Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.2, Apr. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.2.279

소프트웨어 개발 수명 주기(SDLC)를 고려한 국내 무기체계 Fuzz Testing 적용 방법 연구

조 현 석,[1†] 강 수 진,[1] 신 영 섭,[2] 조 규 태[2‡]
[1,2]LIG넥스원 (선임연구원, 수석연구원)

A Study on the Application Method of Fuzz Testing to Domestic Weapon Systems Considering the Software Development Life Cycle (SDLC)

Hyun-suk Cho,[1†] Su-jin Kang,[1] Yeong-seop Shin,[2] Kyu-tae Cho[2‡]
[1,2]LIGNEX1 (Researcher, Senior Researcher)

요 약

현재 국내 무기체계 산업에서는 소프트웨어의 보안 취약점 제거를 위한 규정들이 제/개정되어 적용 중에 있다. 하지만 현재까지 발표된 규정은 알려진 소프트웨어 보안 취약점에 대한 대비만 일부 될 수 있을 뿐이고 알려지지 않은 소프트웨어 보안 취약점에 대해서는 무방비하다. Fuzz testing은 알려지지 않은 소프트웨어 보안 취약점을 분석하는데 가장 많이 사용되고 있는 테스트 기법 중 하나이다. 최근 연구 자료에서는 fuzz testing의 효율을 높이기 위해 fuzzer의 알고리즘 변형 또는 효과적인 seed 값 추출 등의 방법이 많이 제시되고 있다. 하지만 fuzz testing 기법을 SDLC와 연계한 연구 자료는 찾기 힘들다. 본 논문에서는 국내 무기체계 SDLC에서 산출되는 데이터를 fuzz testing에 적용함으로써 효율적인 국내 무기체계 취약점 분석 강화 방안을 제안한다.

ABSTRACT

Currently, regulations for removing security vulnerabilities in software have been enacted/revised and applied in the domestic weapon system industry. However, the regulations published so far can only be part of the preparation for known software security vulnerabilities, and are defenseless against unknown software security vulnerabilities. Fuzz testing is one of the most widely used testing techniques to analyze unknown software security vulnerabilities. In recent research data, many methods such as fuzzer algorithm modification or effective seed value extraction have been suggested in order to increase the efficiency of fuzz testing. However, it is difficult to find research data linking the fuzz testing technique to SDLC. In this paper, we propose a reinforcement method for efficient analysis of weaknesses in domestic weapon systems by applying the data generated by SDLC in the domestic weapon system to fuzz testing.

Keywords: Weapon System, SDLC, Fuzz Testing, Security Vulnerability Analysis

I. 서 론

Microsoft는 개발 초기단계부터 보안 취약점을 분석하고 제거하기 위해 자체 SSDLC(Secure Software Development Life Cycle)인 MS-SDL(Microsoft Secure Development Lifecycle)을 구축하였고 2004년부터 진행하는 모든 소프트웨어 개발 품목에 적용 중이다. 또한 미 국방부에서는 DoDI 8510.01[1]에서 규정한 바와 같이 RMF(Risk Management Framework)를 무

Received(12. 15. 2020), Modified(03. 16. 2021), Accepted(03. 16. 2021)
† 주저자, ambitihyun@naver.com
‡ 교신저자, kyutae.cho@gmail.com(Corresponding author)

기체계 전 수명주기에 적용하여 보안 취약점 분석과 제거 활동을 수행 중이다.

한국의 무기체계 개발 프로세스에서도 개발 단계에서 보안 취약점을 분석하고 제거하기 위한 활동을 발전시키기 위해 관련 규정들을 제/개정 하고 있다. 대표적으로 '무기체계 소프트웨어 개발 및 관리 매뉴얼[2]'에서는 미국 MITRE 기관에서 관리하고 있는 CWE(Common Weakness Enumeration) 항목과 한국인터넷진흥원에서 제정한 시큐어 코딩규칙을 자동화 도구를 사용하여 분석 및 검증하고 있고, '국방 상호운용성 관리지시[3]'과 '상호운용성 관리지침[4]'의 정보보호 항목에서는 오픈소스 취약점 제거 활동과 STRIDE(Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege) 기법의 위협 모델링, 시큐어 코딩규칙 적용을 제시하였다. 하지만 현재까지 국내 무기체계 관련 규정에서 제시하는 보안 취약점 분석 방법은 알려진 보안 취약점에 대한 분석만 수행할 뿐이고 알려지지 않은 보안 취약점에 대한 분석 활동은 없다. 무기체계를 공격하는 해커는 대부분 개인해커가 아닌 국가에서 지원을 받는 전문 해커 조직이다. 이들이 발견한 무기체계의 보안 취약점은 알려지지 않은 zero-day 취약점일 확률이 크며 이는 zero-day 공격으로 이어질 수 있다. 또한 현 규정에서 제시하고 있는 보안 취약점 분석 활동은 알려진 취약점을 조치하기에도 미흡하다. 현 취약점 분석 방법은 정적시험(static test) 자동화 도구를 이용하여 보안 약점을 제거하는 방법으로 도구 분석기에 의존적이다. 때문에 도구 분석기가 보안 약점을 발견하지 못한다면 미 탐지된 상태로 군에서 운용될 수밖에 없다.

Fuzz testing이란 반복적으로 프로그램을 실행하면서 도구(fuzzer) 알고리즘에 의해 생성된 테스트케이스(test case)를 입력해보고 프로그램이 정상 동작을 수행하는지 확인하는 테스트 기술로써, 프로그램의 알려지지 않은 crash나 잠재적인 메모리 leak 등의 보안 약점을 발견한다. 앞서 언급한 MS-SDL과 RMF에서는 모두 fuzz testing 기법이 소프트웨어 개발 수명 주기(Software Development Life Cycle, SDLC)에 내재화되어 수행되는데 반해 국내 무기체계 관련 규정에는 아직 도입되지 않은 기법이다. 따라서 본 논문에서는 fuzz testing 기법을 국내 무기체계 SDLC에 적용하여 기존 취약점 분석을 강화하는 것을 제안한다.

