CODENOMICON

white paper

TOPIC Codenomicon robustness testing – handling the infinite input space while breaking software

Lämsä Jarkko, Kaksonen Rauli, Kortti Heikki
Codenomicon Ltd
Tutkijantie 4E
90570 Oulu, Finland
email: info@codenomicon.com

© Codenomicon 2006. All rights reserved.

TOPIC Codenomicon robustness testing
– handling the infinite input space while breaking software

How does Codenomicon select its test cases from the infinite number of possible inputs that can be fed to a particular interface? How can we achieve a satisfactory input coverage – a good representation of the input space?

1 Introduction

Traditionally, black-box tests are derived from specifications to show that the target software meets its requirements. The purpose of this kind of positive testing is to show that the target software works whereas the purpose of negative testing that it does not [Beizer]. In this paper, we discuss a method of negative black-box robustness testing, Codenomicon testing, and how to achieve a satisfactory input space coverage.

Codenomicon robustness tests are negative tests derived from protocol specifications in black-box manner. Tests are aimed to proactively find robustness and security related design and implementation level bugs in protocol implementations [Codenomicon]. These kind of bugs usually manifest as degradation or denial of service and can in some cases be exploited to compromise the information security of the target system [Kaksonen00].

The testing method is heavily influenced by syntax testing [Beizer] and fault injection [Voas] through software interfaces. Software interfaces are a potential attack vector against the security and robustness of the system [Kaksonen01]. Tests are packaged into test tools. Each tool contains a set of test cases for an individual protocol.

The tests are designed specifically to reveal the undesired, harmful or dangerous functionality in the targe(image)t software. The test method is to systematically break every possible element and structure defined (implicitly or explicitly) in the protocol specifications and then feed the resulting test cases to the target implementation. However, they are only a subset of the infinite number of all possible negative tests. Any protocol specification allows an infinite number of negative inputs. The key is to find a best estimation representation of the input space that is likely to cause the tested software to fail while maintaining a reasonable test execution time.

1.1 Undesired functionality as a source of robustness problems

In a piece of software, the implemented functionality almost always differs from the desired functionality specified by the positive requirements in the software specification. Conformance bugs are introduced when the desired feature set is not attained. Extra functionality inside safety margins results in “creative features” that are not specified in the requirements. [Eronen].

As illustrated in figure 1, undesired functionality is the subset of the implemented functionality that implements negative requirements. Negative requirements define what the software should not do. They may be explicitly specified or implicit. For example, “the software should not crash” is a negative requirement and if it is implemented, the software will have undesired functionality. In other words, area C in figure 1 is where all the vulnerabilities lie. [Eronen].

© Codenomicon 2006. All rights reserved.

TOPIC Codenomicon robustness testing
– handling the infinite input space while breaking software

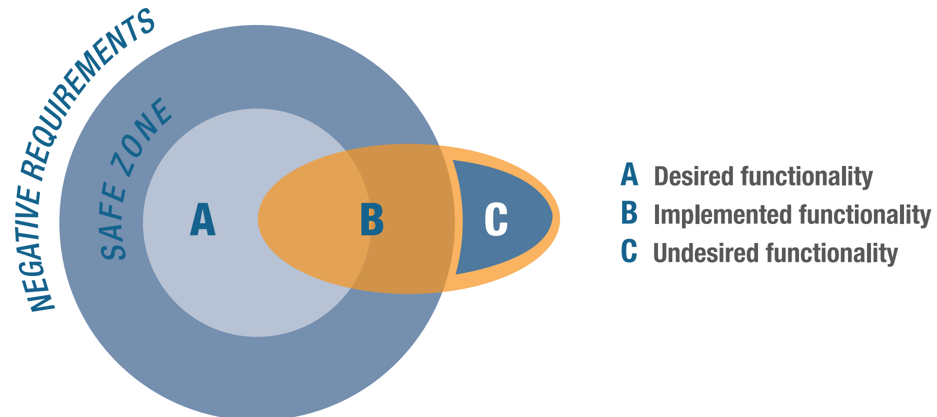

Figure 1. Different types of software functionality.

The "safe zone" in figure 1 is the safety margin between robust and unrobust software. Its thickness depends on the requirements, complexity, software process, programming language, environment, human factor, etc.

