory space is allocated to hold input data. These types of potential errors indicate the need for data sanitization checks such as buffer bounds checking.

2. Mutating raw *value* elements may have much the same effects as above: data over/under runs occurring as a result of memory allocation based on incorrect values.

3. Mutating derived *length* elements could lead to over or under runs. For example, feeding problematic integer values into the *Length* values could cause memory allocation arithmetic routines to corrupt due to integer overflow or signedness issues.

All of these types of potential errors indicate the need for data sanitization checks such as buffer bounds checking, and also the manner in which fuzzing affects parsers more than any other element of an application. However, in order to reach the parser, the data must 'survive' demarshalling processing to some degree, hence the need for a high degree of test data compliance with the effective input space.

10.4.9 Meta Data and Memory Allocation Vulnerabilities

A good general approach to software exploitation is to identify and test the assumptions of designers and developers [20, p. 48]. Incidentally, this is why defining and documenting implementation and security assumptions is a recommended activity for designers and developers alike [21, Chapter 9].

Length values within *Type Length Value* encoding schemes are a prime target for 'assumption testing'. The apparent complexity of analysis and intelligent modification of transfer syntaxes such as *Type Length Value* could lead many designers and developers to errantly trust such data.

It would be highly dangerous to perform memory allocation based on derived Meta data elements: this is akin to trusting a client to validate data before sending it to a server. However, developers might consider it unlikely that someone would go to the trouble of malforming a raw element and recalculating a derived Meta data element,

just as many have believed in the past that it is unlikely that a user might develop a malicious client [20, 48].

For an experienced analyst, neither analysis nor modification of such protocols is complex. Furthermore, by focussing testing on a limited region of application input data, (in this case the relationship between *Length* Meta data and the raw data it is supposed to be derived from), significantly reduces the testers workload. This is an example of the power of intelligent fuzzing: it permits the tester to capitalize upon an understanding of the underlying protocols and focus the test effort on applying intelligent values on tightly defined regions.

Below is an excerpt from a vulnerability alert discovered and published by iDefence, (which has already been quoted in Chapter 2, *Software Vulnerabilities*) which provides a real world example of an instance where a parser errantly trusted values from input for memory allocation.

> *"When parsing the TIFF directory entries for certain tags, the parser uses untrusted values from the file to calculate the amount of memory to allocate. By providing specially crafted values, an integer overflow occurs in this calculation. This results in the allocation of a buffer of insufficient size, which in turn leads to a heap overflow."* [3]

Hence, intelligent fuzzing can be used to focus on a specific area of a protocol implementation that the tester believes may be vulnerable, or it may be used to enumerate an entire protocol specification, based on a complete definition of the specification syntax and the tokenisation of each message.

10.4.10 Realising Fuzzer Tokenisation Via Block-Based Analysis

The SPIKE fuzzing framework, publicly released in 2002 by Dave Aitel, facilitates tokenization via a block-based approach [4], where elements are assembled from *blocks* into a 'SPIKE', as shown in figure 10.5.

In block-based analysis raw elements are specified using blocks. A range of blocks cater for common data types such as bytes, words, strings, and so on. In the example

[3] `http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=593`

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

Figure 10.5: A basic SPIKE, taken from [4].

in figure 10.5, the size and data format (its a binary, big-endian word (16 bits)) is defined in the first line, and the third line defines the type (s_binary) and the value held in it (in the octal values of each 4-bit nibble: 0x01, 0x02, 0x03, 0x04).

In terms of *Type, Length, Value* we have covered *Value* and *Type*. In SPIKE, *Derived* elements are supported by *block listeners* to derive the required block size value dynamically. In this way, the fuzzer could replace the default value of '0x01, 0x02, 0x03, 0x04' with, for example, '0xFF' the length could be recalculated.

A more detailed example of block-based analysis is provided in Chapter 9, *Case Study 2*, Section 9.4.3 *Analyse the Protocol*, using the Sulley Fuzzing framework, the authors of which acknowledge SPIKE's block based analysis approach as being superior to any other.

10.5 Chapter Summary

We have examined the nature of machine communication and the need for protocols to provide contextual information allowing meaning to be derived from data to produce information. We have seen that sending control and raw data along a single channel can lead to security risks, for data recipients unless data sanitization is performed before control data is processed.

We have identified two key requirements for intelligent fuzzing: *stateful message sequencing* and *tokenization*. We have seen how stateful message sequencing can be defined by a formal grammar used to 'walk' a protocol implementation through its component states. We have seen how tokenisation can be used to fuzz data elements in isolation, and based on their identified type.

CHAPTER

ELEVEN

CASE STUDY 3 – PROTOCOL FUZZING A VULNERABLE WEB SERVER

This case study was devised to test whether the Sulley fuzzing framework could be used to develop a simple HTTP fuzzer that, given a (data element-level) definition of a small subset of the protocol (and the configuration of a suitable test environment), would reveal defects in a web server application with known vulnerabilities. The author chose to use a known vulnerable target as the objective of this exercise was not to discover vulnerabilities, but to determine the effectiveness of 'intelligent' fuzzing.

The vulnerable web server that was used is described by it's developers as follows:

> *"Beyond Security's Simple Web Server (SWS) is a web server application created for internal testing of the beSTORM fuzzer, while working on the HTTP 1.0 and HTTP 1.1 protocol modules. The server was built with a large set of common security holes which allows testing of fuzzing tools functionality and scenario coverage."* [40]

Figure 11.1 shows the Graphical User Interface (GUI) of the Simple Web Sever. A total of 14 (known) vulnerabilities are present in the server. These can be 'switched' on or off, using tick-boxes in the GUI. The server also generates a log file of input passed to it, and the most recent request/s can be viewed in real-time via the GUI.

The author reverse-engineered Simple Web Server allowing him to define which boxes were ticked (and hence which vulnerabilities were present) by default. This was done because the method used to ensure target recovery after a crash was to restart the target application, and Simple Web Server launches with only eight of fourteen

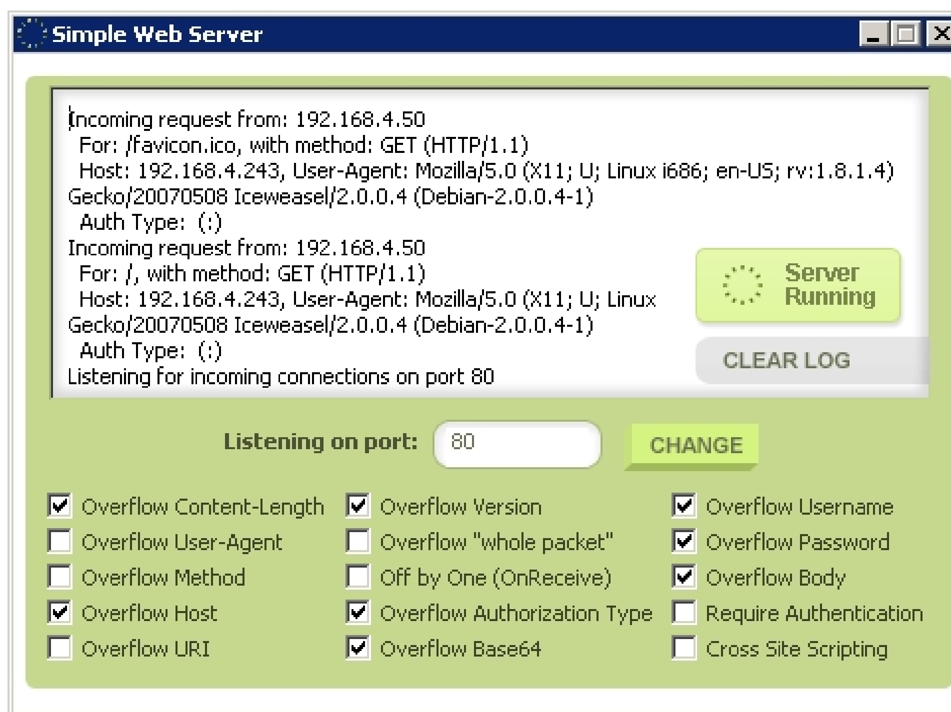


Figure 11.1: The Beyond Security Simple Web Server.

possible defects enabled. Note that the author could have used *vmcontrol.py*, (an agent provided with Sulley for controlling VMWare) to restore the virtual machine to a snapshot with Simple Web Server configured with all fourteen defects enabled. However, the restore process was found to take something in the region of three to four minutes compared to approximately ten seconds required for a restart.

In figure 11.2, the Simple Web Server binary file is shown when opened within a hex editor. By arranging the width of the hex editor to a value of six, we have aligned the data to 6 byte intervals. The data selected, commencing at 0x1A16 and extending to 0x1A6F, contains 15 separate 6-byte lines, each of which correspond to one of the 15 tick-boxes that set which defects are enabled. Observing the third column, note that all of the lines have been set to the value 0x86, bar one (0x1A5E), which has been set to 0x9E. By setting these values to either 0x86 or 0x9E, we can cause tick boxes to be set or un-set, respectively, at launch. Figure 11.2 shows the modified version of Simple Web Server, where all defects have been set to be enabled at launch. The line where column three is set to 0x9E (0x1A5E) corresponds to the

'Require Authentication' setting tick box.

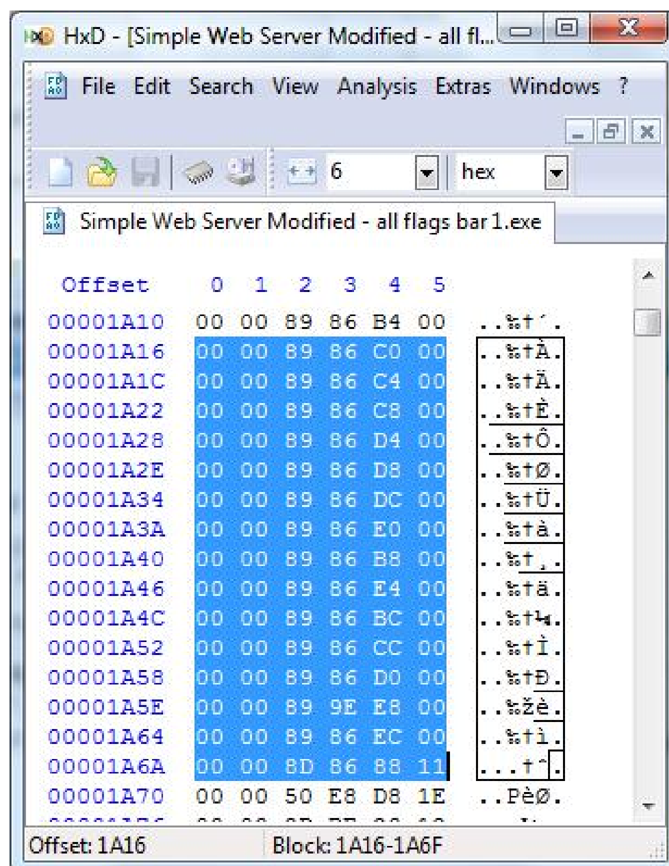


Figure 11.2: Altering Simple Web Server to define which tick boxes are set by default.