무기체계 SDLC에 fuzz testing 적용 방안은 단순히 완성된 소프트웨어를 대상으로 fuzzing을 하는 것은 아니다. Fuzz testing은 소프트웨어를 구성하는 모듈의 범위를 줄여 테스트 대상을 좁히거나 처음 입력 값인 seed 값을 잘 선정해 주면 그 효과를 극대화 시킬 수 있다. 기존 무기체계 SDLC에서 산출되는 데이터를 이용하면 위험도가 높은 소프트웨어 모듈과 seed 값을 추출해 낼 수 있어 효율적인 fuzz testing을 수행할 수 있다. 또한 기술 부채(technical debt)의 이론상 소프트웨어 취약점이 잔존하는 상태에서 개발 단계가 진행될수록 해당 문제를 조치하는데 들어가는 기술 부채는 증가할 수밖에 없다. 따라서 무기체계 SDLC 초기 단계부터 fuzz testing을 수행한다면 잠재적으로 많은 비용을 줄이는데 효과적이다. 본 연구에서는 fuzz testing을 무기체계 SDLC에 적용하는데 있어 기존 산출 문서에서 활용할 수 있는 내용과 추가 반영해야 할 내용, fuzz testing 수행 시기, 종료 조건 등을 포함한다. 또한 국내 무기체계 소프트웨어 중 [2]에서 언급된 테스트 기준을 충족한 소프트웨어 2종을 선택하여 fuzz testing을 적용해 보고 그 결과와 효과를 확인해 본다.

본 논문은 서론에 이어, 2장에서는 관련 연구를 확인하고 분석한다. 3장에서는 본 논문에서 제안하는 fuzz testing 기법을 현재 무기체계 SDLC에 적용한 방법론을 제시하고, 4장에서는 개발이 완료된 무기체계를 대상으로 fuzz testing을 적용해보고 도출된 결과와 효과를 확인해본다. 끝으로 5장에서는 향후 연구 활동을 기술하고 마무리 한다.

II. 관련 연구

2.1 미군 무기체계의 보안 취약점 분석 정책

앞서 서론에서 언급한 바와 같이 미 국방부의 보안 보증 활동은 1983년부터 그 역사를 시작했다. 현재는 미 국방부 지침 문서인 [1]의 문서명인 'Risk Management Framework(RMF) for DoD Information Technology'에서 알 수 있듯이 NIST(National Institute of Standard and Technology)에서 제정한 RMF 보안 보증 모델을 미군 무기체계에도 동일하게 적용하고 있다. RMF란 제품 개발 시 초기 요구사항 분석부터 제품이 완성되어 운용하고 폐기될 때까지 제품의 전체 생명주

기에 보안성을 고려하는 SSDLC이다. RMF를 준수하기 위한 6개의 step별 상세 가이드는 'NIST SP(Special Publication) 800-37[5]'에 기술되어 있고, 이를 준수하기 위한 보안 요구사항 목록은 'NIST SP 800-53[6]'에 기술되어 있다. RMF에서는 시스템을 영향도 레벨에 따라 high, moderate, low와 같이 3단계로 분류하는데 보안 요구사항 항목 중 SA(System and services Acquisition policy and procedures) 11번 항목은 'Developer Security Testing And Evaluation'으로 3단계 모두 필수로 적용하는 항목이다. 이 요구사항 항목에는 dynamic code analysis의 방법 중 하나로 본 논문에서 적용하고자 하는 fuzz testing을 포함하고 있다.

2.2 국내 무기체계의 보안 취약점 분석 정책 동향

국내 무기체계의 보안 취약점 분석 정책은 여러 규정들로 발전해왔다. 먼저 2011년 제정된 '무기체계 내장형 소프트웨어 개발 및 관리 지침(현 [2]로 통합됨)'에서는 미국 비영리 기관인 MITRE에서 관리하는 CWE 항목을 정적시험 항목에 포함하여 자동화 도구로 분석하도록 하였다. 2015년도 '무기체계 소프트웨어 개발 지원에 관한 규정[7]'에서는 전장정보체계를 대상으로 한국인터넷진흥원에서 제정한 시큐어 코딩규칙을 준수하도록 하고 2016년에 [2]에도 그 내용을 반영하였다. 2017년 개정된 '국방전력발전업무훈령[8]'에서는 2015년 제정된 '국방사이버안보훈령[9]'에 대한 역할을 국군기무사령부(현 군사안보지원사령부)에 할당하였다. [9]는 무기체계의 보호 요구 수준을 분석하고 보안요구사항을 설정하는데 역할을 한다. 마지막으로 최근 2020년 개정된 [3]과 [4]에서는 상호운용성 평가 중 정보보호 항목에 오픈소스 취약점 제거 항목과 threat modeling 기법 중 STRIDE 기법 항목인 '신분 위장 위협 대응능력', '데이터 변조 위협 대응능력', '공격행위 부인 위협 대응능력', '정보유출 위협 대응능력', '서비스 거부(DoS) 위협 대응능력', '권한 상승 위협 대응능력'이 추가되었다.

이처럼 국내 무기체계에 대한 보안 취약점 분석 정책은 10년 내 많은 정책에서 다양한 항목이 추가되었다. 하지만 서론에서 언급한 바와 같이 모두 알려진 취약점에 대한 분석 활동일 뿐 알려지지 않은 취약점 분석에 대한 정책은 아직 제시된바 없다. 따라서 본 논문에서는 알려지지 않은 보안 취약점 분석 방법 중 하나인 fuzz testing을 국내 무기체계에 적용하는 방법을 제안한다.