1.2 Protocol specification as a source of undesired functionality

In protocol implementations, undesired functionality results partly from poor specifications and their interpretation. Protocols are not born equal. Many specifications are written in semi-formal way – not quite machine readable and barely human readable. Parser generators cannot be fully utilised and implementations are more likely to contain human made errors. The parser generators themselves may also introduce undesired functionality, as they also can have flaws or be used in environments not designed for.

Furthermore, everything in the specification that

a) is not specified, defined or covered
b) is specified poorly, wrong or differently
c) is complex or difficult
d) resembles syntax or semantics known to have caused undesired functionality before

is a potential source of undesired functionality. Negative black-box testing methods should take advantage of any such lead.

2 Codenomicon test design

In Codenomicon robustness testing, a protocol mini-simulation is built. Mini-simulation is a minimal protocol implementation to achieve the job at hand: generating test cases. The mini-simulation is augmented with anomalies, exceptional elements, designed to reveal the undesired functionality in target protocol implementations. The anomalies are alternative to valid elements. Test cases are generated automatically by permutating anomalies and valid elements. [Kaksonen01]. Generated test suites are documented and packaged into Codenomicon Test Tools.

© Codenomicon 2006. All rights reserved.

TOPIC Codenomicon robustness testing
– handling the infinite input space while breaking software

Input level	In SIP	Target in software
Interface	UDP or TCP port 5060	Software / platform
Protocol	SIP	Stack / parser
Dialog	INVITE-BYE	State machine
PDU	INVITE message	Object / struct
Structure	Request-Line	Method / function
Primitive	"INVITE"	Variable handling

Table 1: Different levels of test design, their typical targets and SIP examples.

The mini-simulation is a six-level model, starting at interface level and ending with the primitive building blocks of an individual protocol data unit (PDU). Anomalies can be used at each level to find undesired functionality in different parts of the target software. The levels and their typical counterparts in the target software are listed in table 1.

2.1 Interface level

The interface level consists of the necessary functionality to communicate with the target protocol implementation. Interfaces range from sockets and files to command-line prompts. For example, Codenomicon SIP (Session Initiation Protocol, [RFC3261]) test tool needs to be able to send and receive UDP (User Datagram Protocol, [RFC768]) and TCP (Transmission Control Protocol, [RFC793]) packets.

Although the test cases are not explicitly designed to do so, they may find bugs at the transport interface implementation. For example, the Codenomicon SIP test tool might break a poorly implemented TCP/IP stack provided by the underlying platform. But in general, it is the languages of the interface, protocols, that form the basis of the test case design. To cover an interface, a test tool for each protocol understood by the interface would be needed.

2.2 Protocol level

The protocol mini-simulation consists of a subset of syntax and semantics of the selected protocol. Syntax is formally defined with BNF (Backus-Naur Form) or ASN.1 (Abstract Syntax Notation 1) metalanguages. Semantics, such as length and checksum calculations or encryption, is implemented with semantic rules. [Kaksonen01]. The whole protocol is not implemented in the simulation. Typically, one protocol entity is selected as the target and the other end is simulated.

2.3 Dialog level

A dialog is sequence of sent and received PDUs. Its purpose is to make the target software go through certain states. The dialog represents a context in which to interpret PDUs. Typical dialogs include single PDUs, various handshakes and error sequences.

© Codenomicon 2006. All rights reserved.

Dialog level anomalies include

* removing or repeating PDUs
* reordered or out-of-context PDUs
* incomplete dialogs

which all, more or less, attack the state handling portion in the target software. Especially many incomplete dialogs can cause degradation or denial of service due to resource consumption.

In Codenomicon SIP Test Tool, for example, there is an INVITE-BYE dialog. The test tool first sends an INVITE PDU, then waits for 200 OK from the target software, ACKs it, and terminates the call with BYE. By removing the BYE and repeating the dialog, we can create a massive amount of active calls.

2.4 PDU level

A protocol data unit (PDU) is a message of a given protocol. A PDU forms a context in which its structure is evaluated. PDUs are usually parsed into objects or structs in the target software. A typical PDU comprises of protocol-specific control information, header, and payload which may contain PDUs of upper layer protocols. Protocols are generally divided into binary and text-based protocols according to their PDUs.