The author set all of the tick-boxes save for the 'Require Authentication' box since this adds an authentication layer, requiring that the user logs into the server. As we shall see later, the fuzzer was only provided with sufficient information to fuzz the method, the identifier and protocol version, which correspond to the Method, Universal Resource Indicator (URI) and the Version overflow vulnerabilities in Simple Web Server, respectively.

In order to complete this case study, a Personal Computer running Windows Vista Operating System software and the following applications were used:

1. VMware Server software[1]
2. Virtual machine running Windows XP Service Pack 2 Operating System software
3. Sulley fuzzing framework[2]
4. Simple Web Server[3]
5. Wireshark network protocol analyser[4]

1, 2. The primary reason for using a virtual machine as a test platform was convenience. Once a virtual machine has been configured, a snapshot can be taken which captures the system state. The system can then be restored to the state captured by the snapshot with a single mouse click.

3. The Sulley fuzzing framework was chosen for its advanced features, particularly the support it provides to block-based protocol analysis.

4. Simple Web Server was developed by Beyond Security as a purposefully vulnerable web server specifically for fuzzer testing.

5. Wireshark is a network protocol analyser which was used to independently monitor the output from the fuzzer before, during and after testing. It is important to verify that fuzzer output takes the format that is expected, since a failure on the part of the fuzzer or the tester can invalidate testing, which can result in misplaced conclusions.

11.1 Methodology

The component tasks were:

1. Establish and configure the test environment

[1] http://www.vmware.com/products/server/
[2] http://www.fuzzing.org/2007/08/13/new-framework-release/
[3] http://blogs.securiteam.com/index.php/archives/995
[4] http://www.wireshark.org/

2. Analyse the target (determine the process name, and the commands required to start and stop it)
3. Analyse the protocol (create the HTTP BASIC Sulley request)
4. Configure the fuzzer session (create the http Sulley session)
5. Configure the oracle (create netmon and procmon batch files)
6. Launch the session
7. Process the results

11.1.1 Establish and Configure the Test Environment

VMware was installed on the host operating system. Within VMware, a virtual machine running Windows XP Service Pack 2 operating system was installed. This was to be the test platform upon which the target application would be installed.

For clarity, the host operating system will hereafter be referred to as the *host*, the virtual machine test operating system will be referred to as the *guest*, and the target application will be referred to as the *target*.

The Sulley fuzzing framework was installed onto the host and guest systems. A virtual network was established between the host and guest systems. The folder on the host holding the Sulley fuzzing framework was mapped to the test system. Sulley would be executed on the host, and the oracles *procmon* and *netmon* would be executed via shortcuts to batch files located in the mapped folder. Batch files were used for convenience because the arguments to netmon and procmon can be lengthy.

The Simple Web Server web server application was installed on the test system. A browser was launched on the host and pointed at the web server on the guest to confirm the server was running and was accessible from the host. The web server's default page was displayed.

11.1.2 Analyse the Target

Analysing the target inputs was simple since the scope of testing was limited to remote access inputs and it is trivial to determine that the server was listening to port 80.

When fuzzing an application, it is possible, likely even, that the application will crash. In the event that the target does crash, Sulley is able to stop and start a target process in order to resume testing without intervention. The procmon oracle monitors the target process and issues the required commands. This requires that (a) the target application process name is known and (b) commands to stop and start the target are identified and tested by running them at the command line.

Task manager was used to confirm the process name was the name of the executable file: *SimpleWebServerA.exe*. From the command line, the *Start* command followed by the full path and file name of the executable was found to start the target. The taskkill command with the *IM* switch was found to cleanly close the application. Although the executable did shut down cleanly from the command line without the IM switch, it did not during testing. When the application crashed, an error dialogue box was launched and this seemed to stall the shutdown/restart process managed by procmon, hence the use of the IM switch.

Figure 11.3 shows an excerpt from the http session script. This excerpt shows how the information gathered about the target was passed to procmon.

```
target.procmon_options = \
{
"proc_name" : "SimpleWebServerA.exe",
"start_commands" : ['Start C:\B\S\SimpleWebServerA.exe'],
"stop_commands" : ['taskkill /IM SimpleWebServerA.exe'],
```

Figure 11.3: An excerpt from the http session script.

11.1.3 Analyse the Protocol

Consider a basic HTTP request for a resource that may be issued from a client to a server, such as:

GET /index.html HTTP/1.1

According to RFC 2616, *"A request message from a client to a server includes, within the first line of that message, the method to be applied to the resource, the identifier of the resource, and the protocol version in use."* [12]

In our simple example, 'GET' is the method to be applied, '/index.html' is the identifier and 'HTTP/1.1' is the protocol. In order to define this protocol for the fuzzer, it must be broken further down into data elements. Working from left to right:

GET /index.html HTTP/1.1

1. 'GET' is declared as a string
2. A white space character is declared as a delimiter
3. '/' is a delimiter
4. 'index.html' is a string
5. Another white space delimiter
6. 'HTTP' is a string
7. '/' is a delimiter
8. '1' is a string
9. '.' is a delimiter
10. '1' is a string
11. '\r\n\r\n' is declared as type 'static', which instructs the fuzzer not to modify this element. This is done as this element is required to satisfy the protocol as defined in RFC2616, that is, if requests are not followed by '\r\n\r\n' then they will not be processed by the server.

Once a message has been separated into data elements (a process termed *tokenisation*, since *token* is another term used to describe an individual data element), and the type of each element is defined, the Sulley fuzzer framework is able to:

1. treat each element separately, such that individual elements can be fuzzed in isolation while the validity of the rest of the elements is maintained

2. individually fuzz elements using one of the following approaches as specified by the tester: intelligently selected heuristics[5], brute force all possible values, randomly generated data, or maintain the specified value.

In Sulley, messages are described using blocks (Sulley's creators credit Dave Aitel's block-based analysis approach to protocol dissection and definition). A block can be assembled from multiple defined elements and could be used to describe an HTTP GET request. Blocks can also be used to define a group of similar requests as is shown in figure 11.4. This example, taken from [46], will programmatically generate and fuzz GET, HEAD, POST and TRACE requests.

In figure 11.4 we see the HTTP request, which contains a single block called HTTP_BASIC, which in turn defines HTTP GET, HEAD, POST and TRACE messages. Note that the HTTP request has been separated into data elements, and each element has been declared as a specific data type (in this case s_delim, s_static or s_string for delimiter, static, or string element types, respectively).

In Sulley, one or many *messages* are defined in *blocks*, of which one or many are grouped into a *request*, and one or many requests are imported into a *session*: the term used to describe a test run in Sulley.

11.1.4 Configure the Fuzzer Session

In Sulley a test run of multiple test instances is termed a session. Sulley is designed such that once a session has been created and configured the entire test run can be completed without user intervention.

Figure 11.6 shows a Sulley session titled 'http', which was created for this case study. The second line from the top causes the http request to be imported, allowing the later use of the HTTP_BASIC block. The third line defines a path to a file

[5]The heuristics that are applied vary depending on the specified data type: strings are fuzzed with a library of strings heuristics, delimiters are fuzzed with delimiter heuristics, and so on.

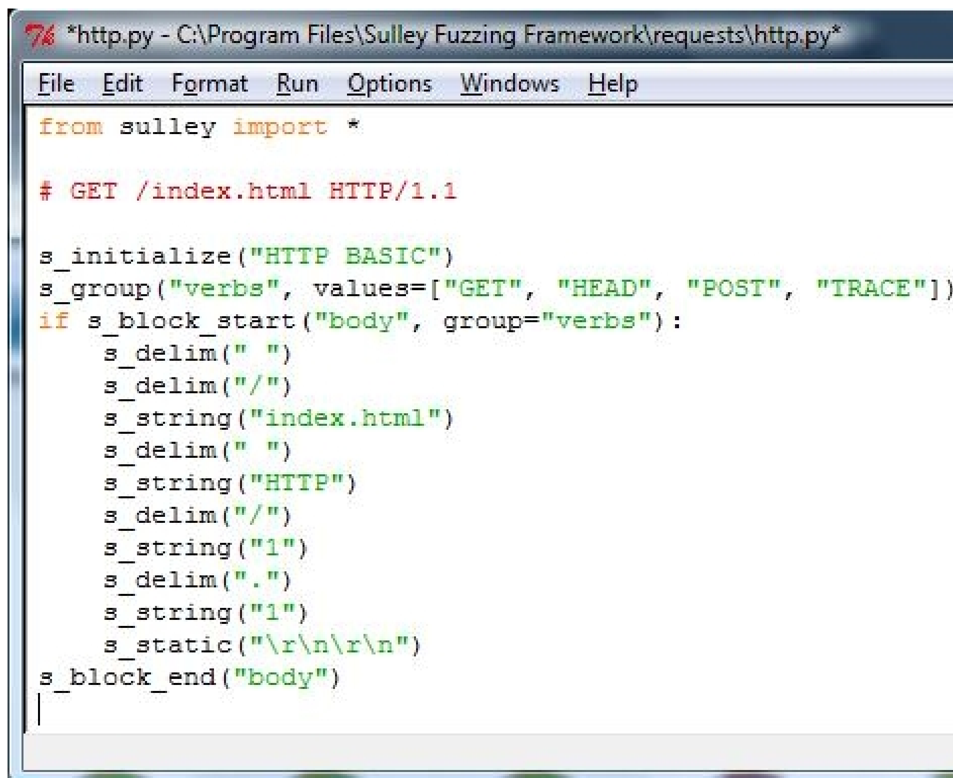


Figure 11.4: The completed http request, containing a single block called HTTP_BASIC, taken from [46]

(in green) where the session data can be held. By creating a file to hold session information, a session can be paused and resumed at any time. This even allows the test system to be de-powered (as long as the guest OS configuration is preserved upon resumption - facilitated by restoring to a snapshot), which is useful for very long test runs.