2.3 국내 무기체계의 보안 취약점 분석 관련 연구

대부분의 국내 무기체계의 보안 취약점 분석과 관련된 연구 결과는 보안 약점인 CWE와 관련된 내용이지만 일부 연구 결과는 선진화된 개발 보안 프로세스를 국내 무기체계 개발 프로세스에 적용하자는 제안 연구가 있다. [10]과 [11]은 국내 무기체계 SDLC에 MS-SDL 프로세스에서 수행하는 보안 취약점 분석 활동을 적용하자는 연구다. Microsoft사의 MS-SDL은 소프트웨어를 주로 개발하는 민수회사이기 때문에 무기체계에서 다루는 통합 시스템 또는 임베디드 환경과 맞지 않다. 기존 SDLC도 다를 뿐만 아니라, 다품종 소량 생산하는 국내 무기체계와는 전혀 다른 산업 구조 형태를 띤다. 또한 두 연구 결과는 취약점 분석 테스트 항목만 언급되어 있을 뿐이고 적용 방법과 관련 산출물 등 세부적인 내용은 찾아볼 수 없다. [12]는 미국의 RMF를 국내 무기체계에 적용하는 것을 제안하는 연구 결과이지만 RMF 전체 프로세스에 초점이 맞춰져 있어 세부 취약점 분석 방법과 종류에 대해서는 없다. 본 논문은 미국의 RMF 정책을 국내에 적용시키기 위하여 [12]의 연구 결과를 상세화하는 데에도 일부 목적이 있다.

2.4 Fuzz testing 관련 연구

Fuzz testing은 소프트웨어 내부에 알려지지 않은 보안 약점을 테스트하는 대표적인 방법이고, 1988년에 그 개념이 등장하여 지금까지 활발히 연구가 진행되고 있다. Fuzz testing 기법은 현재까지 연구된 기법과 적용 방법이 매우 다양하여 관련된 국제 표준이 없다. [13]에서는 현재까지 fuzz testing의 발전 방향과 출시된 자동화 도구들의 특징을 정리하고 fuzz testing과 관련된 용어들을 표준화 하려는 노력을 하였다. 본 논문에서는 [13]에서 제시하는 용어 정의를 따른다. 대표적으로 fuzz testing은 테스트 자체를 말하고, fuzzing은 fuzz testing을 수행하는 상황을 말한다. 또한 [13]에서는 fuzz testing의 알고리즘을 간략화하여 제시하고 있다. 알고리즘은 크게 2 단계로 나눠진다. 첫 번

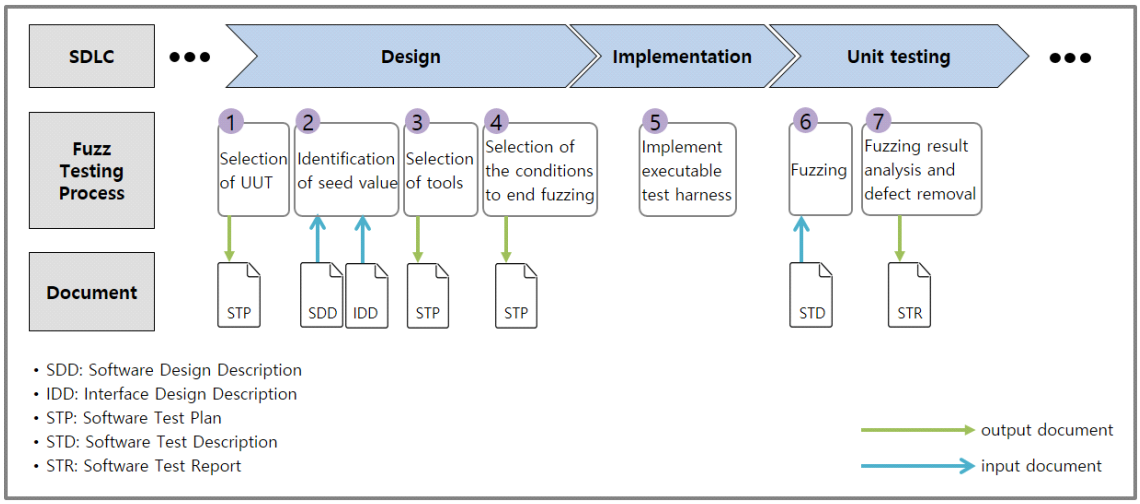


Fig. 1. Applying fuzz testing to each step of SDLC

째는 전처리(pre-process) 과정이고 두 번째는 fuzz iteration 내에서 4가지 과정(input generation, input evaluation, configuration update, schedule)이 순환하며 실행된다. AFL fuzzer를 예로 들면 전처리 단계에서는 커버리지 정보를 획득하기 위한 코드 주입 과정인 instrumentation이 수행된다. 다음 단계에서는 주어진 seed의 mutation을 통해 test case를 생성(input generation)하고 이 입력을 가지고 실행하여 [1]bug oracle을 통한 crash 여부를 판단(input evaluation)한다. 입력 값의 집합인 queue의 최적화 및 최소화를 진행하는 configuration update 과정 이후 마지막으로 queue 내에서 가장 선호하는 set을 선정하는 schedule 과정을 거치고 다시 input generation을 수행한다.

[14]에서는 최근 fuzz testing과 관련된 연구 결과들을 확인해 볼 수 있다. 효과적인 fuzz testing을 수행하기 위해 대부분의 연구 결과는 알고리즘 변형 및 효율적인 seed값 추출 방법을 제시하고 있다. 하지만, SDLC 단계별로 fuzz testing 활동을 제안하는 연구 결과는 찾기 힘들다. Fuzz testing을 SDLC에 반영하는 일은 매우 중요하다. [15]에서는 fuzz testing을 SDLC에 적용해야하는 8가지 이유에 대해 설명하였다. 때문에 국내 무기체계에 fuzz testing을 적용하기 위해서는 SDLC에 체계적으로 적용할 수 있어야 한다. 따라서 본 논문에서는 무기체계 SDLC에 fuzz testing을 효과적으로 적용할 수 있는 방안에 대해 제안한다.