The SIP INVITE message is a fairly typical text PDU, composed of header (Request-Line and multiple message-headers) and payload (message-body). In Codenomicon SIP Test Tool, the default payload in the message-body is a PDU of Session Description Protocol (SDP, [RFC2327]).

Anomaly-wise, PDUs are collections of structures in context of dialogs. Any anomaly changing the context, such as removing or repeating a PDU, is already covered at the dialog level. On the other hand, the internal PDU structure is mutated according to structure level anomalies, discussed next.

2.5 Structure level

PDUs are generally built up of three fundamental kinds of elements:

* type (T)
* length (L)
* value (V)

Type indicates what kind of value is expected, length how large it will be, and the value contains the payload. The value can be either a primitive or another TLV, in which case TLV structures continue until primitive values are met. [Kaksonen04].

TLV can be explicit or implicit. For example, the Request-Line in SIP INVITE is expected to be the first line of the PDU but it has no explicit type identifier. Its length is determined by the terminating CRLF and its value contains TLV substructures. For INVITE message the Request-Line is specified as [RFC3261]:

Request-Line = Method SP Request-URI SP SIP-Version CRLF

where

© Codenomicon 2006. All rights reserved.

Method = “INVITE”
Request-URI = SIP-URI / SIPS-URI / absoluteURI
SIP-Version = “SIP” “/” 1*DIGIT “.” 1*DIGIT
SP is white space
CRLF is a line feed
DIGIT is one US-ASCII coded character from 0x30 to 0x39

Structure level anomalies mutate and break TLV variations and their dependencies. They look fairly similar to dialog level anomalies:

* removed or repeated structures
* reordered or out-of-context structures
* incomplete or underflowed structures
* deep recursion or infinite loops

Because the possible combinations are endless, single permutations are mostly favored to keep the amount of test cases within sane limits. Already the SIP Request-Line can be mutated in several ways. Any of the three tokens may be reordered, the whole line left out, the terminating CRLF repeated, etc.

The more the structure is mutated, the more likely it becomes that the target parser altogether rejects the PDU because it no longer resembles the protocol. Keeping things small and subtle may yield
better results, especially when human made, not parser generated code is reached. Structure level test design is also adjusted according to the protocol.

Certain structures within or across protocol families (context-free) are known “death zones”. When mutated, they surprisingly often reveal clusters of bugs in implementations. For example, deep recursions are effective against embedded systems due to their limited stack space [Hayrynen]. Infinite loops are easily constructed and work well with file formats using offsets to find next elements.

2.6 Primitive level

Primitives come in many shapes: strings, characters, bytes, bit-patterns, etc. One could argue that in binary-based systems the only primitive is the bit -- everything can be constructed out of zeroes and ones. Whether something is a primitive or a structure is often a matter of viewpoint. For example an URI (Uniform Resource Identifier, [RFC2396])

http://www.nosuch.invalid

may be handled as a primitive string in a logging function. But when entered as the destination address in a web browser, it will first go through input sanity checking until a HTTP request is sent. In black-box testing, it is good to remember that the developers usually define primitives according to the specifications.

The primitive level anomalies consist of breaking and mutating individual T, L and V elements. They are constructed with extended boundary-value analysis and equivalence partitioning. T anomalies generally break the context in which length and value are evaluated. L anomalies will lie the length of the value, possibly causing busy loops, extreme memory allocation, etc. V anomalies are exception-

© Codenomicon 2006. All rights reserved.

ally crafted or sized values, that may have not been considered during implementation. Typical results include buffer overflows, memory corruption, busy loops, etc. Because implementations differ, the exact effects of an anomaly are impossible to know beforehand.

All primitive elements can contain:

* generic bit pattern anomalies
* T, L or V specific bit pattern anomalies.

Variable size elements can in addition be

* overflowed
* underflowed

without breaking the PDU structure (too much). String and character set specific anomalies include

* Encoding anomalies
* Escaping anomalies
* C-style format string anomalies
* ASCII control character anomalies.

On SIP Request-Line, there are three tokens: Method, Request-URI and SIP-Version of which only the Method can be considered a primitive, a variable sized US ASCII string without a substructure. It also implies the type of the SIP message; many protocols are full of these kind of dependencies. Thus we would exercise the Method field with variable length string anomalies and all the Method values possible in SIP.

