

도구와 기법 문서 (ALC_TAT.3)

- 고등급 보안 마이크로커널 개발 -

<제목 차례>

1. 개요	5
1.1. 문서의 목적	5
1.2. 문서의 구성	5
2. 위협모델링 도구	6
2.1. Microsoft Threat Modeling Tool	6
2.2. 위협 분석	11
2.3. 위협 분석 보고서 생성	11
3. 정형기법 도구	12
3.1. Isabelle/HOL	12
4. 개발 도구	16
4.1. C 언어	16
4.2. Gcc 컴파일러	16
5. 테스트 도구	20
5.1. Polyspace Bug Finder	20
5.2. AFL Fuzzer	24

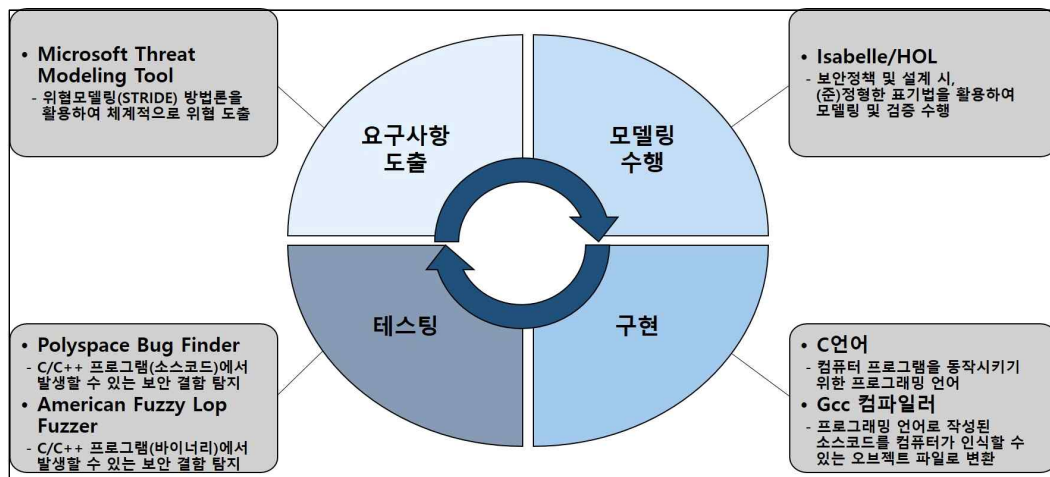
1. 개요

본 문서는 보안성이 향상된 ChibiOS인 TOE를 개발, 분석, 구현하는 데 사용되는 도구에 대한 상세 정보를 제공한다. 해당 문서를 통해 TOE를 개발, 분석, 구현하는데 사용되는 도구는 명확하게 사용할 수 있음을 입증해야 한다. 잘못 정의되거나 설치된 기능이 모호한 부정확한 개발 도구가 TOE 개발에 사용되면 결함을 발생시킬 수 있다. 따라서 개발자뿐만 아니라 third-party에서 사용한 모든 개발 도구를 식별하고 모호한 규정을 사용하는 도구를 사용하지 않음을 입증해야 한다.

1.1. 문서의 목적

우리는 TOE의 고등급(EAL 6 이상)보안 수준을 보증하기 위해 요구사항 분석, 설계, 구현, 테스트 각 단계에서 Microsoft Threat Modeling Tool, Isabelle/HOL, C언어, Gcc 컴파일러, Polyspace Bug Finder, American Fuzzy Lop Fuzzer를 각각 사용하였다. 해당 도구를 사용한 목적은 아래와 같다.