III. 국내 무기체계 SDLC에 fuzz testing 적용 제안

국내 무기체계 SDLC는 미국의 무기체계 SDLC를 참고하여 구축했기 때문에 그 절차가 매우 유사하다. 하지만 미국 무기체계 SDLC에 fuzz testing을 적용하는 절차 및 상세 방법은 공개되고 있지 않아 참고할 수 없다. MS-SDL등 유명 업체의 SDLC 또한 개발 단계 중 fuzz testing을 수행하는 시기는 제시되어 있지만 구체적인 방법이나 산출물은 가이드 되고 있지 않다. 따라서 본 논문의 3장에서는 국내 무기체계 SDLC에 fuzz testing 적용 방안을 제안한다.

3.1 Fuzz testing 적용 프로세스

[16]에서는 fuzz testing 시작을 SDLC 단계 중 단위시험(unit test) 단계로 권고하고 있지만, 사전에 fuzz testing을 위한 설계가 수반되어야 한다. 설계 단계에서는 fuzz testing 대상이 되는 소프트웨어 단위(unit)를 식별해야 되고, fuzz testing 자동화 도구도 선정해야 하며, 시험 절차와 완료 조건이 계획되어 있어야 한다. 구현단계에서는 소프트웨어 소스코드 작성 시 테스트 단위로 선정한

1) system fetal signal 발생 여부를 모니터링 하고, unique crash를 판단하는 일련의 기능

UUT(Unit Under Test)가 별도 실행파일로 만들어져야 테스트 환경(test harness)이 만들어지기 때문에 이를 고려한 소스코드 작성이 필요하다. 또한 fuzz testing은 소프트웨어 단위시험 뿐만 아니라 통합시험 및 시스템 전체를 시험하고 운용하는 동안 소프트웨어가 업데이트 될 때마다 지속적으로 수행되어야 한다. Fig.1.에서는 무기체계 SDLC에 fuzz testing을 수행하기 위한 활동을 단계별로 제시한다. Fuzz testing 적용을 위한 각 단계별 상세 활동은 다음 장절에서 소개한다.

3.2 소프트웨어 설계 단계 활동

3.2.1 UUT 선정 (① selection of UUT)

무기체계 소프트웨어 설계 단계에서는 소프트웨어 구성요소인 CSC(Computer Software Component)를 분류하고 세부 기능에 따라 CSU(Computer Software Unit)를 식별한다. 소프트웨어를 구성하는 모든 CSU를 대상으로 fuzz testing을 수행하면 높은 보안 신뢰도를 가질 수는 있으나, fuzz testing을 위한 준비 과정과 수행 시간을 고려했을 때에는 비효율 적인 테스트가 될 수 있다. [2]에서는 소프트웨어 설계 단계에서 결함의 발생빈도(exposure), 영향성(severity) 및 제어가능성(controllability)을 기준으로 기능별 위험 분석을 수행하고 동적시험 기준을 설정하도록 가이드하고 있다. 이 가이드를 이용하여 위험도가 높은 모듈을 대상으로 fuzz testing 대상 UUT를 선정한다.

Fuzz testing 대상이 되는 UUT는 시험 환경인 test harness를 구성할 때 실행 가능한 파일로 만들어져야 한다. 이때 테스트 대상이 되는 UUT의 범위를 과도하게 크게 설정할 경우 fuzz testing을 수행하는데 많은 시간과 자원이 소요될 뿐만 아니라 fuzzing 효율성을 떨어트릴 수 있기 때문에 UUT의 범위를 적당한 범위로 선정하고 설계에 반영해야 한다. 선정한 UUT는 '소프트웨어통합시험계획서(STP)'에 기록한다.

3.2.2 Seed 값 식별 (② identification of seed value)

Fuzz testing은 입력되는 테스트케이스를 무작위로 추출한 다음 fuzzer 알고리즘에 따라 그 값을 변형시킨다. 따라서 fuzz testing의 충족도인 coverage를 만족하기 위해 꽤 많은 시간이 소요된다. 이를 보완하기 위해 대부분의 fuzzer들은 무작위로 생성하는 테스트케이스를 효율적으로 생성하기 위해 초기 테스트케이스 값을 입력할 수 있게 구현되어 있고 이를 seed 값 또는 seed 파일이라 한다. [17]에서는 입력되는 seed값에 따라 효율적인 fuzzing을 수행할 수 있음을 연구하였다. 따라서 fuzz testing을 할 때 초기 seed 값을 결정하여 넣어주는 일은 꽤 중요한 일이다. 하지만 UUT 별로 seed 값을 식별하는 데는 또 다른 시간과 노력이 필요하다. 무기체계 개발 과정에서 산출되는 기술 문서를 이용하면 seed 값을 효율적으로 추출할 수 있다. '소프트웨어설계기술서(SDD)'는 소프트웨어 형상항목에 대한 구조설계 및 상세설계 내용을 기술하는 필수 산출 문서다. 이 문서에는 함수별로 기능과 입/출력 값을 기술하게 되어있다. 또한 '인터페이스 설계기술서(IDD)'는 소프트웨어의 내/외부 인터페이스 설계 내용을 기술하는 필수 산출 문서이다. 소프트웨어 모듈 간 또는 통합된 소프트웨어 간에 주고받는 데이터의 형태와 값의 범위를 기술하게 되어있다. 여기서 함수의 입/출력 값과 인터페이스 간 데이터의 정보를 seed 값으로 이용한다면 seed 값을 찾는 추가적인 시간을 들이지 않고 test data를 활용할 수 있다.

