



Codonomicon whitepaper:

# Fuzzing in the SDL



- 1 Introduction
- 2 Microsoft SDL
- 3 Pre-SDL Requirements: Security Training
- 4 Phase 1: Requirements
- 5 Phase 2: Design
- 6 Phase 3: Implementation
- 7 Phase 4: Verification
- 8 Phase 5: Release
- 9 Post-SDL Requirement: Response
- 10 NOTES

CODENOMICON Ltd. | [info@codenomicon.com](mailto:info@codenomicon.com) | [www.codenomicon.com](http://www.codenomicon.com)

Tutkijantie 4E | FIN-90570 OULU | FINLAND | +358 424 7431  
10670 North Tantau Avenue | Cupertino, CA 95014 | UNITED STATES | +1 408 252 4000

---

PREEMPTIVE SECURITY AND ROBUSTNESS TESTING SOLUTIONS

# 1 Introduction

The Security Development Lifecycle (SDL) is the industry-leading software security assurance process. It was originally created by Microsoft in 2004 and since then, it has been used to achieve measurable security improvements in their flagship products. Microsoft's own experiences with SDL and the impact it had on the security of its flagship products clearly demonstrate that a proactive approach to security in software development significantly improves product security. However, even though, leading software companies utilize best practices, the number of security issues continues to grow. Application layer attacks are becoming more frequent and they are starting to pose a real threat to your customers and sensitive information.

Software vulnerabilities are often found in third party products and in applications and software running on top of the operating system components. Thus, it is increasingly important for software developers to adopt practices such as the SDL. Nowadays, the majority of security flaws are introduced by third party software developers, therefore best product security practices need to be endorsed by the entire industry to really promote software security. According to a IBM x-force 2008 security report, only 11% of security vulnerabilities are caused by the five leading software vendors. Thus, the wider adaptation of secure software development practices is critical for improving the overall security.

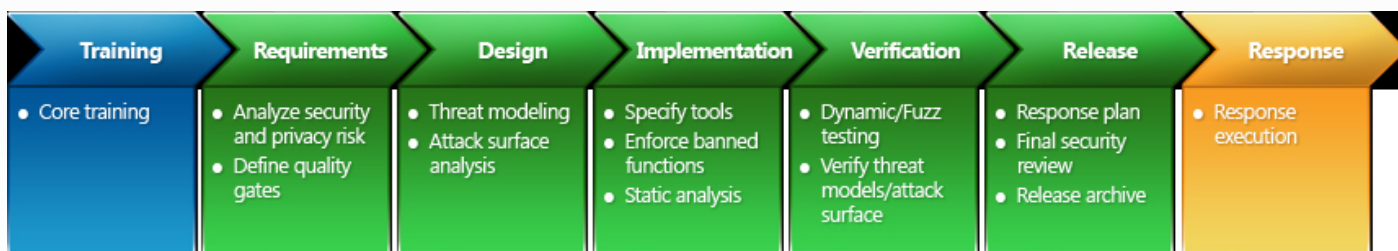
As a member of the Microsoft SDL Pro Network, Codenomicon helps development organizations adopt automated security testing (fuzzing) into their SDL and embed security and robustness into their software and development culture. In addition, Codenomicon assists you in implementing SDL in their own product development environment and helps them develop more secure applications and reduce the risks of malicious and costly attacks. Codenomicon specializes in fuzzing, and provides tools, training and consulting in the area of software security. Codenomicon has more than ten years of experience and expertise with fuzzing methodology and various fuzzing technologies. Codenomicon has helped its customers integrate fuzzing into various forms of software development processes.

Codenomicon's model based fuzzers are the most effective tools for testing network protocols and different file formats. Codenomicon's ground breaking method of combining model based fuzzing with proprietary XML or network capture based samples.

For general information regarding various types of fuzzers, see Buzz on Fuzzing white paper at: <http://www.codenomicon.com/products/buzz-on-fuzzing.shtml>

# 2 Microsoft SDL

SDL security activities are divided in seven phases (Figure 1).



We will next go through each step, explaining how they can influence adoption of fuzzing and other automated security testing practices.