- 위협모델링 도구: 보안기능 요구사항을 체계적으로 도출하는 도구
- 정형기법 도구: 보안정책/설계에 대한 정형 명세 및 검증하는 도구
- 개발 도구: 구현에 사용하는 프로그래밍 언어와 컴파일러
- 테스트 도구: 소스코드 및 바이너리에 대한 정적 분석 도구 및 동적 분석 도구



[그림 1] TOE 개발에 사용된 개발 도구 개요

1.2. 문서의 구성

본 문서는 1장에서 본 문서에 대한 개요에 대해 서술하며, 2장에서 요구사항 분석 단계에 사용하는 위협모델링 도구에 대해 서술된다. 3장에서는 보안정책 및 설계 수행 시, 활용할 수 있는 정형 기법 도구인 Isabelle/HOL에 대해 소개한다. 4장에서는 구현 단계에서 사용되는 프로그래밍 언어와 컴파일러(컴파일 옵션 포함)에 대해 서술된다. 5장에서는 소스코드와 바이너리에 대한 테스트를 수행할 수 있는 도구를 서술하고 6장에서는 프로젝트 전체에서 각 문서들과 구현물의 형상을 관리하는 형상관리 도구에 대해 서술한다.

2. 위협모델링 도구

본 장에서는 요구사항 분석 단계에서 위협모델링을 통해 보안기능 요구사항을 체계적으로 도출할 수 있는 도구 중 하나인 Microsoft Threat Modeling Tool에 대해 서술한다.

위협모델링은 분석 대상에서 발생할 수 있는 보안 위협을 체계적으로 도출할 수 있는 방법론이다. 해당 방법론에 대한 연구는 1990년대 이후 본격적으로 진행되었으며, 마이크로소프트가 발표한 STRIDE가 가장 대표적인 위협모델링 방법이다.

STRIDE는 개발자 관점에서의 분석 방법으로 데이터 흐름도를 바탕으로 분석 대상에서 발생할 수 있는 위협을 식별할 수 있다. STRIDE를 통해 식별할 수 있는 위협은 Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege로 총 6가지이다. 해당 위협의 상세 설명은 아래 표와 같다.

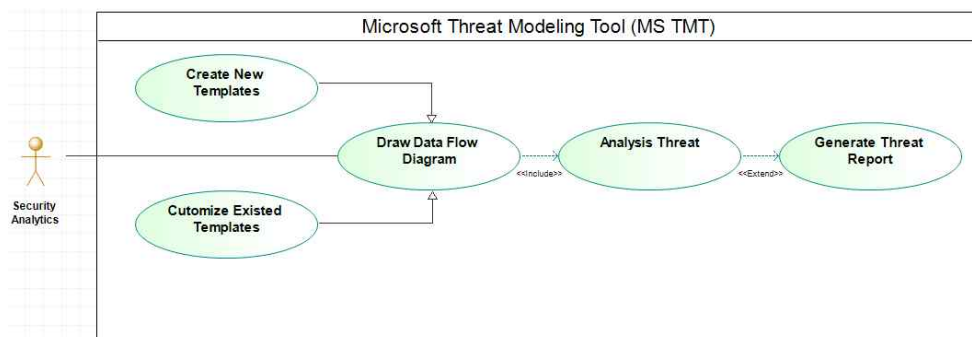
위협 유형	설명
S poofing	인증 또는 허가되지 않은 주체가 허가받은 주체로 위장하여 시스템에 접근하는 위협
T ampering	파일, 네트워크 패킷, 설정 데이터 등과 같은 다양한 형태의 기밀 정보를 고의적으로 변조하는 위협
R epudiation	특정 주체가 작업 수행 여부를 부인하는 위협
I nformation disclosure	대상 시스템의 중요 정보가 접근이 허가되지 않은 사용자에게 노출되는 위협
D enial of service	다수의 트래픽을 발생시키는 등의 방법을 통하여 대상 시스템의 정상적인 동작을 방해하거나 중지시키는 위협
E levation of privilege	접근 및 실행 권한이 없는 주체가 액세스 권한을 획득하여 특정 권한이 필요한 대상에 접근하거나 해당 권한으로 실행하여 시스템을 손상시키거나 파괴하는 위협

[표 1] STRIDE 위협 분류

2.1. Microsoft Threat Modeling Tool

Microsoft는 2004년에 STRIDE 위협모델링을 활용하여 보안 위협을 도출할 수 있는 Microsoft Threat Modeling Tool(이하, MS TMT)을 최초로 개발하였으며 현재까지 업데이트된 버전을 오픈소스 형태로 제공하고 있다.