3.2.3 도구(fuzzer) 선정 (③ selection of tools)

공개소프트웨어 및 상용으로 배포되는 fuzzer는 무수히 많다. [13]에서는 많이 쓰이는 fuzzer를 정리하였다. 이 중에서 무기체계 소프트웨어에 사용될 수 있는 fuzzer는 높은 인지도와 성능이 검증되어야 하며 테스트 수행 정도를 측정할 수 있는 coverage 데이터를 제공할 수 있어야 한다. Fuzzer를 선정할 때는 이해관계자의 합의가 필요하며 선정된 fuzzer는 '소프트웨어통합시험계획서(STP)'에 명시해야 한다. 본 논문 4장인 fuzz testing 적용 결과에 사용된 fuzzer는 공개소프트웨어로 배포되는 fuzzer 중 가장 인지도가 높은 AFL(American Fuzzy Lop) 도구를 사용하였다.

3.2.4 종료 기준 선정 (④ selection of the conditions to end fuzzing)

Fuzz testing 종료 조건을 설정한다. 종료 조건

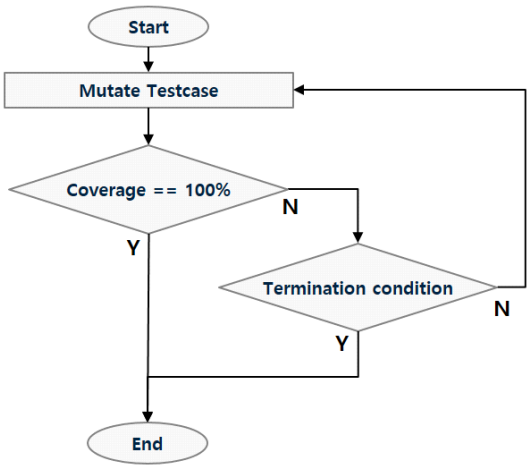


Fig. 2. Fuzzing flow

은 Fig.2.와 같이 fuzzing 결과 coverage 100%를 만족할 경우와 만족하지 않더라도 정해놓은 테스트 종료 조건(termination condition)을 만족할 때다. Coverage 값은 statement coverage, branch coverage 등이 될 수 있다. 테스트 종료 조건은 fuzzing 중 실시간 coverage 값이 기준 시간 내에 더 이상 높아지지 않는 경우(timeout) 또는 fuzzer에서 종료를 위한 여러 가지 부가 정보가 될 수 있다. Fuzzing을 위해 시간을 무한정 쓸 수 없기 때문에 coverage 값이 100%가 안 되더라도 fuzzing을 멈추는 것이 효율적인 테스트가 될 수 있다. 다만, coverage 100%를 달성하지 못하면 그 이유를 분석하고 '소프트웨어통합시험결과서'에 기록해야 한다. 지정한 coverage 종류와 이외 종료 조건은 '소프트웨어통합시험계획서(STP)'에 기록해야 한다.

3.3 소프트웨어 구현 단계 활동

3.3.1 실행 가능한 test harness 구현 (⑤ implement executable test harness)

단위시험 단계에서 fuzz testing을 수행하기 위해서는 fuzz testing 대상으로 선정한 UUT 별로 실행 가능한 테스트용 파일(test harness)이 생성되어 있어야 한다. 따라서 구현단계에서는 fuzz testing 대상 코드를 UUT단위별 실행 파일로 생성하기 위한 test harness 파일을 작성하고, UUT 내부로 fuzzing시 생성되는 테스트케이스를 입력할 수 있도록 코드를 구현한다. 이 때, 기능이 구현된 소스코드의 형상이 test harness 구현으로 인해 변경되지 않도록 해야 하기 때문에, UUT 단위로 소프트웨어 기능 구현 시 test harness 생성도 함께 고려해야 한다.

3.4 소프트웨어 단위시험 단계 활동

3.4.1 Fuzz Testing 수행 (⑥ fuzzing)

Fuzzing을 위한 하드웨어 및 소프트웨어 환경 구성도와 UUT별 fuzzing 절차는 '소프트웨어통합시험절차서(STD)'에 상세히 기술해야 한다. 이 과정에서 '소프트웨어설계명세서(SDD)'에서 추출한 UUT별 최초 입력되는 seed 값과 설계 단계에서 설정한 종료 조건을 포함해야 한다. 작성한 절차를 참고하여 fuzzing을 수행한다.

Fuzz testing은 단위시험부터 지속적으로 수행해야하며, 이 때 test 자원과 시간의 효율성을 높이기 위해서 CI(Continuous Integration) 시스템 환경에서 수행할 것을 제안한다. Fuzzing 수행은 testing 종료까지 많은 컴퓨팅 자원과 시간을 요구하기 때문이다. 사용자는 형상관리 서버에 소스코드 및 test harness 형상을 최신으로 업데이트하고 CI 시스템을 통해 테스트 요청을 하면 fuzzing 환경이 구축된 빌드 서버(build server)에서 종료조건까지 효율적으로 fuzzing을 수행할 수 있다. 따라서 CI 시스템을 구축하면 형상관리와 소프트웨어 테스트 자동화 등 많은 이점을 가져올 수 있다. 본 논문 4장에서 제시한 실험 결과 또한 CI 시스템을 이용하여 fuzzing을 효율적으로 할 수 있었다.