The next three lines define the IP address and port numbers of the target application, and also the netmon and procmon oracles. Target IP address and port information allows Sulley to determine what support for networking is required. In this case, Sulley will be fuzzing HTTP, residing at the application layer. Sulley will automatically generate the required lower layers such as establishing a Transport Control Protocol (TCP) ‘three way handshake’ at the beginning of each test instance. For each modification of each of the HTTP elements, Sulley will invisibly generate valid

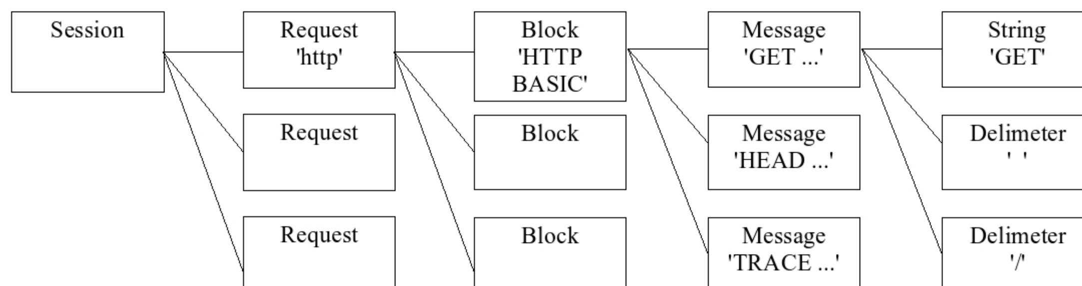


Figure 11.5: The hierarchical relationship between sessions, requests, blocks, messages, and data elements in the Sulley fuzzing framework.

lower layer encapsulating protocols such as TCP segments, IP packets and Ethernet frames.

Since the oracles run on the OS that is hosting the target (a virtual guest OS), not the OS that is hosting Sulley, the fuzzer will need to communicate with the oracles over a (in our case virtual) network. This is achieved by a proprietary protocol termed 'pedrpc' developed by one of the Sulley architects.

The next six lines are configuration information for the procmon oracle: these have already been discussed. The final three lines actually trigger the fuzzing session. The first and third lines are always present as shown. The middle line is where the HTTP_BASIC block is defined. One could call multiple blocks in the manner shown, and Sulley would call and fuzz these as required.

11.1.5 Configure the Oracle

We have already mentioned the two oracles provided with Sulley: procmon and netmon. Procmon is used to monitor the target process, including:

1. detecting whether the target process is running
2. launching the target process if it's not running

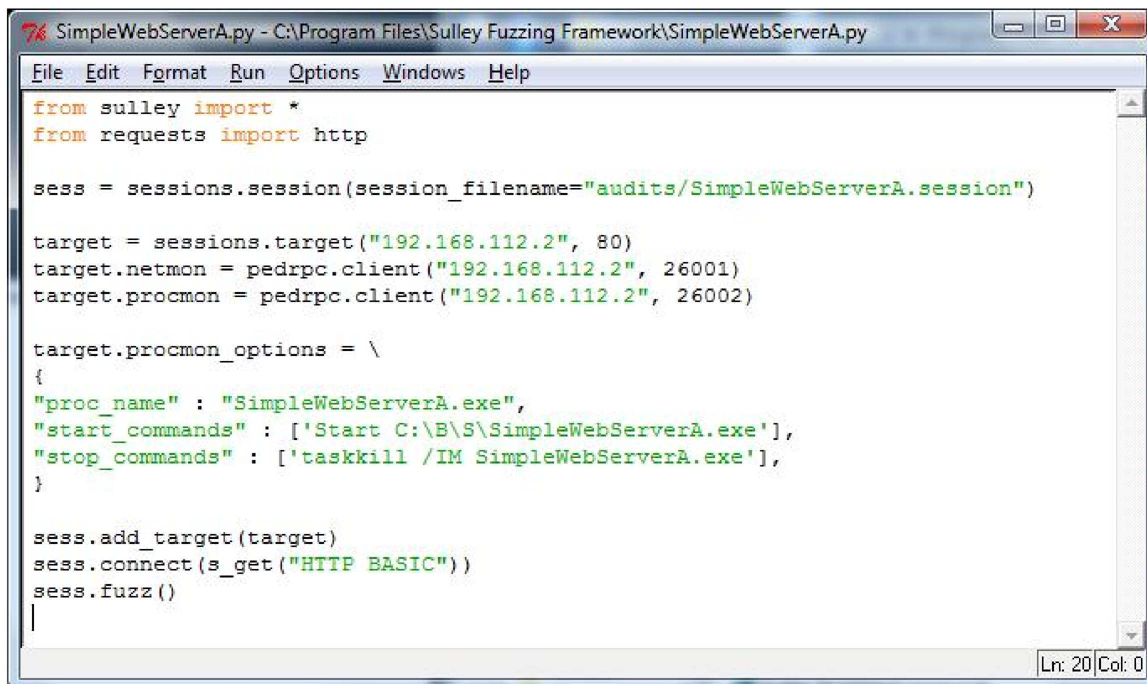


Figure 11.6: The completed http session script, showing how a Sulley test run (termed a *session*) may be configured.

3. cleanly shutting the target process down if it fails

4. capturing detailed crash reports when the application fails

Netmon is used to monitor network traffic between Sulley and the target, which means capturing Sulley test instances in the form of packet capture (termed *pcap*) files. These files capture the bi-directional network traffic exchange between Sulley and the target, and can be viewed a network protocol analyser such as Wireshark.

In figure 11.7 Wireshark is used to view a packet capture of test case instance number 137. Note that Sulley has replaced the resource element of the HTTP message *index.html* with a long string of repeated octets that spell out DEADBEEF.

Since Sulley generates test data iteratively as required, pcap files are important as they act as a 'recording' of each test case instance. A pcap file which captures an instance where Sulley output causes the target to fail is one half of the output

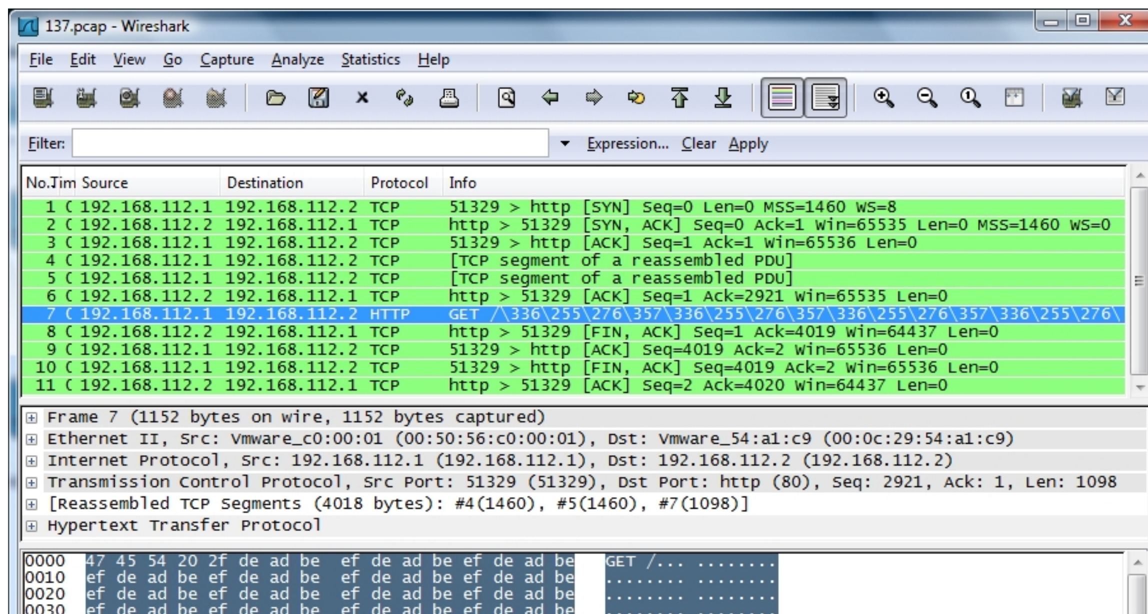


Figure 11.7: Packet capture of test case instance number 137.

of a Sulley test run. The other half is the accompanying detailed crash report as generated by procmon.

Configuring the procmon oracle had been partially completed via the entries placed in the http session script shown in figure 11.6. This was completed via the creation of a batch file to run procmon with the command line options shown in figure 11.9.

The path and file name required to create a crashbin file are defined using the -c switch. Crashbin files store the detailed crash reports which procmon generates when the target fails a report is generated such as that seen in figure 11.8.

The log level is set to 9999 using the -l switch because the author wanted verbose logging for troubleshooting purposes. The process name is supplied using the -p switch. Configuring the netmon oracle meant setting the Network Interface Card