MS TMT는 Microsoft 공식 사이트에서 설치를 진행할 수 있으며, 설치 이후 해당 프로그램을 실행할 경우, 아래와 같은 기능에 따라 위협모델링을 수행함으로써 보안 위협을 식별할 수 있다. 이와 관련한 상세 사용 방법은 아래 유즈케이스 다이어그램과 같다.



[그림 2] MS TMT 유즈케이스 다이어그램

2.1.1. 설치

MS TMT는 아래 URL에서 다운로드 링크를 제공한다. 해당 링크에 접속해서 화면 하단에서 다운로드 아이콘을 확인할 수 있다. 해당 아이콘을 클릭하면, PC에 MS TMT가 자동으로 설치되는 것을 확인할 수 있다.

■ URL: <https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling>

Threat Modeling

Threat modeling is a core element of the Microsoft Security Development Lifecycle (SDL). It's an engineering technique you can use to help you identify threats, attacks, vulnerabilities, and countermeasures that could affect your application. You can use threat modeling to shape your application's design, meet your company's security objectives, and reduce risk.

There are five major threat modeling steps:

- Defining security requirements.
- Creating an application diagram.
- Identifying threats.
- Mitigating threats.
- Validating that threats have been mitigated.

Threat modeling should be part of your routine development lifecycle, enabling you to progressively refine your threat model and further reduce risk.

Microsoft Threat Modeling Tool

The Microsoft Threat Modeling Tool makes threat modeling easier for all developers through a standard notation for visualizing system components, data flows, and security boundaries. It also helps threat modelers identify classes of threats they should consider based on the structure of their software design. We designed the tool with non-security experts in mind, making threat modeling easier for all developers by providing clear guidance on creating and analyzing threat models.

The Threat Modeling Tool enables any developer or software architect to:

- Communicate about the security design of their systems.
- Analyze those designs for potential security issues using a proven methodology.
- Suggest and manage mitigations for security issues.

The SDL Threat Modeling Tool plugs into any issue-tracking system, making the threat modeling process a part of the standard development process.

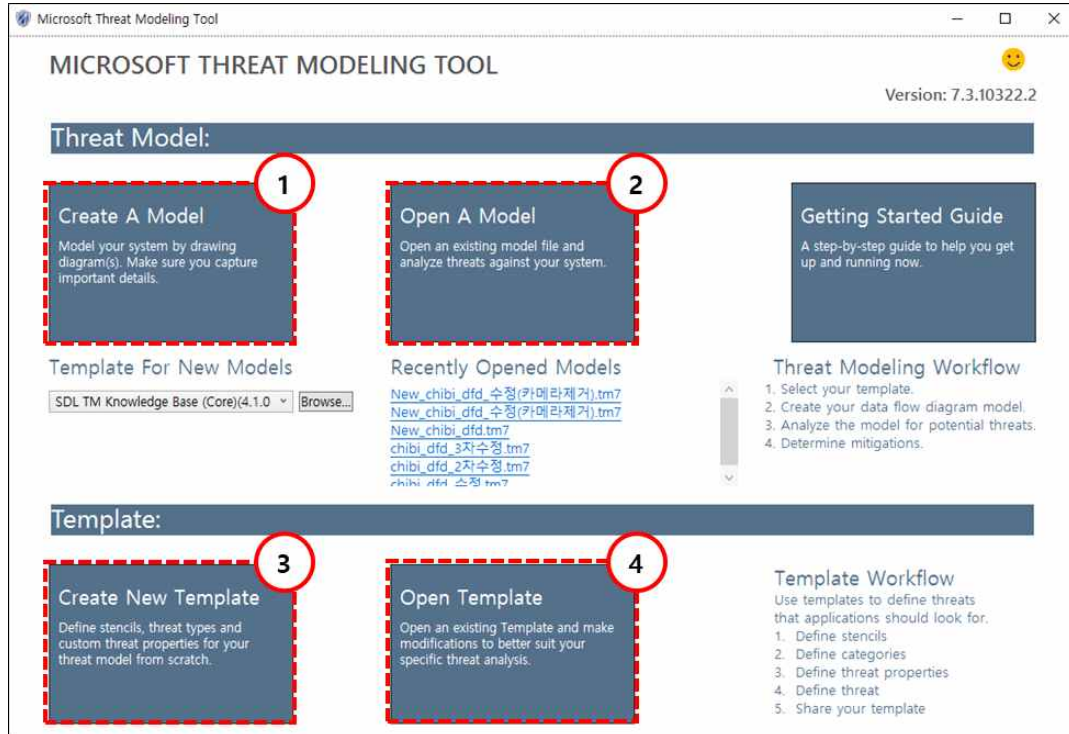
The following important links will get you started with the Threat Modeling Tool:

- [Download the Threat Modeling Tool](#)
- [Read Our getting started guide](#)
- [Get familiar with the features](#)
- [Learn about generated threat categories](#)

[그림 3] MS TMT 다운로드 화면

2.1.2. 실행

설치된 MS TMT를 실행하면 아래 그림과 같은 화면이 출력된다. 설치된 MS TMT의 버전은 7.3.10322.2임을 확인할 수 있다.



[그림 4] MS TMT 실행 화면



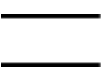
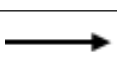

마이크로소프트에서 제공하는 기본 요소를 활용하여 데이터 흐름도를 작성하는 경우, ①, ② 기능을 활용한다. ①은 신규 위험모델링 프로젝트에서 데이터 흐름도를 작성하는 경우 활용되는 기능이다. ②는 기존에 존재하는 프로젝트에서 데이터 흐름도를 수정하거나 추가 작업을 수행할 때 활용되는 기능이다.

만약 분석하려는 시스템의 데이터 흐름이 마이크로소프트에서 제공하는 기본 요소로 표현이 안되는 경우 ③, ④ 기능을 활용하여 새로운 구성요소들로 구성된 템플릿을 작성할 수 있다. ③은 신규 템플릿을 생성하는 경우 활용되는 기능이고 ④는 기존 템플릿에 추가 작업을 수행할 때 활용하는 기능이다. 신규 템플릿을 통해 새롭게 시스템의 세부 구성 요소는 각 요소 명, 아이콘, 제약 조건, 성질, 세부 설명 등의 정보가 기입될 수 있다. 신규 구성 요소 내 기입된 정보를 바탕으로 MS TMT는 관련한 위험 정보를 제공한다.

이에 대한 자세한 설명은 ⑤ 기능을 통해 확인할 수 있다.

2.1.2.1. 데이터 흐름도 작성

위협모델링은 분석하려는 대상의 데이터 흐름을 도식화하기 위해 5가지 구성 요소를 활용한다. 데이터 흐름도를 나타내는 구성요소는 아래 표와 같다.

구성요소	기호	설명
객체 (Entity)		분석 대상에게 서비스 및 데이터를 요구하는 외부 주체(사용자, 기관, 서버 등)를 의미하는 심볼
프로세스 (Process)		시스템 내 수행되는 기능(모듈, 함수 등)을 의미하는 심볼
데이터 저장소 (Data Store)		임시 또는 영구적으로 데이터가 저장 및 검색될 수 있는 저장소(데이터베이스, 파일, 레지스트리, IT 제품, 종이 문서 등)를 의미하는 심볼
데이터 흐름 (Data Flow)		데이터 흐름도 내 요소(객체, 프로세스, 데이터 저장소) 간 정보 흐름을 의미하는 심볼
신뢰 영역 (Trust Boundary)		제품 내 접근 권한 및 정보를 처리하는 주체에 따라 분리된 영역을 의미하는 심볼