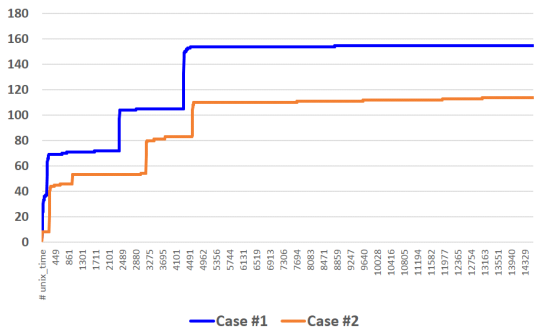


Fig. 3. Total crashing test case comparison according to seed value

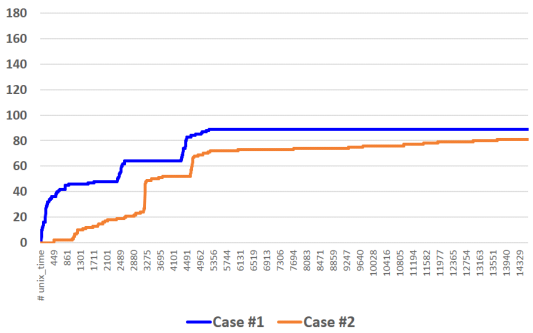


Fig. 4. Unique crashing test case comparison according to seed value

3.4.2 Fuzzing 결과 분석 및 조치 (⑦ fuzzing result analysis and defect removal)

Fuzz testing 결과에서 crash error를 발생하는 테스트케이스가 실제로 본 프로그램에 crash error를 발생시키는지 분석해야 한다. 분석 결과 테스트케이스가 결함을 발생시킬 수 있다면 해당 결함을 조치한다. 반면에 fuzzer 또한 오검출(false alarm)을 포함할 수 있다. 단위시험 레벨에서 테스트가 수행되기 때문에 전체 운용 환경에서 입력될 수 없는 값 등은 fuzzer에서 알 수 없기 때문이다. 오검출을 발생시킨 테스트케이스에 대해서는 소명자료를 만들어야 한다. Fuzz Testing의 수행 결과는 '소프트웨어통합시험결과보고서(STR)'에 상세히 정리한다.

IV. 무기체계 소프트웨어 Fuzz Testing 적용 결과

본 4장에서는 무기체계 소프트웨어 2종을 선택하여 fuzz testing을 적용해보고 그 결과를 제시한다. 소프트웨어 2종은 구분을 위해 이하 A, B로 명명한다. A, B 소프트웨어 모두 [2]에서 제시하는 시험기준을 자동화 도구를 이용하여 분석하고 조치 완료한 소프트웨어이다. 3장에서 제안한 fuzz testing 적용 프로세스에 맞춰 세부 내용을 제시한다.

4.1 UUT 선정

A 소프트웨어는 미들웨어 프레임워크로써 무기체계 소프트웨어 개발에 적용되는 공통 프레임워크이다. 이중 소프트웨어 코드 복잡도가 높은 XML 파싱(SW A-I)과 처리 모듈(SW A-II)을 fuzz testing UUT로 선정하였다. B 소프트웨어는 무기체계에 탑재되는 내장형 소프트웨어로써 현재 소프트웨어 단위시험을 완료한 소프트웨어이다. 이중 소프트웨어 외부 인터페이스로 GPS(Global Positioning System) 데이터를 연동하도록 구현된 모듈을 fuzz testing UUT로 선정하였다.

4.2 Seed 값 식별

A, B 각 소프트웨어의 '소프트웨어설계기술서(SDD)'와 '인터페이스설계기술서(IDD)'를 참고하여 초기 input data인 seed 값을 선정하였다. seed 값은 무기체계 보안상 본 논문에서 제시하지 않는다.

무기체계 소프트웨어에 대해 효과적인 seed값 선정을 위해 B 소프트웨어를 대상으로 '소프트웨어설계기술서(SDD)'에서 seed 값을 추출(case #1) 했을 때와 의미 없는 5자리 값을 seed 값(case #2)으로 넣었을 때 fuzzing 결과를 비교하였다. 두 케이스에 대하여 동일한 시간(x축)에 따라 Fig.3.은 fuzzing시 crash를 발생하는 총 테스트케이스 수(y축)를 비교한 결과이고, Fig.4.는 unique crash 수(y축)를 비교한 결과를 나타낸다. 그리고 Table 1.은 두 케이스의 fuzzing 결과를 비교하여 정리한 표이다. '소프트웨어설계기술서(SDD)'에서 seed 값을 가져왔을 때 crash를 발생시키는 테스트케이스 수가 약 35% 증가했으며, unique crash 수는 약 9.8% 증가하였다. case #1에서 발견한 테스트케이스는 case #2에서 발견한 모든 테스트케이스를 포

Table 1. Comparison of fuzzing results of two cases according to seed value

	Case #1	Case #2
Running Time [s]	14,570	21,265
Number of Crashing Test Case	155	116
Number of Unique Crash	89	81
Statement Coverage	89.7 %	56.7 %
Function Coverage	100 %	75 %

함하고 있었다. 또한, fuzzing 수행 시간에서도 의미 없는 seed 값을 입력한 경우 수행 시간이 45% 더 소요됨을 확인할 수 있다. 결론적으로 seed 값의 선정 결과에 따라 효과적인 fuzzing이 가능하며 그 값은 '소프트웨어설계기술서(SDD)'와 '인터페이스설계기술서(IDD)'를 참고하면 쉽게 추출할 수 있다.

4.3 도구(fuzzer) 선정

Fuzzer는 AFL을 사용했다. AFL fuzzer는 code coverage를 효율적으로 늘리기 위해서 유전 알고리즘을 사용하여 입력 값을 변이시켜 나가고 gray box testing을 지원해 주기 때문에 coverage 값을 쉽게 추출해 낼 수 있다. [13]에서도 알 수 있듯이 AFL fuzzer는 세계적으로 가장 많이 사용하는 fuzzer 중 하나고, AFL fuzzer를 근간으로 다양한 fuzzer들이 AFL 알고리즘을 변형하여 출시되고 있다.