(using the -d switch) to ‘1’. The BPF filter string was set (using the -f switch) to “src or dst port 80”, since the traffic to sniff was going to be web traffic aimed at port 80. The batch file configuration arguments can be seen in figure 11.10

```
SimpleWebServerA.exe:00403524 rep movsd from thread 3928 caused access violation
when attempting to write to 0xbeade3e5

CONTEXT DUMP
  EIP: 00403524 rep movsd
  EAX: 00544547 (   5522759) -> N/A
  EBX: 00000000 (         0) -> N/A
  ECX: 00000100 (       256) -> N/A
  EDX: ffffd43a (4294956090) -> N/A
  EDI: beade3e5 (3199067109) -> N/A
  ESI: 00129ef8 (   1220344) -> HTTP/1.1 (stack)
  EBP: 0012e7a8 (   1238952) -> @J6,xxuwff@@K@@$7GET/ (stack)
  ESP: 00129ae8 (   1219304) -> GET/ (stack)
  +00: 0012bac7 (   1227463) -> GET/ (stack)
  +04: 0012bacb (   1227467) -> GET/ (stack)
  +08: 00544547 (   5522759) -> N/A
  +0c: 00000000 (         0) -> N/A
  +10: beadde2f (3199065647) -> N/A
  +14: beaddeef (3199065839) -> N/A

disasm around:
        0x004034ed mov [esi+0xf0],eax
        0x004034f3 mov [esi+0xf4],cx
        0x004034fa lea edi,[esi+0xf6]
        0x00403500 mov ecx,0x100
        0x00403505 lea esi,[esp+0x10]
        0x00403509 rep movsd
        0x0040350b mov edi,[esp+0x814]
        0x00403512 add edi,0x4f6
        0x00403518 mov ecx,0x100
        0x0040351d lea esi,[esp+0x410]
        0x00403524 rep movsd
        0x00403526 pop esi
        0x00403527 mov al,0x1
        0x00403529 pop edi
        0x0040352a add esp,0x808
        0x00403530 retn 0x8
        0x00403533 int3
        0x00403534 int3
        0x00403535 int3
        0x00403536 int3
        0x00403537 int3

SEH unwind:
        0012e514 -> SimpleWebServerA.exe:00408ef8 mov eax,0x40d604
        0012e59c -> MFC80.DLL:7829bf68 mov edx,[esp+0x8]
        0012e604 -> MFC80.DLL:7829bc0b mov edx,[esp+0x8]
        0012e694 -> MFC80.DLL:782998a7 mov edx,[esp+0x8]
        0012e6f4 -> USER32.dll:77d70494 push ebp
        0012e784 -> USER32.dll:77d70494 push ebp
        0012ff14 -> MFC80.DLL:7829f054 mov edx,[esp+0x8]
        0012ffb0 -> SimpleWebServerA.exe:004092fd mov eax,0x40dae4
        0012ffe0 -> SimpleWebServerA.exe:004086df push dword [esp+0x10]
        ffffffff -> kernel32.dll:7c8399f3 push ebp
```

Figure 11.8: A detailed crash report generated by the procmon oracle for test case 137.

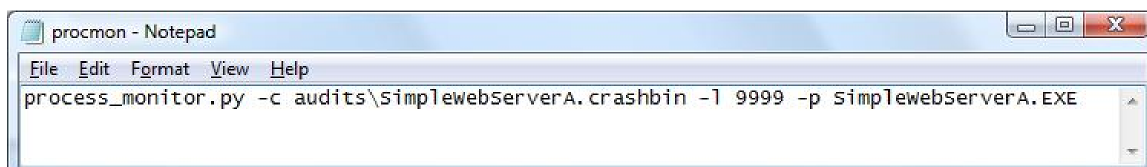


Figure 11.9: The batch file created for the procmon oracle.

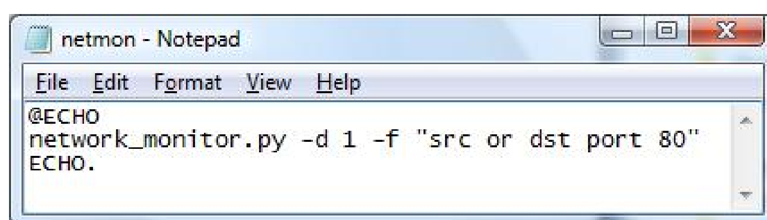


Figure 11.10: The batch file created for the netmon oracle.

11.1.6 Launch the Session

Launching the session simply consisted of switching to the guest operating system, launching the Simple Web Server application on the target, running the procmon and netmon batch file shortcuts, and finally switching back to the host operating system and launching the session by double clicking on the http session file.

Once the session was launched, progress could be monitored by pointing a browser to IP address 127.0.0.1:26000 on the host operating system, this is the localhost address with port number 26000 selected, which is where Sulley's web server is located.

11.2 Results

Based on the tokenised http request we provided for Sulley, it automatically generated 18,092 tests. Sulley completed the test run without any human intervention taking approximately 3 hours to complete the tests. It should have taken approximately 5-10 hours as each test instance requires about 1-2 seconds to complete. The difference appeared to be due to the fact that Sulley did not run every test case. This might be because Sulley skips test cases when a long series of test cases result

in a target failure. Restarting the target application consumes approximately 10 to 15 seconds, and it is not unusual for a single software defect (termed a *noisy bug*) to be triggered by hundreds or thousands of sequential test cases. In this case, it would make sense to skip test cases when such behaviour is observed. However, the author can find no documentation to support this theory (some aspects of Sulley are apparently undocumented and it is up to the user to discover and reverse engineer certain nuances).

The results are presented in the form of a web page which is shown in figure 11.11.

The format of the crash synopsis logs are as follows. Each line represents a crash - i.e. a terminal exception has been raised. Monitoring is not limited to the target application, but extends to any linked libraries. From left to right:

- the test case number is the sequential number of the test case that triggered the crash report,
- the crash synopsis comprises the process (or library) name that raised an exception, the memory address of the last executed current instruction at the time of the crash,
- the specific nature of the exception,
- the number of the process thread that raised the exception,
- the general class of the exception.
- the size in bytes of the pcap file.

Sulley was able to identify 23 individual test cases that triggered a failure of the target application, and one that caused the ntdll.dll dynamic link library to fail.

11.3 Analysis of One of the Defects

Twenty one of the discovered instances appear to be as a result of a single software defect in the Simple Web Sever application, located at the process memory address location 0x00403524. These defects were triggered when a `rep movsd` instruction was processed. This is a string operation instruction that causes a single DWORD (a double word: a 32 bit data unit on X86 processors) to be copied from one string into another.

String operations, like almost all assembler operations, generally require that specific values be passed to specific registers. Hence, `rep movsd` only tells us about the general instruction: in order to understand the specifics, we will have to determine the value of certain registers.

When performing string operations, the ESI register (the Extended Source Index) is used to hold the value of the source offset: this is the first memory address to copy *from*. The EDI register (the Extended Destination Index) is used to hold the value of the destination offset: this is the first memory address to copy *to*.

Referring to figure 11.7, the pcap file from test case 137, we can see that Sulley has fed a long string of octets that have the value DEADBEEF, making them very recognisable when reading stack traces from crash reports.

If we then refer to figure 11.8, the crash report from test case 137, we can see that an access violation occurred when the target attempted to write to memory address 0xbeade3e5. The context dump shows the state of the registers at the time of the crash. Note that EDI holds the value 0xbeade3e5. We may surmise that the cause of the crash was the presence of the value 0xbeade3e5 in the EDI register. But how did this value get into EDI? The first four octets 'bead' look very similar to DEADBEEF. Referring to the last two values in the context dump, ESP +10 and ESP + 14 are 0xbeadde2f and 0xbeaddeef. Working from left to right, if one takes the first and the fifth octets and swaps them around, the result will be 0xdeadbeef. From this, we may surmise that the inserted string of DEADBEEF values has overflowed a buffer, overwriting the most significant word of the EDI register.

11.4 Conclusions

The hypothesis was proven in that the simple HTTP protocol fuzzer created using Sulley was able to trigger and detect software defects in a known vulnerable application based on a tokenised sample of the HTTP protocol.

Sulley Fuzz Control **RUNNING**

Total: 18,091 of 18,092 [==] 99.994%
HTTP BASIC: 18,091 of 18,092 [==] 99.994%

Pause Resume

Test Case #	Crash Synopsis	Captured Bytes
000137	SimpleWebServerA.exe:00403524 rep movsd from thread 3928 caused access violation	4,654
000138	ntdll.dll:7c90eaf4 push ebx from thread 3400 caused access violation	42,268
001168	SimpleWebServerA.exe:00400047 int 0x21 from thread 2496 caused access violation	
001242	SimpleWebServerA.exe:00403524 rep movsd from thread 3368 caused access violation	2,606
001245	SimpleWebServerA.exe:00403524 rep movsd from thread 1684 caused access violation	2,606
002289	SimpleWebServerA.exe:00403524 rep movsd from thread 3596 caused access violation	
002360	SimpleWebServerA.exe:00403524 rep movsd from thread 3080 caused access violation	2,609
002363	SimpleWebServerA.exe:00403524 rep movsd from thread 3072 caused access violation	2,609
003407	SimpleWebServerA.exe:00403524 rep movsd from thread 2332 caused access violation	
003478	SimpleWebServerA.exe:00403524 rep movsd from thread 1196 caused access violation	2,609
003481	SimpleWebServerA.exe:00403524 rep movsd from thread 2424 caused access violation	2,609
004525	SimpleWebServerA.exe:00403524 rep movsd from thread 976 caused access violation	
005690	SimpleWebServerA.exe:00403524 rep movsd from thread 852 caused access violation	
006811	SimpleWebServerA.exe:00403524 rep movsd from thread 3800 caused access violation	
007929	SimpleWebServerA.exe:00403524 rep movsd from thread 988 caused access violation	
009047	SimpleWebServerA.exe:00403524 rep movsd from thread 2792 caused access violation	
010212	SimpleWebServerA.exe:00403524 rep movsd from thread 3248 caused access violation	
011333	SimpleWebServerA.exe:00403524 rep movsd from thread 3800 caused access violation	
012451	SimpleWebServerA.exe:00403524 rep movsd from thread 3196 caused access violation	
013569	SimpleWebServerA.exe:00403524 rep movsd from thread 2176 caused access violation	
014734	SimpleWebServerA.exe:00403524 rep movsd from thread 2788 caused access violation	
015855	SimpleWebServerA.exe:00403524 rep movsd from thread 1220 caused access violation	
016973	SimpleWebServerA.exe:00403524 rep movsd from thread 2536 caused access violation	
018091	SimpleWebServerA.exe:00403524 rep movsd from thread 3884 caused access violation	

Figure 11.11: The results webpage generated by Sulley. Colours have been inverted.

CHAPTER

TWELVE

CONCLUSIONS

12.1 Key Findings

We have seen that fuzzing is a unique method for vulnerability discovery in that it requires minimal effort, comparatively low technical ability, no access to source code, minimal human intervention and minimal financial investment. It also produces output that is *"verifiable and provable at runtime"* [48].

The low barriers to entry, in comparison to other security test methods, mean that fuzzing *should* be applied within development teams internally as a matter of course [11, Slide 21], and that it *could* be employed by a wider demographic such as end users, corporations, and Small and Medium Enterprises to detect the presence of implementation defects in software products. If such a widespread adoption of fuzzing were to occur, it might drive the software development industry to produce software products with fewer vulnerabilities.

However, we have also seen that it is generally infeasible to exhaustively enumerate the application input space, and that as a result we must either try to enumerate the *effective* input space rather than the *absolute* input space, (which, if we do this 'blindly' may mean we will not fully enumerate the effective input space, and if we do this 'intelligently' we will have to devote effort to define a grammar to describe the rules obeyed by input data) or be prepared to accept a high level of test inefficiency, meaning many useless test cases are executed.