[표 2] 데이터 흐름도 구성요소

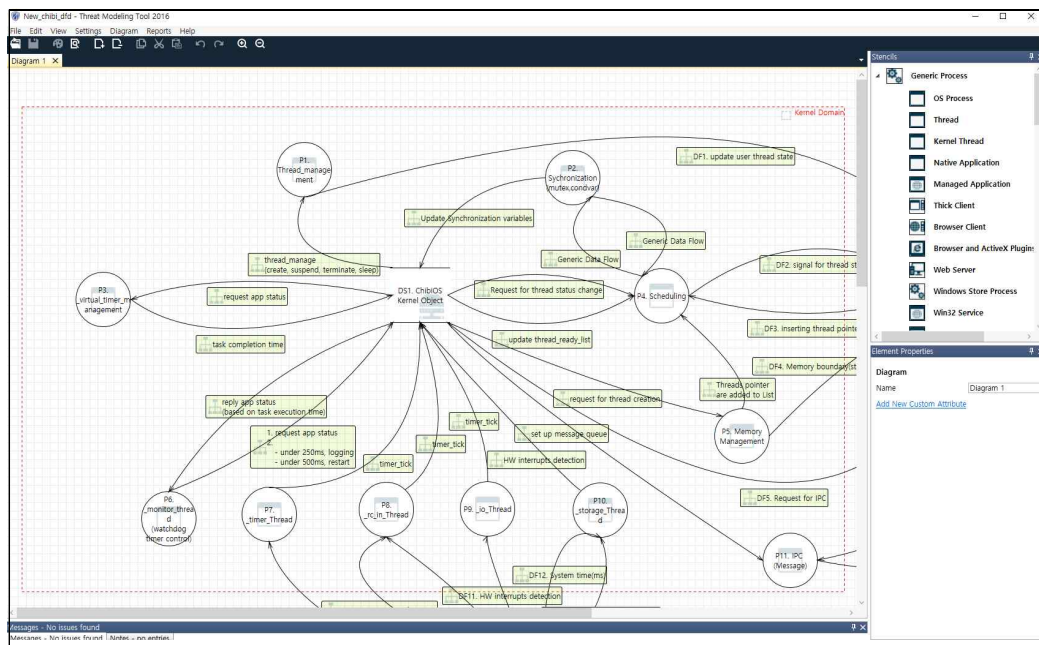
MS TMT의 경우, 각 구성 요소를 소프트웨어를 표현하기 위해 아래 그림과 같이 구체화하여 제공한다.

구분	MS TMT 구성요소	설명
객체	Browser	외부 웹 브라우저
	Authorization Provider	외부 인가된 서비스 공급자
	External Web Application	외부 웹 어플리케이션(포탈, 프론트 ed 등)
	External Web Service	외부 웹 서비스
	Human User	일반 사용자
	Megaservice	인터넷을 통해 제공되는 메가서비스
	Windows Runtime	런타임 라이브러리를 호출하는 어플리케이션 콜
	Windows .NET Runtime	.NET 프레임워크를 호출하는 어플리케이션 콜
	Windows RT Runtime	WinRT를 호출하는 어플리케이션 콜
프로세스	OS Process	윈도우 프로세스
	Thread	스레드 (윈도우 프로세스의 실행 일부)
	Kernel Thread	스레드 (윈도우 커널 프로세스의 실행 일부)
	Native Application	Win32, Win64 애플리케이션
	Managed Application	.NET 웹 어플리케이션
	Thick Client	서버에 의지하지 않고 스스로 다량의 데이터를 처리하는 클라이언트
	Browser Client	브라우저를 활용하여 데이터를 처리하는 클라이언트
	Browser and Active Plugins	브라우저 플러그 인
	Web Server	웹서버 관련 프로세스
	Windows Store Process	윈도우 스토어 관련 프로세스
	Win32 Service	네트워크 관련 프로세스 또는 서비스
	Web Application	일반 사용자에게 웹 콘텐츠를 제공하는 어플리케이션
	Web Service	웹을 통해 사용자 인터페이스를 제공하는 서비스
	Virtual Machine	Hyper-V 기반 가상 머신
	Applications Running on a non Microsoft OS	구글이나 애플 기반 운영체제에서 동작 중인 마이크로소프트 어플리케이션