### 3 Pre-SDL Requirements: Security Training

Background reading:

<http://msdn.microsoft.com/en-us/library/cc307407.aspx>

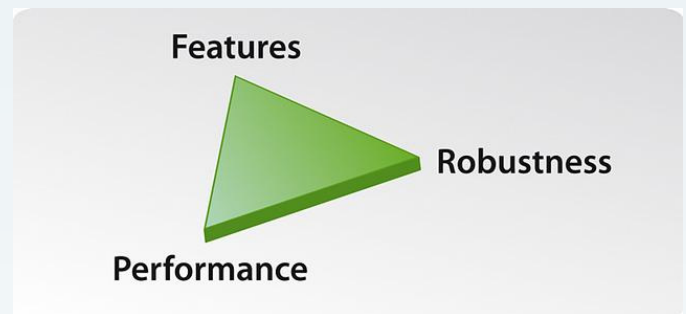
In regards to fuzzing in the SDL, these are the three key topics for training:

**Attack Surface:** The most important pre-requisite for fuzzing is identifying, minimizing, and prioritizing the attack surfaces. In black-box testing, this process is often simplified by assuming that only identification and interface and protocol analysis are performed on attack surfaces. However, before even starting the requirements step of the development life-cycle, you need to understand the concepts of communication interfaces: message structures (ASCII, ASN.1, XML) and sequences (Client-Server, Request-Response, Dynamic Message Sequences, Stateless and Stateful Sequences). The more complex the attack surface is the more error-prone the resulting software will be, and the more resources you need to allocate to security testing and fuzzing. All formats used in the messages should be identified, for example, all the media formats in HTTP requests, constitute their own separate attack surfaces into the system.

**Threat Modeling:** When the attack surface is identified and analyzed, threat modeling (from a fuzzing perspective) becomes fairly straight-forward. If the interfaces use a syntax like XML, you need to understand the anomalies attackers use to attack XML structures. If the architecture contains back-end databases, then you need to know the threats posed by SQL injection, and adapt your own fuzz tests accordingly. Knowing the basics of threat modeling will improve your fuzz tests, indeed, without understanding the threats associated with different technologies, your fuzzing suites would probably lack some critical attack simulations.

**Security Testing:** Three types of functional security testing are always needed: Features, Performance and Robustness (Figure 2). The focus of fuzzing is robustness testing, i.e., validating that the software can withstand unexpected inputs. For fuzzing to be effective, it needs to be a fully integrated and automated part of a build system. Understanding the maturity levels of fuzzers helps in choosing the right fuzzing techniques for the different phases of the SDL.

Attack surface and threat modeling are also used to understand the risks of the tested systems and to focus the attention on the right areas in the protocols. In testing, this information is used to implement test automation such as fuzzing, or acquire correct testing tools and to plan the tests.



### 4 Phase 1: Requirements

Background reading:

<http://msdn.microsoft.com/en-us/library/cc307412.aspx>

A key part of the requirements phase is identifying the critical interfaces in your software (or module), this is also essential step in determining the need for fuzzing. You can approach this matter by first answering the following questions, and then gradually move to more specific questions:

1. Will the software communicate directly with a network? (for example as a client, or as a server component)
2. Can the software be launched by network enabled software components? (for example IE can launch Notepad.exe)

If a software communicates with a network, it requires fuzz testing. Fuzz testing can take place either through a network interface, via files, or through an API. As a minimum requirement, fuzz testing should cover all the used network interface protocols, if network is used in system-internal and external communications.

Fuzzing is a critical part of security testing, and therefore the bug reporting capabilities of fuzzers need to be adapted into reports that fit your own bug tracking systems. A fuzz test will not consist of a single input, which triggers a flaw. Instead it can consist of a group of tests and millions of test cases. The verification of a discovered bug requires executing the same test setup partially or in full, often in the development and in testing organizations. People involved in accepting and prioritizing the found issues and in verifying the corrections need to know some basics on how different types of fuzzers work.

Fuzzers in regard to STRIDE: A fuzzer often demonstrates a failure up to a crash, or memory leak. Further analysis of the bug is not always required, but if time is dedicated to

analyzing the failure mode, more STRIDE flags can often be triggered. “D” as for “Denial of Service” label in STRIDE analysis of a bug found using fuzzers does not always mean that the flaw cannot be used to e.g. Elevate Privileges if time is spent on analyzing the exploitability of the flaw.