Furthermore, it is impossible to accurately measure the effectiveness of fuzzing,

since the only practical metric, code coverage, only measures one 'dimension' of fuzzing: the amount of (reachable) code executed; it does not measure the range of input values fed to the target at each code region. We have also seen that target monitoring is often less than ideal, resulting in wasted effort and false negatives as errors are triggered by fuzzing, but are not detected [48].

The disadvantages of fuzzing mean that it may arguably offer a lower degree of assurance than 'white box', source code-centric approaches such as code auditing and dynamic structural analysis [48].

HD Moore, the man behind the Metasploit website and the *month of browser bugs*[1], where fuzzing was used to discover a large number of bugs has described fuzzing as:

> *"[...] the process of predicting what types of programming errors may exist in the product and the inputs that will trigger those errors. For this reason, fuzzing is much more of an art than a science."* [46, p. xix]

This statement acknowledges the inability of zero knowledge fuzzing methods to provide absolute input space enumeration, and hence measurably high levels of code coverage. A better approach would be to limit the scope of testing by tuning the test data to suit the application by predicting the error types and the input required to trigger them. In so doing, we may accept the limitations of fuzzing and move away from theoretical perfection (the 'science' of fuzzing) toward a pragmatic approach (the 'art' of fuzzing) by employing an awareness of each of the relevant software and hardware layers to identify intelligent heuristics or 'educated guesses'.

Drawing on the benefits of fuzzing, we could employ grammars and automation to chart the effective input space for us, or combine fuzzing with white box techniques to apply brute force testing to small regions of code.

It might argued be that fuzzing is ideal for security researchers: it does not require access to source code, it is not complex or demanding, it is largely automatable and is ideal for uncovering vulnerabilities in code that has not been properly security tested. These features combined mean that large numbers of complex applications can be (albeit shallowly) reviewed for 'low-' and 'medium-hanging fruit' in terms of software vulnerabilities. Fuzzing has two key advantages over all other security

[1] `http://blog.metasploit.com/2006/07/month-of-browser-bugs.html`

testing approaches: the source code is not required and the volume, the scalability, the transferability across applications is unrivalled [48], [27].

It might also be argued that fuzzing is ideal for attackers for all of the reasons above. I would argue that fuzzing could be a force for good as long as this undoubtedly dangerous and powerful tool is used by the right people (namely, internal development teams, security researchers, vendors and end-users) to identify and fix software defects so that they cannot be used for malicious purposes.

12.2 Outlook

Fuzzing has moved from being a home-grown, almost underground activity during the 1980s to falling under the spotlight of academia (particularly, but not limited to, the PROTOS test suite development at the University of Oulu, Finland) and hacking conventions such as Black Hat, Defcon and the Chaos Communication Congress during the mid-1990s, to moving into the commercial world during the last few years. Sutton et. al list six different commercial offerings [46, p. 510], including *Codenomicon*[2], a suite of commercial protocol testing tools based on PROTOS [46, p. 510], and the *Mu Security Mu-4000*[3], a stand alone hardware appliance aimed at testing network devices [46, p. 512].

A number of fuzzer development frameworks have now been freely released to the public. Some examples are: *SPIKE*[4], *Peach*[5], *Antiparser*[6], *Autodafe*[7], *Sulley*[8], *GPF*[9], *DFUZ*[10]. Anyone who wishes to explore fuzzing can do so using some of the most advanced toolsets for free.

Sutton et al. suggest that the limitations of individual security testing approaches mean that hybrid approaches (colloquially termed *grey box testing*) will see continued

[2] `http://www.codenomicon.com`
[3] `http://www.musecurity.com/products/mu-4000.html`
[4] `http://www.immunitysec.com/resources-freesoftware.shtml`
[5] `http://peachfuzzer.com/`
[6] `http://antiparser.sourceforge.net/`
[7] `http://autodafe.sourceforge.net/`
[8] `http://www.fuzzing.org/2007/08/13/new-framework-release/`
[9] `http://www.vdalabs.com/tools/efs_gpf.html`
[10] `http://www.genexx.org/dfuz/`

development. An example is the use of fuzzing to test the output of static source code analysis thus gaining high code coverage (a problem for fuzzing) and low false positives (a problem for static analysis) [46, p. 515].

In a paper entitled *Automated Whitebox Fuzz Testing* [17], a hybrid approach to fuzzing is presented that employs x86 instruction-level tracing and emulation to analyse an application at run-time, trace control paths, and map the manner in which input influences control path selection. This mapping between input and control path flow is used to create input that will exercise different control paths. This can be used to achieve very high code coverage.

Automatic protocol dissection offers the fuzz tester an automated means to perform the tokenization and message sequencing aspects of intelligent fuzzing. This area is relatively immature and is the subject of research by a number of individuals [46, p. 419]. If achievable, this technology would be particularly threatening to those who employ 'security through obscurity' by employing closed, proprietary protocols in the hope that the effort required to analyse them may dissuade testers and attackers alike. This is also an area that, perhaps while not strictly required by determined, experienced analysts, is particularly important for commercial products, where such functionality would be a strong attractor for customers seeking a 'turnkey' solution.[11]

Sutton et al. provide two examples of research advances in areas adjacent to security testing are feeding into automatic protocol dissection, namely *bioinformatics* [46, p. 427] and *genetic algorithms* [46, p. 431]. The former is concerned with making sense of naturally occurring sequences of data such as gene sequences, and theorems from this field have been applied for network protocol analysis.

The PI (Protocol Informatics) framework[12] written by Marshall Beddoe employs a number of bioinformatic algorithms in order to automatically deduce field boundaries within unknown protocols [46, p. 428].

Genetic algorithms (GA) can be applied to solving computational problems through *fitness* and *reproduction* functions, mimicking natural selection. An example of the use of GA in fuzzing is *Sidewinder*, a fuzzer that employs GA to craft input that will cause targeted vulnerable code (such as a vulnerable function) to be executed [42].

[11]'Turnkey' solutions require no input from the user to perform their function, beyond simply pressing a button or turning a key.

[12]`http://packetstormsecurity.org/sniffers/PI.tgz`

The problem is defined in the form of a requirement to traverse a call graph function from an input node (e.g. `recv()`) to the target node (e.g. `strcpy()`) [43, Slide 11]. This appears to be a form of automated *red pointing*, as described in Chapter 3, Section 3.3.2 *Dynamic Structural Testing*.

The area of fuzzer target monitoring is certainly ripe for development. Sutton et al. describe Dynamic Binary Instrumentation (DBI) as being the *"panacea of error detection"* [46, p. 492]. DBI appears to offer the potential to detect errors before they cause the application to fail, speeding up the identification of the root causes of failures. For example, DBI can be used to perform bounds checking at the granularity of individual memory allocation instructions. This means that an overrun can be detected and trapped at the moment the first byte is overwritten, rather than when the application subsequently fails due to a read or write operation triggering an access violation [46, p. 492].

I foresee the development of fuzzing spreading in two directions: leading edge research will drive deeper application inspection through areas such as enhanced code coverage via directed execution and the application of advanced algorithms. At the same time, commercial fuzzer product vendors will drive development toward increasingly automated, transparent 'turnkey' solutions.

12.3 Progress Against Stated Objectives

The influence of fuzzing on the information security community has been covered in chapter 1. I would argue that fuzzing has not directly influenced the development community at this time. Most information security professionals have at least heard of fuzzing; most developers have not. Fuzzing has assisted attackers and security researchers alike to identify security vulnerabilities in software products, and has likely increased the number and frequency of vulnerability reports. This, in turn, may have increased some software vendors' awareness of the need for security testing throughout the development lifecycle, which, in turn may impact upon software developers. Until software security testing becomes as commonplace and accepted as, say unit testing, developers, vendors and customers will continue to be exposed to software vulnerabilities.

We have briefly covered the fact that there is little overlap between Common Criteria-based software evaluations and fuzzing. We have placed fuzzing within the

range of software security testing methods, and compared it against these alternative approaches.

We have presented a general model of a fuzzer, and explored some of the different approaches to fuzzing, examining 'dumb' and 'intelligent' fuzzers, and exploring network-level fuzzing and file-level fuzzing. We have charted the evolution of fuzzing and examined some of the problems that have had to be solved in order for the field of fuzzing to advance.

In addition to presenting the theory behind fuzzing, we have documented practical vulnerability discovery with two very different fuzzers We have provided a practical example of how the output of fuzz testing could be used to develop 'proof-of-concept' code to assess and demonstrate the nature of a vulnerability, or to produce malicious software designed to exploit a discovered vulnerability.

Although we have explored the use of two different types of fuzzer, we have not compared two or more fuzzers in a side-by-side comparison. Charlie Miller has conducted a like-for-like comparison of 'dumb' mutation-based and 'intelligent' generation-based fuzzing, and concluded that intelligent fuzzing found more defects in less time, even though it took longer to initiate testing [35]. Jacob West has conducted a similar study comparing fuzzing with source code analysis [48].

We have not been able to properly examine what metrics may be used to compare fuzzers. This is mainly because fuzzing cannot currently be measured. We have identified and discussed code coverage and code path tracing as being the only currently available fuzzer tracking metric, and we have highlighted its failings as a metric for assessing the completeness of testing.

We have covered the evolution of fuzzing from its almost accidental 'discovery' in the late '80s through to the current period of transition from underground development and academic research to commercial product.

Table 12.1 provides a summary of progress against objectives defined at the outset of the project.