4.4 종료 기준 설정

A, B 소프트웨어의 fuzzing 종료 기준을 선정하기 위하여 coverage 값은 statement coverage로 설정하고, 추가 AFL fuzzer에서 AFL_EXIT_WHEN_DONE 옵션을 적용하였다. AFL_EXIT_WHEN_DONE 옵션은 fuzzing 중 설정된 횟수 이상의 테스트케이스 변이를 발생 하였는데도 불구하고 coverage 정보를 통해 더 이상 새로운 path를 탐색하지 못하면 fuzzing을 종료(timeout)하는 옵션이다.

4.5 Fuzz Testing 준비 파일 작성

대상 UUT로 지정한 소스코드는 fuzzing을 위해 별도의 실행 가능한 파일인 test harness를 만들어야 한다. Test harness는 AFL에서 제공한 컴파일 환경으로 빌드를 수행시켜 gray box testing을 수행할 수 있게 만들고, 초기 입력 값인 seed 값은 사전 '소프트웨어설계기술서(SDD)'에서 추출한 값을 넣어 fuzzing 준비를 마친다.

4.6 Fuzz testing 수행 결과 확인 및 효과 분석

Table 2.와 같이 fuzz testing 결과 A-I 소프트웨어의 UUT는 사전 확인되지 않았던 crash를 발생시키는 테스트케이스가 135건, unique crash가 63건이 확인 되었고, A-II 소프트웨어의 경우 테스트 수행 시간이 월등히 높아 1,118건의 crash를 발생시키는 테스트케이스와 156건의 unique crash을 검출하였다. B 소프트웨어의 경우에는 테스트 기준 시간이 가장 짧음에도 불구하고 unique crash를 89건 검출하여 기준 시간 대비 가장 높은 검출률을 보였다.

Table 2. Fuzzing and analysis of results

	A-I	A-II	B
Running Time [s]	27,235	85,035	14,723
Number of Crashing Test Case	135	1,118	155
Number of Unique Crash	63	156	89
Statement Coverage	89.7 %	100 %	82 %
Function Coverage	100 %	100 %	86 %
Defect Example	Null Pointer Dereference	Buffer Overflow	Null Pointer Dereference

대표적인 crash를 발생시키는 테스트케이스를 확인해 보면 A-I 소프트웨어는 Fig.5.와 같이 XML 파일에 여는 태그 없이 닫는 태그만 있는 경우의 테스트케이스를 입력했을 때, Fig.6.과 같이 Null 포인터 역참조가 발생하였다. A-II 소프트웨어는 비정상적으로 큰 입력 값을 테스트케이스를 입력했을 때,

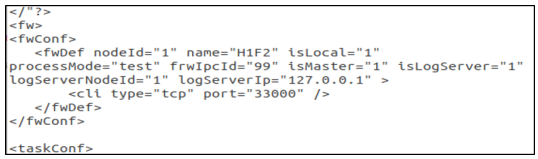


Fig. 5. crashing test case of A-I

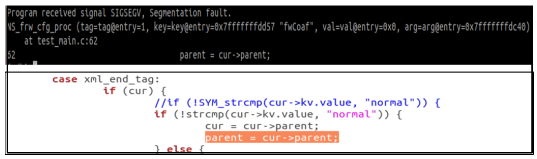


Fig. 6. defects from A-I

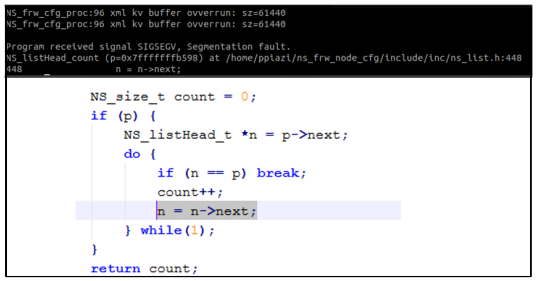


Fig. 7. defects from A-II

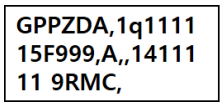


Fig. 8. crashing test case of B

Fig. 9. defects from B

Fig.7.과 같이 Buffer Overflow가 발생하였다. B 소프트웨어의 경우 GPS 메시지의 정상 유무를 체크하는 함수에서 Fig.8과 같이 체크썸 값이 누락되는 테스트케이스를 입력할 경우가 고려되지 않아 Fig.9. 와 같이 Null 포인터 역참조가 발생하는 것을 확인하였다.

산출된 결과를 토대로 두 가지의 효과를 확인할 수 있었다. 첫 번째로 최근 fuzz testing 관련 연구와 달리 알고리즘의 변형 또는 의미 있는 seed 값을 찾는데 노력을 들이지 않고 무기체계 SDLC에서 산출되는 데이터를 이용하여 fuzz testing의 효과를 극대화 시켰다. 두 번째로 SDLC 초기 단계부터 fuzz testing을 적용하여 발생 가능한 취약점에 대해 확인 하였다. 이는 무기체계 소프트웨어 보안 취약점을 조기에 확인하고 조치하기 때문에 보안성 확보를 위한 비용과 시간을 절약할 수 있다.

4.7 결론

결론적으로 현 무기체계 소프트웨어에 fuzz testing 적용 결과 치명적인 보안 약점을 발견하여 조치할 수 있었다. 발견된 보안 약점 대부분은 현 무기체계에서 적용하는 알려진 취약점에 대한 대비만으로는 분석할 수 없는 항목이었다. 본 실험을 통해 발견된 무기체계 보안 약점을 조치하지 않고 군에서 운용했다면 추후 발견하기도 쉽지 않을뿐더러 적군에 의해 취약점이 발생될 수도 있었다. 따라서 무기체계 소프트웨어에 보안성을 확보하기 위해선 본 논문에서 제시하는 fuzz testing 적용 제안을 필히 고려할 필요가 있다.