Mutating a single primitive at a time is preferred, but double permutations are sometimes effective, especially TL, TV, LV, LL and VV pairs. For example, if an image file format has elements for horizontal and vertical size, it makes sense to permutate them together. However, too greedy selections easily explode the amount of test cases.

2.7 Multi-leveled failures

Bugs having complex activation patterns or subtle failure modes remain hard to detect with robustness testing. Failures caused by anomalies can propagate or trigger elsewhere in the target system. It is not uncommon for a protocol stack to parse and store an anomaly, only to be read by an unrobust application. For example, consider an framework that offers web services only for certain web browsers. Supposed the framework has weak User-Agent handling, a following HTTP request might trigger a buffer overflow in the framework when it queries the value:

```
GET / HTTP/1.0
Host: targethost:80
Accept: text/html
Accept-Encoding: gzip, compress
Accept-Language: en
User-Agent: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa... [33000 x a]
```

Anomalies can also travel through the input handling and application logic, causing failure only until output processing. If our web framework logged all the User-Agents and providing its input handling was robust, the same anomaly could cause buffer overflow in the logging phase.

© Codenomicon 2006. All rights reserved.

TOPIC Codenomicon robustness testing
– handling the infinite input space while breaking software

3 Targeting critical areas in the input space

Any interface can be fed an infinite number of random inputs. Testing a piece of software with them would be an eternal and infeasible task. Codenomicon robustness testing exploits the fact that some tests make more sense than others -- an approach that is effective against the finite realm of implementation-level errors. To achieve a balance in the amount of test cases, tests primarly focus on inputs and anomalies that have been shown to trigger problems in the past. Most test cases are based on publicly available computer security research from the past 20-30 years. But a large portion of them also rely on our in-house experience in breaking software, and new kinds of inputs are constantly pioneered.

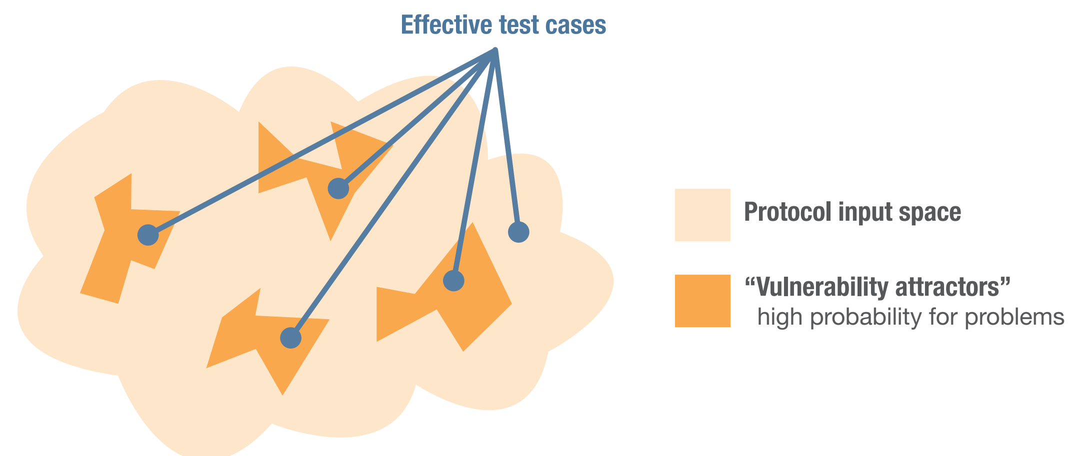

Figure 2. Robustness testing targets critical areas in the input space.

A good representation of input space raises the robustness baseline beyond the level of public threat in breaking software. Robustness test coverage requirements are different from traditional testing. The aim is to find the most critical problems in the software (figure 2). From a security standpoint these problems are in components that are easily accessible to attackers. Routines buried deep inside protocol logic may not be that security-critical. The most critical and obvious vulnerabilities are tested first and the elaborate, complex and rare techniques come second.