Analysis of the “Security Bug Effect” can also be laborious, and developer participation is needed in the analysis. A simple fuzz test report will not always indicate what type the flaw is. Developers often need to have the access to the test tool that produces the bug. For example, Codenomicon SW based fuzzing tools are easy to install and also to integrate into development environments, for reproduction by the programmers themselves.

## 5 Phase 2: Design

Background reading:

<http://msdn.microsoft.com/en-us/library/cc307414.aspx>

Design specifications should formally describe all the interfaces and the syntax of all messages or file formats used by the software. These specifications are the requirements for developing intelligent model-based fuzz tests. If possible, industry standards should be used in the most critical interfaces, because there is a ready supply of test tools for such interfaces.

Another critical design decision is regards the observability of tests during fuzzing. By using debug versions of software, the visibility of failures during fuzzing process is improved by helpful assertions and debug messages.

To start testing as early as possible, submit your interface specification to the team or company (such as Codenomicon) responsible for creating the fuzz tests as soon as they are ready. While you implement the software, the test tool development team can focus on preparing the fuzz tests. If the specification cannot be shared or the SW is already implemented, traffic capture based fuzzing can be done to achieve a higher level of security in the software. Capture based fuzzing cannot reach all the execution paths of the code, as it is restricted to test the sample messages existing in the capture. Indeed, a careful analysis should be carried before capture based fuzzing to determine whether it is sufficient for testing the interface.

## 6 Phase 3: Implementation

Background reading:

<http://msdn.microsoft.com/en-us/library/cc307416.aspx>

To use fuzz testing effectively to test programs that use network or Internet communications, it is important to describe all the communications channels and ports, protocols, and common communications configuration options.

All deviations from the industry standards also need to be documented, and any proprietary extensions of protocols need to be formally described. These documents can remain internal, but they should be available for the team responsible for building the fuzz tests for the security critical interfaces.

If you need to support earlier versions of the software and older protocols, remember to document those to the interface specification documents. Fuzz testing is often very efficient against legacy versions of the protocols and file formats, and therefore it is critical to test all legacy code also in each new release of the software.

When implementing the software, pay extra attention to error handling code. Fuzzing tools can be helpful for programmers also in unit tests, as they are easy means of triggering most error handling sections in the code.

Remember observability. Even if you catch failures such as memory exceptions, you cannot quietly ignore them.

When fuzzing is integrated into the development process, it should be more iteration based: 1) Developer unit tests 2) Build and regressing tests 3) Acceptance test 4) And back to developer test for bug repair process. Software based fuzzing solutions can be used in the entire development cycle, whether the development follows waterfall style, evolving or agile development practices. In large scale software projects the verification step is unfortunately often pushed to integration testing or system testing. This is often too late for significant vulnerabilities requiring major changes to the overall design of the product.

# 7

## Phase 4: Verification

Background reading:

<http://msdn.microsoft.com/en-us/library/cc307418.aspx>

Note that before you go into the actual verification phase, you must already possess the fuzzing tools or suites, and they must be ready to be used. Developing or acquiring test tools too late in the process will delay the acceptance and release of the product. Most important question related to the preparation is: "How critical is the interface?" It is easy to build a quick ad-hoc fuzzer, but it is very hard to build a good and intelligent fuzzer that understands the interface and behaves correctly, to be able to proceed to test all states in the protocol. Immediately when the interface design is completed, the fuzzer development can start in parallel. Companies such as Codenomicon are specialized in building fuzzers for both standard and proprietary interfaces. Codenomicon's model based implementation and 10 years expertise in modeling protocols and building our test development platform make it possible to implement comprehensive test suites fast and flexibly.

Functional testing, fuzzing among them, is done by the QA or test team in the organization or project. First test should be performed and analysed by people, preferably by application area experts. For manual usage and to help the automated configuration, Codenomicon platform has a graphical user interface to configure, test and save test configurations. When the issues and quality of the SW is understood, the configurations for test automation have been created.