Objective	**Comment**
Comment on the influence of fuzzing on the information security and software development communities.	We have covered the positive influence of testing in Chapter 1.
Compare fuzzing with other forms of software security assurance - i.e. Common Criteria evaluations.	We have compared fuzzing with Common Criteria evaluation in Chapter 1, and with other security testing approaches in Chapter 3.
Briefly explain where fuzzers fit within the field of application security testing: i.e. who might use them, why they are used, and what value they offer the Information Security industry, software developers, end-users, and attackers.	We have placed fuzzing within the context of various security testing approaches in Chapter 3.
Describe the nature, types and associated methodologies of the various different classes of fuzzers. Identify some of the limitations of, and problems with, fuzzing.	We have identified problems and limitations with random, brute force and blind data mutation in Chapters 5, 6 and 8, we have discussed the limitations of exception monitoring in Chapters 7, 8 and 11, and we have covered problems with protocol fuzz testing in Chapters 10 and 11.
Examine the use of fuzzing tools for discovering vulnerabilities in applications.	We have examined the practical discovery of vulnerabilities in Chapters 8 and 11.
Examine how the output of a fuzzing tool might be used to develop software security exploits (case study);	We have examined how fuzzer output might be employed for exploit development in Chapter 9.
Compare some of the available fuzzing tools and approaches available possibly using two or more types of fuzzer against a single target application with known vulnerabilities.	We have not completed this objective, though we have been able to explore 'blind' data mutation fuzzing and 'intelligent' protocol fuzzing in Case studies 1 and 3.
Examine what metrics may be used to compare fuzzers.	We have not completed this objective, though we have explored the primary metric: code coverage in a number of chapters.
Examine the evolution of fuzzing tools, comment on the state of the art and the outlook for fuzzing.	We have examined the evolution of fuzzing tools from the earliest examples (starting in Chapter 4) through to the state of the art (Chapters 10 and 11), and we have explored the outlook in this chapter.

Table 12.1: Summary of progress against objectives.

APPENDIX

A

APPENDIX 1 – A DESCRIPTION OF A FAULT IN THE FILEFUZZ APPLICATION

A.1 Description of Bug

There were two areas where the bug manifested:

1. An error that went something along the lines of:

   > *"Count 1: The output char buffer is too small to contain the decoded characters, encoding 'Unicode (UTF-8)' fallback 'System.Text.DecoderReplacementFallback"'*

   This error is described at a number of websites. [1], [2]

2. Test files generated were all zeros after the point where the exception occurred, usually less than 10 bytes into reading the target file.

A.2 Cause of the Bug

This was apparently due to a problem with a component of FileFuzz called Read.cs which used a problematic function called *peekchar*. Peekchar appeared to be throwing an exception on non-legal UTF8 values while checking for the End Of File and

[1] http://forums.microsoft.com/MSDN/ShowPost.aspx?PostID=127647&SiteID=1

[2] http://www.msdner.com/dev-archive/193/12-44-1939995.shtm

this stalled the read-in of the target file. One theory is that the .Net framework was updated post development of the application in such a way as to interfere with peekchar's operation.

A.3 Addressing the Bug

FileFuzz is open source, in the sense that the developer (Michael Sutton) provides all of the source code and project files required to modify the application. This meant that the author was able to fix the bug by editing Read.cs, replacing the code shown in figure A.1 with that of figure A.2 and then, of course, rebuilding the application using Microsoft Visual Studio. The following region of Read.cs appears to be the source of the problem:

```
try
{
        while (brSourceFile.PeekChar() != -1)
        {
        sourceArray[sourceCount] = brSourceFile.ReadByte();
        sourceCount++;
        }
        brSourceFile.Close();
}
```

Figure A.1: The problematic region of Read.cs

Note that all that is changed is the line where peekchar is called.

```
try
{
        while (brSourceFile.BaseStream.Position < brSourceFile.BaseStream.Length)
        {
        sourceArray[sourceCount] = brSourceFile.ReadByte();
        sourceCount++;
        }
        brSourceFile.Close();
}
```

Figure A.2: The modified region of Read.cs

APPENDIX

B

APPENDIX 2 – THE SULLEY FUZZING FRAMEWORK LIBRARY OF FUZZ STRINGS

As Sutton, et al. put it: *"A heuristic is nothing more than an a fancy way of saying 'educated guessing.'"* [46, p. 81]. Yet, heuristics are a pragmatic solution to the problem that exhaustive testing is not possible. If we can't submit every possible value to a vulnerable function, then let us select a handful of particularly challenging values and submit those instead. However, a requirement for an educated guess is some contextual information, and for our purposes this will consist of an understanding of the reasons why some forms of input are particularly problematic.

What follows is an analysis of the library of heuristics employed to fuzz string elements provided with the Sulley fuzzing framework. Excerpts have been taken from the source code for the sulley.primitives module, specifically the string (base_primitive) class. This library has been largely based upon the extensive library included in the SPIKE Fuzzer Creation Kit Version 2.9 [3], and an examination of the SPIKE source code, particularly comments within it, has been used to shed light on some of the lines in the Sulley source code.

Sulley contains additional heuristics libraries dedicated to fuzzing other data element types (or primitives as they are termed in Sulley) such as bytes, dwords, delimiters, and so on.

```
399                # omission and repetition.
400                "",
401                self.value * 2,
402                self.value * 10,
403                self.value * 100,
```

Figure B.1: String omission and repetition [6].

B.1 Omission and Repetition

Figure B.1 is an excerpt from the string (base_primitive) class of the sulley.primitives module [6], as are all of the figures in this Appendix. In figure B.1, line 400 omits the original string. Lines 401, 402 and 403 repeat the original string twice, 10 and 100 times. This is aimed at buffer overflows and also possibly string expansion (see below) attacks.

B.2 String Repetition with \xfe Terminator

```
405                # UTF-8
406                self.value * 2   + "\xfe",
407                self.value * 10  + "\xfe",
408                self.value * 100 + "\xfe",
```

Figure B.2: The \xfe terminator is appended to repeated strings [6].

In figure B.2, strings are repeated as in lines 400 to 403 in figure B.1, but are terminated with a \xfe. This is puzzling, since the comment suggests UTF-8 encoding, yet the value 0xfe is never used in UTF-8. However, the author did find an internet discussion posting [1] that indicated that 0xfe is the string terminator for Type Length Value in some implementations.

[1] http://discussion.forum.nokia.com/forum/showthread.php?p=408889

B.3 A Selection of Strings Taken from SPIKE

```
410                # strings ripped from spike (and some others I added)
411                "/.:/"  + "A"*5000 + "\x00\x00",
412                "/.../" + "A"*5000 + "\x00\x00",
413                "/.../.../.../.../.../.../.../.../.../.../",
414                "/../../../../../../../../../../../../etc/passwd",
415                "/../../../../../../../../../../../../boot.ini",
416                "..:..:..:..:..:..:..:..:..:..:..:..:..:",
417                "\\\\*",
418                "\\\\?\\",
419                "/\\" * 5000,
420                "/." * 5000,
421                "!@#$%%^#$%#$@#$%$$@#$%^^**(()",
422                "%01%02%03%04%0a%0d%0aADSF",
423                "%01%02%03@%04%0a%0d%0aADSF",
424                "/%00/",
425                "%00/",
426                "%00",
427                "%u0000",
```

Figure B.3: A selection of different strings taken from SPIKE [6].

In figure B.3, lines 411 and 412 contain delimiter characters followed by long strings of A's (intended to overflow a buffer), followed in turn by two null bytes (in order to terminate the string correctly). The origin of the "/.:/" string pattern appears to be the Spike.c source code. This particular string is commented: *"weird dce-rpc for the locator service, from MS pages ..."* [3, Line 2,113]. The "/.../" string is also from the same region of the spike.c source code, but is not commented. Lines 413 and 416 appear to be repeated versions of the string patterns mentioned above.

Line 414 is intended to cause directory traversal fault where the target application is denied access to a file. The path requested is the password file in the /etc folder of the host: a resource that should not normally be accessible. If the application cannot gracefully handle being denied access to a file, it might fail in a vulnerable manner. In this case, the file requested would suggest a Unix based host.

Line 415 is similar to line 414 but is aimed at a Windows-based host. Note that in both cases, the oracle would not detect that the file in question had been accessed. Only if a failure occurred, likely due to an access violation exception being raised, would the oracle be made aware.

Lines 417 and 418 are uncommented in the Spike source code.

Lines 419 and 420 generate very long (5000 character) strings of delimiters possibly aimed at triggering string expansion faults.

Lines 421, 422 and 423 may be related to shell escape attacks (also known as command injection), based on comments in the Spike.c source code, where these line originate from.

Lines 424 to 427 are all forms of null terminators: special characters that are used to indicate the end of a character string. The standard null termination character is "%00", but there are many variations. Injecting null terminators could result in an unexpectedly short or empty string, which might trigger a buffer underflow. Where a received input value usually has a string appended to it, an injected null character can prevent the appended value from being attached, which might aid a directory traversal attack where a server tries to make input values 'safe' by appending strings to them [37, p. 217]. An encoded newline character ("%0a") can have a similar 'truncation' effect on dynamically assembled strings.

B.4 Format Specifiers

```
429            # format strings.
430            "%n"     * 100,
431            "%n"     * 500,
432            "\"%n\"" * 500,
433            "%s"     * 100,
434            "%s"     * 500,
435            "\"%s\"" * 500,
```

Figure B.4: Strings aimed at triggering format string vulnerabilities [6].

In figure B.4 we see a number of different methods of delivering %n and %s format specifiers. These have been discussed in detail in Chapter 2, *Software Vulnerabilities*.

B.5 Command Injection

```
437                 # command injection.
438                 "|touch /tmp/SULLEY",
439                 ";touch /tmp/SULLEY;",
440                 "|notepad",
441                 ";notepad;",
442                 "\nnotepad\n",
```

Figure B.5: Strings aimed at triggering command injection/shell escape defects [6].

As per the inline comment, the code in figure B.5 attempts to inject commands using the '|' (Unix pipeline) and ';' command delimiter characters, and also the '\n' new line control characters. As before both Windows and UNIX based platforms are catered for. Here, *notepad.exe* is the windows application that is intended to execute, while *touch* is the UNIX application. As before, the oracle would not detect if any of the commands were successfully detected, only if injection led to a crash.

A possible addition to the above might be a similar line with encoded new line characters ("%0a").

B.6 SQL Injection

```
444                 # SQL injection.
445                 "1;SELECT%20*",
446                 "'sqlattempt1",
447                 "(sqlattempt2)",
448                 "OR%201=1",
```

Figure B.6: Strings intended to trigger SQL injection vulnerabilities [6].

Lines 445 through 448 in figure B.6 are intended to trigger SQL errors which might indicate whether the target is susceptible to SQL injection attacks. All of these are taken from Spike.c.

B.7 Binary Value Strings

```
450            # some binary strings.
451            "\xde\xad\xbe\xef",
452            "\xde\xad\xbe\xef" * 10,
453            "\xde\xad\xbe\xef" * 100,
454            "\xde\xad\xbe\xef" * 1000,
455            "\xde\xad\xbe\xef" * 10000,
456            "\x00"             * 1000,
```

Figure B.7: Tainted binary value strings [6].