Therefore, more emphasis is put on covering every primitive element in PDUs before access control, authorization, session validation, encryption, etc. Testing the first PDU in a dialog is always the most important task. Gradually, the tests are expanded around this primary part of the input space, achieving better and better overall input coverage, but at the same time progressing away from the "ground zero", the epicenter of critical software problems.

An anomaly can be more or less effective depending on when and where it is placed. One way to find suitable "protocol surroundings", eg. pieces of structures in relation with the anomaly, is to cross-check inputs that cause problems (mostly test tool specific) with the anomalies behind the cause (mostly test tool independent). Successful combinations of anomalies and surroundings are extracted, and, as many protocols share similar structures, these combinations can be reused. Anomalies can also be tracked by scoring. Each time a problem is found with an input and assuming an anomaly can be identified as the root cause, the anomaly is awarded a point to its cumulative score. High score anomalies are then applied in the early phases of the test design.

© Codenomicon 2006. All rights reserved.

From the input perspective, the achieved test coverage depends not only on the anomalies, but also on the combination of valid values. Changing valid values may change the code coverage. Including more optional structures in the tested PDU may well increase it. But some sections of the target software will usually remain unreachable with a given interface and protocol [Rontti]. The environment, used compiler, the internal state of the target software and platform of course have their share on the outcome of the tests. All in all, even 100% code or input coverage does not prove that all possible behaviour of the software has been observed.

4 Conclusions

Codenomicon robustness testing is a systematic black-box method capable of revealing robustness problems and undesired behaviour in protocol implementations. Rather than relying on randomly generated inputs, the input space is carefully partitioned and augmented with anomalies, focusing on the primitives and structure of the tested protocol. Exploiting weaknesses in protocol specifications and knowledge on software vulnerabilities yields highly effective test cases. The aim is to find the most critical problems and to increase software quality beyond the level of public threat in breaking software.

© Codenomicon 2006. All rights reserved.

TOPIC Codenomicon robustness testing
– handling the infinite input space while breaking software

References

[Beizer]
Beizer B. (1990) Software Testing Techniques, Second Edition, ISBN 0-442-20672-0.

[Codenomicon]
Codenomicon Ltd. http://www.codenomicon.com/.

[Eronen]
Eronen J., Laakso M. (2005) A Case for Protocol Dependency. In proceedings of the First IEEE International Workshop on Critical Infrastructure Protection. Darmstadt, Germany. November 3-4, 2005.

[Hayrynen]
Hayrynen A. (2001) Vulnerability Assessment of Embedded Network Application Software. University of Oulu, Department of Electrical Engineering. Diploma thesis, 65 pages.

[Kaksonen00]
Kaksonen R., Laakso M., Takanen A. (2000) Vulnerability Analysis of Software through Syntax Testing. http://www.ee.oulu.fi/research/ouspg/protos/analysis/WP2000-robustness/.

[Kaksonen01]
Kaksonen R. (2001) A Functional Method for Assessing Protocol Implementation Security (Licentiate thesis). Espoo. Technical Research Centre of Finland, VTT Publications 447. 128 p. + app. 15 p. ISBN 951-38-5873-1.

[Kaksonen04]
Kaksonen R. (2004) Destructive Data. Presentation at T2'04 conference. October 10, 2004.

[RFC768]
Postel J. (1980) User Datagram Protocol. RFC768. August 28, 1980.

[RFC793]
Postel J., et al. (1981) Transmission Control Protocol. RFC793. Information Sciences Institute. University of Southern California. September 1981.

[RFC2327]
Handley M., Jacobson V. (1998) SDP: Session Description Protocol. RFC2327. ISI/LBNL. April 1998.

[RFC2396]
Berners-Lee T., et al. (1998) Uniform Resource Identifiers (URI): Generic Syntax. RFC2396. Xerox Corporation. August 1998.

[RFC3261]
Rosenberg J., Schulzrinne H., et al. (2002) SIP: Session Initiation Protocol. RFC3261. June 2002.

[Rontti]
Rontti T. (2004) Robustness Testing Code Coverage Analysis. University of Oulu. Department of Electrical and Information Engineering. Master's Thesis, 52 p.

[Voas]
Voas M. J., McGraw G. (1998) Software Fault Injection. John Wiley & Sons Inc. 1998. ISBN 0-471-18381-4.

© Codenomicon 2006. All rights reserved.