In critical projects, automated fuzzing can be configured to be executed for each SW build. Nightly builds can be run with optimized test case amounts and weekly or bi-weekly builds with the full test set. Codenomicon provides a CLI interface for all test suite functionality to tailor it in different test environments. Designs can change rapidly for more efficient implementations, new features are integrated, issues are fixed, different execution configurations, especially in embedded development the different HW environments, they all cause new issues to appear or uncover different issues in the same SW. Efficient test automation is the only way to make sure that new features do not add new vulnerabilities.

Each level of a large scale enterprise must perform the testing. SW release paths can be long, with different integration steps, and considerable time and money is saved by detecting the

issues as early as possible. Corrections are always easier close to the developer. A full fuzz test must be completed as part of the final acceptance test (a gate) in the end of the verification phase, against both debug and final "retail" build.

Detection of the weaknesses is critical step in the test process. Codenomicon platform has the capability to integrate to Microsoft and third party tools for vulnerability detection, execute external commands and user-built batch scripts that assist in bug detection at every stage in the test execution cycle, and in different stages of individual test cases. These can be utilized to find out additional information of the tested SW during the tests, and to e.g. restart the SW in failure situations or to generate bug tracker issues for findings.

Most important factor in testing is the reporting and reproduction of the discovered issues. Reporting needs to be detailed for the developers to understand the issue, and after correction the issue must be reproduced to verify the correction. Codenomicon tests are fully reproducible, and detailed logs and reports are generated for all test cases and detected failures. Trend analysis helps to understand the SW quality trend, giving more confidence in the release decision of the software.

Integrate all used fuzzing tools into your regression test. Most tools can be configured in regression testing mode, to not only test the problems that were found in earlier steps of the verification phase, but also around those problems. For example, verifying a buffer overflow regression problem with only one regression test will not notice quick fixes where the buffer size was just increased to pass that one single test case. At minimum, the entire test group (a set of test cases with the same purpose) need to be executed in regression test suite.

For testing any proprietary communication interface, look for the Codenomicon Traffic Capture Fuzzer:  
<http://www.codenomicon.com/defensics/traffic-capture-fuzzer/>

For testing any XML or SOAP interfaces or files, look for the Codenomicon Defensics for XML:  
<http://www.codenomicon.com/defensics/xml/>

For testing any industry standard communication interfaces in e.g. Web browsers or servers, Email clients or servers, or any other standard communication products, look for Codenomicon test suites for that communication domain:  
<http://www.codenomicon.com/products/test-suites.shtml>

## 8 Phase 5: Release

Background reading:

<http://msdn.microsoft.com/en-us/library/cc307420.aspx>

The FSR is the step where you review all the security activities completed in the previous phases of the SDL. In terms of fuzzing, the FSR is your opportunity to review the results of fuzzing in the verification phase.

## 9 Post-SDL Requirement: Response

Background reading:

<http://msdn.microsoft.com/en-us/library/cc307411.aspx>

Post-SDL activities including response and software maintenance can build up to be the biggest and most expensive part of any software lifecycle. Especially for operators and any big enterprise system customers, the maintenance phase is the most crucial part, with regression testing going on all the time and careful integration of new fixes and features to the system. This raises the importance of integrating used verification practices to Post-SDL requirements.

When responding to vulnerability disclosures by third parties, you need to be prepared to run third party fuzzing tool to reproduce the issues. Basic training on various types of fuzzers will help you analyze the findings and speed up the response process. Availability of Codenomicon Traffic Capture Fuzzer for all developers will allow them to not only test the reported vulnerabilities, but also test around them.

## 10 NOTES

This interpretation of the Microsoft SDL in regards to fuzzing was created by Codenomicon, and is based on Codenomicon's customer experiences regarding integration of fuzzing into various software development processes.



CODENOMICON Ltd. | [info@codenomicon.com](mailto:info@codenomicon.com) | [www.codenomicon.com](http://www.codenomicon.com)

Tutkijantie 4E | FIN-90570 OULU | FINLAND | +358 424 7431  
10670 North Tantau Avenue | Cupertino, CA 95014 | UNITED STATES | +1 408 252 4000