In figure B.7, the longer strings resulting from lines 451 to 455 are obviously aimed at inducing buffer overflows. The letters used spell out the words DEAD BEEF, which is an unusual, eye catching combination. This makes it an excellent data tainting value that is a valid binary value and is easy to spot if it propagates into output, onto the stack or a register, or into an error message.

Line 456 is a long string of nulls. The author's theory is that this might be treated as a string of 1000 nulls by one aspect of an application, and as a string of zero length by another, resulting in a buffer overflow.

In Chapter 9, *Case Study 2*, an instance is described where one of the above 'deadbeef' strings causes a vulnerable application to crash and aids the analysis phase.

B.8 Miscellaneous Strings

```
458            # miscellaneous.
459            "\r\n" * 100,
460            "<>" * 500,
```

Figure B.8: Strings intended for command truncation and string expansion [6].

Line 459 in figure B.8 is intended to cause command truncation attacks like those already seen employing null characters.

Line 460 is aimed at inducing string expansion. Angle brackets are common in HTML and it is not unusual for them to be expanded as they are decoded [46]. String expansion bugs are more likely to induce a fault when there a large number of the characters that are subject to expansion are passed to the application. This is because total string expansion is a function of the number of expanding characters.

B.9 A Number of Long Strings Composed of Delimiter Characters

In lines 464 to 493 in figure B.9, a number of special characters (and two-character combinations) are defined. Each of these will be used to generate a sequence of long strings.

The hexadecimal values 0xFE and 0xFF expand to four characters under UTF16 [46, p. 85]. By replacing a string with a large number of these unusual characters[2], the aim is to trigger stack or heap overruns as a string is expanded in an unforeseen manner. String expansion faults may only be triggered when large numbers of special characters are passed to an application. Since such faults are only likely to be triggered by actively malicious input, they are less likely to be trapped by standard (i.e. non security-related) testing methodologies.

B.10 Long Strings with Mid-Point Inserted Nulls

The code shown in figure B.10 might be aimed at triggering buffer over or underflows where buffer length arithmetic might be upset by the mid-point null bytes.

```
463        # add some long strings.
464        self.add_long_strings("A")
465        self.add_long_strings("B")
466        self.add_long_strings("1")
467        self.add_long_strings("2")
468        self.add_long_strings("3")
469        self.add_long_strings("<")
470        self.add_long_strings(">")
471        self.add_long_strings("'")
472        self.add_long_strings("\"")
473        self.add_long_strings("/")
474        self.add_long_strings("\\")
475        self.add_long_strings("?")
476        self.add_long_strings("=")
477        self.add_long_strings("a=")
478        self.add_long_strings("&")
479        self.add_long_strings(".")
480        self.add_long_strings(",")
481        self.add_long_strings("(")
482        self.add_long_strings(")")
483        self.add_long_strings("]")
484        self.add_long_strings("[")
485        self.add_long_strings("%")
486        self.add_long_strings("*")
487        self.add_long_strings("-")
488        self.add_long_strings("+")
489        self.add_long_strings("{")
490        self.add_long_strings("}")
491        self.add_long_strings("\x14")
492        self.add_long_strings("\xFE")   # expands to 4 characters under utf16
493        self.add_long_strings("\xFF")   # expands to 4 characters under utf16
```

Figure B.9: Strings composed of delimiter characters [6].

B.11 String Length Definition Routine

In figure B.11, lines 537 to 549 cause a sequence of long strings to be generated which are comprised of the special characters defined in lines 464 to 493. Note the 'fence post' values: 128, 256, 1024, 2048 and so on, and also note the '+1' and '-1' values: e.g. 255 and 257.

A special class of buffer overflow termed an 'off-by-one' error occurs when bounds

[2]In fact, there are many such 'unusual' characters, in that delimiter characters may be treated differently, (i.e. expand when decoded or translated) to alpha-numeric characters.

```
495        # add some long strings with null bytes thrown in the middle of it.
496        for length in [128, 256, 1024, 2048, 4096, 32767, 0xFFFF]:
497            s = "B" * length
498            s = s[:len(s)/2] + "\x00" + s[len(s)/2:]
499            self.fuzz_library.append(s)
```

Figure B.10: Strings of ‘B’s with inserted null bytes [6].

```
537        Given a sequence, generate a number of selectively chosen strings lengths of the given sequence and add to the
538        string heuristic library.
539
540        @type  sequence: String
541        @param sequence: Sequence to repeat for creation of fuzz strings.
542        '''
543
544        for length in [128, 255, 256, 257, 511, 512, 513, 1023, 1024, 2048, 2049, 4095, 4096, 4097, 5000, 10000, 20000,
545                       32762, 32763, 32764, 32765, 32766, 32767, 32768, 32769, 0xFFFF-2, 0xFFFF-1, 0xFFFF, 0xFFFF+1,
546                       0xFFFF+2, 99999, 100000, 500000, 1000000]:
547
548            long_string = sequence * length
549            self.fuzz_library.append(long_string)
```

Figure B.11: Long strings of varying problematic lengths [6].

checking is applied to prevent buffer overflows, but the bounds checking fails to account for the fact that arrays start at zero, not one, or that strings have null terminators. The result may be an exploitable buffer overflow vulnerability where only a single byte of an adjacent logical object may be overwritten.

B.12 User Expansion Routine

Here, in figure B.12, a simple means to extend the string fuzzing library is provided: the tester simply creates a file ‘.fuzz_strings’ inside which user generated fuzz strings can be placed.

```
501         # if the optional file '.fuzz_strings' is found, parse each line as a new entry for the fuzz
502         try:
503             fh = open(".fuzz_strings", "r")
504 
505             for fuzz_string in fh.readlines():
506                 fuzz_string = fuzz_string.rstrip("\r\n")
507 
508                 if fuzz_string != "":
509                     self.fuzz_library.append(fuzz_string)
510 
511             fh.close()
512         except:
513             pass
```

Figure B.12: User expansion of the Sulley string fuzz library [6].

APPENDIX

C

APPENDIX 3 – COMMUNICATION WITH MICROSOFT SECURITY RESPONSE CENTRE

—–Original Message—–
From: Clarke TP
Sent: Sunday, March 02, 2008 3:02 AM
To: Microsoft Security Response Centre
Subject: Possible Bug Report

Hi,

Please find below a bug report relating to rundll32.exe and access.cpl. I believe that this not a major bug, but I thought I'd report it since it seems like the correct thing to do.

I am a believer in responsible disclosure and have not disclosed the details of this bug to anyone, but I do plan to discuss this matter with my project supervisor - I am writing a thesis around fuzzing and I would like to use this as case study material. This is why I have submitted supporting information in the form of two (draft) case studies - see Bug Report.pdf.

It is my hope that you will find this to be a non-bug or of little relevance. Please advise me as to whether the details of this bug can be shared or discussed?

If you require any further information, I will be happy to assist.

Regards,

Toby Clarke

Type of issue (buffer overflow, SQL injection, cross-site scripting, etc.)

By overwriting four bytes commencing at location (decimal) 1068 of access.cpl, the value of the four overwritten bytes will be passed to EIP, meaning that an attacker can redirect program flow. It is also possible to overwrite regions of access.cpl with shellcode, and this shellcode is mapped into memory when access.cpl is launched by rundll32.exe.

Combining the two factors above we have a local only arbitrary code execution without privilege escalation.

Product and version that contains the bug

Windows XP Professional SP 2, fully patched as of February 08 More specifically access.cpl and rundll32.exe

Service packs, security updates, or other updates for the product you have installed

- test machine was fully patched via windows update as of February 08

Any special configuration required to reproduce the issue

- None

Step-by-step instructions to reproduce the issue on a fresh install

please see Bug Report.pdf. This contains two case studies. Case study 1 details how the error was found using FileFuzz, and case study 2 (starting on page 11) covers how it was exploited.

Proof-of-concept or exploit code

See two attached files:

access-test6.cpl causes the motherboard beeper to sound on unpatched XP Professional SP2 machines. The shellcode was taken from the milw0rm website.

access-test-exp-6.cpl binds a shell to a listening port (8721) on a fully patched (as of feb 08) XP Professional SP 2 install. The shellcode was taken from the Metasploit website. The shellcode runs under the process name of the compromised host process - rundll32.exe. If you kill the process, the listening port will close. I think that the shellcode will survive a reboot. Just to re-iterate: its not my shellcode, I'm just using it to prove the injection vector is functional.

Impact of the issue, including how an attacker could exploit the issue

This is a local only attack, and rundll32.exe runs with the same privileges as the user that launches it, so this may not have any impact at all. However, I am not qualified to properly determine the potential impact, nor have I fully explored this area. One concern is that access.cpl writes to the registry - this could lead to the possibility of a file that could be sent by email to a victim which when clicked on could alter registry settings. I have not explored this.

—–Original Message—–
From: Microsoft Security Response Centre [mailto:securemicrosoft.com]
Sent: Sun 02/03/2008 23:43
To: Clarke TP
Cc: Microsoft Security Response Centre
Subject: RE: Possible Bug Report [MSRC 8041dm]

Thanks very much for your report. I have opened case 8041 and the case manager, Dave, will be in touch when there is more information. In the meantime, we ask you continue to respect responsible disclosure guidelines and not report this publicly until users have an opportunity to protect themselves. You can review our bulletin acknowledgement policy at

http://www.microsoft.com/technet/security/bulletin/policy.mspx

and our general policies and practices at

http://www.microsoft.com/technet/security/bulletin/info/msrpracs.mspx

If at any time you have questions or more information, please respond to this message.

Warren

—–Original Message—–
From: Clarke TP
Sent: Monday, March 03, 2008 6:05 AM
To: Microsoft Security Response Centre
Subject: RE: Possible Bug Report [MSRC 8041dm]

Hi

I note that my email service has blocked me from accessing my proof of concept files from my Sent Items folder:

Access to the following potentially unsafe attachments has been blocked: access-test6.cpl, access-test-exp-6.cpl

If you haven't received these files and wish to access them, let me know and I'll rename the extensions to .txt and try resending them.