V. 향후 연구

무기체계 소프트웨어의 보안성 확보를 위해선 fuzz testing 뿐만 아니라 SDLC 전 단계에서 보안을 고려해야한다. 이미 미군에서는 RMF 프로세스를 무기체계 SDLC 전 단계에 적용하여 보안을 고려한 체계적인 개발 프로세스를 구축하고 적용 중이다. 본 논문은 한국형 RMF 프로세스를 구축하는데 있어 현재 수행중인 취약점 분석 활동을 강화하기 위해 fuzz testing 적용을 제안하였다. 한국형 RMF를 구축하는데 본 논문에서 제시한 fuzz testing 이외의 필요 연구항목을 더 발굴해야 하며 기존 SDLC를 발전시켜 나갈 필요가 있다.

또한, 무기체계의 보안 보증을 위하여 SDLC 전 단계에 보안 전문가 검토가 필요하다. 이를 위해선 전문화된 조직 신설 및 개발업체 자체 내 보안 전문가 양성이 필요하고 개발 단계별 보안 보증 활동을 SDLC 내에 반영해야 한다.

References

[1] "Risk Management Framework (RMF) for DoD Information Technology (IT)," DoDI 8510.01, Mar. 2014

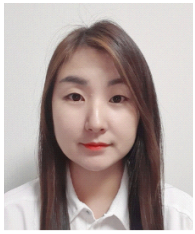
[2] "Weapon System Development and Management Manual," DAPA(Defense Acquisition Program Administration), Nov. 2018

[3] "Defence Interoperability Management Instruction," MND(Ministry of Nation

al Defense), No.2020-003, Jan. 2020
〔4〕 "Interoperability Management Guideline," DAPA No.673, Aug. 2020
〔5〕 "Risk Management Framework for Information Systems and Organizations: A System Life Cycle Approach for Security and Privacy," NIST SP 800-37 Rev.2, Dec. 2018
〔6〕 "Security & Privacy Controls for Federal Information Systems and Organizations," NIST SP 800-53 Rev.4, Apr. 2013
〔7〕 "Instruction for Supporting the Development of Weapon System Software" DAPA, No.626, Sep. 2020
〔8〕 "National Defense Work Instruction," MND, No.2040, Jun. 2017
〔9〕 "National Defense Cyber Security Instruction", MND, No.2234, Dec. 2018
〔10〕 Yeonoh Jeong, "A Study about Development Methodology for Ensure the Software Security of Weapon System," The Korean Institute of Information Scientists and Engineers, 2018(6), pp. 77-79, Jun. 2018
〔11〕 Woncheol Lee, Kanghyun Kim and Seunghyeon Lee, "A Study of Software Security of Embedded Weapon Software Development Lifecycle," The Korean Institute of Information Scientists and Engineers, 2016(12), pp. 92-94, Dec. 2016
〔12〕 Hyunsuk Cho, Sungyong Cha and Seungjoo Kim "A Case Study on the Application of RMF to Domestic Weapon System", Journal of The Korea Institute of information Security & Cryptology, 29(6), pp. 1463-1475, Dec. 2019
〔13〕 V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," CoRR, vol. abs/1812.00140, 2018. [Online]. Available: https://arxiv.org/pdf/1812.00140.pdf
〔14〕 Cheng Wen, "Recent Papers Related To Fuzzing" https://wcventure.github.io/FuzzingPaper/, Mar. 2021
〔15〕 Beyond Security, "To Fuzz or Not to Fuzz: 8 Reasons to Include Fuzz Testing in Your SDLC" https://blog.beyondsecurity.com/fuzz-testing-sdlc/, Sep. 2020
〔16〕 Adith Sudhakar, VMWare; Mohit Arora, Dell; and Souheil Moghnie, Norton LifeLock, "Focus on Fuzzing: Fuzzing Within the SDLC" https://safecode.org/focus-on-fuzzing-fuzzing-within-the-sdlc/, Sep. 2020
〔17〕 A. Rebert, S. K. Cha, T. Avgerinos, J. M. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing." 23rd USENIX Security Symposium, Aug. 2014.

〈저 자 소 개〉

조 현 석 (Hyun-suk Cho) 정회원
2014년 2월: 한성대학교 컴퓨터공학과 학사
2020년 2월: 고려대학교 정보보호대학원 석사
2014년 1월~현재: LIG넥스원 지능형SW연구소 선임연구원
〈관심분야〉 무기체계 소프트웨어 신뢰성/보안성 시험, 소프트웨어 테스팅



강 수 진 (Su-jin Kang) 정회원
2008년 2월: 경북대학교 전자전기컴퓨터 학부 졸업
2008년 3월~현재: LIG넥스원 지능형SW연구소 선임연구원
〈관심분야〉 무기체계 소프트웨어 신뢰성/보안성 시험, 소프트웨어 테스팅

신 영 섭 (Yeong-seop Shin) 정회원
2007년 2월: 충남대학교 전자전파정보통신 학사
2009년 2월: 충남대학교 전자전파정보통신 석사
2009년 1월~현재: LIG넥스원 지능형SW연구소 수석연구원
〈관심분야〉 무기체계 소프트웨어 신뢰성/보안성 시험, 소프트웨어 테스팅

조 규 태 (Kyu-tae Cho) 정회원
2002년 2월: 숭실대학교 컴퓨터학부 학사
2004년 2월: 한국과학기술원 전산학과 석사
2007년 2월: 한국과학기술원 박사 수료
2007년 2월~현재: LIG넥스원 지능형SW연구소 수석연구원
〈관심분야〉 인공지능, 머신러닝, 소프트웨어 설계