Regards,

Toby

—–Original Message—–
From: Microsoft Security Response Centre [mailto:securemicrosoft.com]
Sent: Wed 3/5/2008 7:52 PM
To: Clarke TP
Cc: Microsoft Security Response Centre
Subject: RE: Possible Bug Report [MSRC 8041dm]

Hi Toby,

Thank you for submitting this report to us. We have concluded our investigation and determined that an attack can only be leveraged locally and in the context of the logged on user.

Issue Summary

A potential issue was reported in Windows XP. By overwriting four bytes commencing at location (decimal) 1068 of access.cpl, the value of the four overwritten bytes will be passed to EIP. This infers an attacker can redirect program flow. It is also possible to overwrite regions of access.cpl with shell code, and this shell code is mapped into memory when access.cpl is launched by rundll32.exe. However, it was determine that this could not lead a user to be granted access they did not already posses.

Root Cause:

Users can execute code at their current privilege level.
The file used (access.cpl) cannot be accessed without elevated rights.
A good explanation of this issue may also be found at:

http://blogs.msdn.com/oldnewthing/archive/2006/05/08/592350.aspx

At this time we are closing this case. But if you discover any additional vectors

to amplify this attack please report back to us and I can easily reopen this case.

Thanks,
Dave
MSRC

BIBLIOGRAPHY

[1] *Skillsoft 241292_eng - c sharp 2005: Serialization and i/o*, Online Course Content.

[2] *What is a parser?*, Online-Article, Site visited on 2nd August 2008, Not dated., `http://www.programmar.com/parser.htm`.

[3] Dave Aitel, *Spike.c*, C source code, Part of the SPIKE Fuzzer Creation Kit Version 2.9, `http://www.immunitysec.com/resources-freesoftware.shtml`.

[4] ______, *The advantages of block-based protocol analysis for security testing*, `http://www.net-security.org/dl/articles/advantages_of_block_based_analysis.pdf` (2002).

[5] George A. Akerlof, *The market for 'lemons': Quality uncertainty and the market mechanism*, Quarterly Journal of Economics 84 (3): 488500., 1970, `www.econ.ox.ac.uk/members/christopher.bowdler/akerlof.pdf`.

[6] Pedram Amini and Aaron Portnoy, *Primitives.py, part of the sulley fuzzing framework source code*, Python source code, August 2007, `http://www.fuzzing.org/2007/08/13/new-framework-release/`.

[7] Danilo Bruschi, Emilia Rosti, and R. Banfi, *A tool for pro-active defense against the buffer overrun attack*, ESORICS '98: Proceedings of the 5th European Symposium on Research in Computer Security (London, UK), Springer-Verlag, 1998, pp. 17–31.

[8] Jared DeMott, *The evolving art of fuzzing*, 2006, `www.vdalabs.com/tools/The_Evolving_Art_of_Fuzzing.pdf`, `video.google.com/videoplay?docid=4641077524713609335`.

[9] Mark Dowd, John McDonald, and Justin Schuh, *The art of software security assessment: Identifying and preventing software vulnerabilities*, Addison-Wesley Professional, 2006.

[10] Erwin Erkinger, *Software reliability in the aerospace industry - how safe and reliable software can be achieved*, 23rd Chaos Communication Congress presentation, 2006, `http://events.ccc.de/congress/2006/Fahrplan/events/1627.en.html`.

[11] Gadi Evron, *Fuzzing in the corporate world*, Presentation, 23rd Chaos Communication Congress: Who can you trust?, December 2006, `http://events.ccc.de/congress/2006/Fahrplan/attachments/1248-FuzzingtheCorporateWorld.pdf`.

[12] R. Fielding, J. Mogul, H. Frytsyk, L. Masinter, P. Leach, and T. Berners-Lee, *Rfc 2616 - hypertext transfer protocol-http/1.1*, Technical Report, June 1999, IETF.

[13] James C. Foster and Vincent Liu, *Writing security tools and exploits*, Syngress Publishing, 2005.

[14] Andreas Fuschberger, Software Security course lecture, 2008, March.

[15] Lars Marius Garshol, *Bnf and ebnf: What are they and how do they work?*, Online-Article, `http://www.garshol.priv.no/download/text/bnf.html`.

[16] Patrice Godefroid, *Random testing for security: blackbox vs. whitebox fuzzing*, RT '07: Proceedings of the 2nd international workshop on Random testing (New York, NY, USA), ACM, 2007, pp. 1–1.

[17] Patrice Godefroid, Michael Y. Levin, and David Molnar, *Automated whitebox fuzz testing*, Technical Report MS-TR-2007-58, May 2007, `www.isoc.org/isoc/conferences/ndss/08/papers/10_automated_whitebox_fuzz.pdf`.

[18] Dick Grune and Ceriel J. H. Jacobs, *Parsing techniques: a practical guide*, Ellis Horwood, Upper Saddle River, NJ, USA, `http://www.cs.vu.nl/~dick/PTAPG.html`, 1990.

[19] Les Hatton, *Five variations on the theme: Software failure: avoiding the avoidable and living with the rest*, University of Kent, Canterbury, November 2003.

[20] Greg Hoglund and Gary McGraw, *Exploiting software: How to break code*, Pearson Higher Education, 2004.

[21] Michael Howard and Steve Lipner, *The security development lifecycle*, Microsoft Press, Redmond, WA, USA, 2006.

[22] Senior Consultant Information Risk Management Plc, John Yeo, Personal conversation, July 2008.

[23] Intel, *Pentium processor family developer's manual*, `http://www.intel.com/design/pentium/MANUALS/24143004.pdf`.

[24] Rauli Kaksonen, *A functional method for assessing protocol implementation security*, VTT Publications, `http://www.ee.oulu.fi/research/ouspg/protos/analysis/VTT2001-functional/`, 2001.

[25] Rauli Kaksonen, M. Laakso, and A. Takanen, *System security assessment through specification mutations and fault injection*, Proceedings of the IFIP TC6/TC11

International Conference on Communications and Multimedia Security Issues of the New Century (Deventer, The Netherlands, The Netherlands), Kluwer, B.V., 2001, p. 27.

[26] Marko Laakso, *Protos - mini-simulation method for robustness testing*, Presentation, 2002, Oulu University Secure Programming Group.

[27] Scott Lambert, *Fuzz testing at microsoft and the triage process*, Online-Article, Site visited on 2nd August 2008, September 2007, `http://blogs.msdn.com/sdl/archive/2007/09/20/fuzz-testing-at-microsoft-and-the-triage-process.aspx`.

[28] John Leyden, *Penetrate and patch e-business security is grim*, Online-Article, Published Wednesday 20th February 2002 09:42 GMT, Site visited July 30th, 2008, `http://www.theregister.co.uk/2002/02/20/penetrate_and_patch_ebusiness_security/`.

[29] Tim Lindholm and Frank Yellin, *Java virtual machine specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[30] B. Littlewood (ed.), *Software reliability: achievement and assessment*, Blackwell Scientific Publications, Ltd., Oxford, UK, UK, 1987.

[31] Thomas Maller, Rex Black, Sigrid Eldh, Dorothy Graham, Klaus Olsen, Maaret Pyhjrvi, Geoff Thompson, Erik van Veendendal, and Debra Friedenberg, *Certified tester — foundation level syllabus*, Online-Article, April 2007, `http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf`.

[32] Gary McGraw, *Testing for security during development: Why we should scrap penetrate-and-patch*, Online-Article, Site visited July 30th 2007, `http://www.cigital.com/papers/download/compass-2.pdf`.

[33] Microsoft, *Info: Windows rundll and rundll32 interface*, Online-Article, November 2006, http://support.microsoft.com/kb/164787.

[34] Barton P. Miller, Louis Fredriksen, and Bryan So, *An empirical study of the reliability of unix utilities*, Commun. ACM **33** (1990), no. 12, 32–44.

[35] Charlie Miller, *How smart is intelligent fuzzing - or - how stupid is dumb fuzzing?*, Conference Presentation, Defcon 15, September 2007, http://video.google.co.uk/googleplayer.swf?docid=-6109656047520640962&hl=en&fs=true.

[36] Christiane Rutten, *Fuzzy ways of finding flaws*, Online-Article, Site visited on 1st August 2008, January 2008, http://www.heise-online.co.uk/security/Fuzzy-ways-of-finding-flaws--/features/100674.

[37] Joel Scambray, Mike Shema, and Caleb Sima, *Hacking exposed web applications, second edition (hacking exposed)*, McGraw-Hill Osborne Media, 2006.

[38] Bruce Schneier, *How security companies sucker us with lemons*, Online-Article, April 2007, http://www.wired.com/politics/security/commentary/securitymatters/2007/04/securitymatters_0419.

[39] scut / team teso, *Exploiting format string vulnerabilities*, September 2001, crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf.

[40] Beyond Security, *Simple web server (sws) test case*, Online-Article, http://www.beyondsecurity.com/sws_overview.html.

[41] S.E. Smith, *What is garbage in garbage out?*, Online-Article, Site visited July 30th, 2007, http://www.wisegeek.com/what-is-garbage-in-garbage-out.htm.

[42] Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou, *Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting*, www.acsac.org/2007/papers/22.pdf.

[43] ———, *Sidewinder: An evolutionary guidance system for malicious input crafting*, Presentation at Black Hat Conference, August 2006, www.blackhat.com/presentations/bh-usa-06/BH-US-06-Embleton.pdf.

[44] Brad Stone, *Moscow company scrutinizes computer code for flaws*, Online-Article, January 2007, http://www.iht.com/articles/2007/01/29/business/bugs.php.

[45] Michael Sutton, *Fuzzing - brute force vulnerability discovery*, Presentation, RECON conference, Montreal, Canada, Friday June 16th 2006.

[46] Michael Sutton, Adam Greene, and Pedram Amini, *Fuzzing: Brute force vulnerability discovery*, Addison-Wesley Professional, 2007.

[47] David Thiel, *Exposing vulnerabilities in media software*, Black Hat conference presentation, BlackHat EU 2008, www.isecpartners.com/files/iSEC_Thiel_Exposing_Vulnerabilities_Media_Software_bh07.pdf.

[48] Jacob West, *How i learned to stop fuzzing and find more bugs*, Defcon conference presentation, available as a video recording, DefCon 15 Las Vegas, 2007, August 2007, video.google.com/videoplay?docid=-5461817814037320478.

[49] Peter Winter-Smith and Chris Anley, *An overview of vulnerability research and exploitation*, 2006, www.cl.cam.ac.uk/research/security/seminars/archive/slides/2006-05-16.ppt.