데이터 저장소	Cloud Storage	클라우드 기반의 데이터 저장소
	SQL Database	SQL 데이터베이스
	Non Relational Database	NoSQL 데이터베이스
	File System	파일시스템
	Registry Hive	레지스트리 하이브
	Configuration File	설정 파일
	Cache	캐시
	HTML5 Local Storage	HTML5 로컬 스토리지
	Cookies	쿠키
	Devices	일반 기기
데이터 흐름	HTTP	HTTP 통신을 활용한 데이터 흐름
	HTTPS	HTTPS 통신을 활용한 데이터 흐름
	Binary	바이너리
	IPsec	IPsec 프로토콜을 활용한 데이터 흐름
	Named Pipe	명명된 파이프 통신을 활용한 데이터 흐름
	SMB	SMBv2 프로토콜을 활용한 데이터 흐름
	RPC or DCOM	원격 프로시저나 분산 컴포넌트 오브젝트 모델을 활용한 데이터 흐름
	ALPC	고급 로컬 프로시저를 활용한 데이터 흐름
	UDP	UDP 통신 프로토콜을 활용한 데이터 흐름
	IOCTL Interface	ioctl을 활용한 데이터 흐름
신뢰 영역	Internet Boundary	인터넷 망 내 신뢰 경계
	Machine Trust Boundary	컴퓨터 내 신뢰 경계
	User or Kernel mode Boundary	사용자 / 커널 간 신뢰 경계
	Appcontainer Boundary	앱 컨테이너 간 신뢰 경계
	CorpNet Trust Boundary	CorpNet 신뢰 경계
	Sandbox Trust Boundary	샌드박스로 인한 신뢰 경계
	Internet Explorer Boundary	인터넷 익스플로러 기반 신뢰 경계
	Othre Browsers Boundary	기타 브라우저 간 신뢰 경계

[표 3] 데이터 흐름도 구성요소

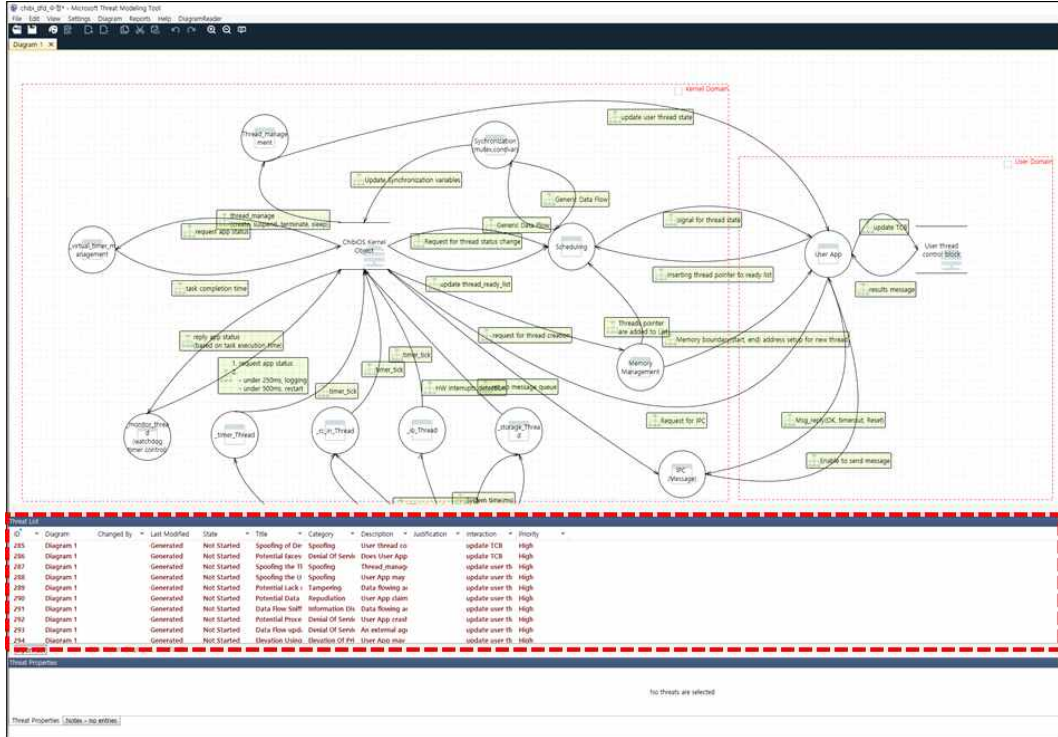
이를 기반으로 아래와 같이 데이터 흐름도를 작성할 수 있다.



[그림 10] 데이터 흐름도 예시

2.1.2.2. 위협 분석

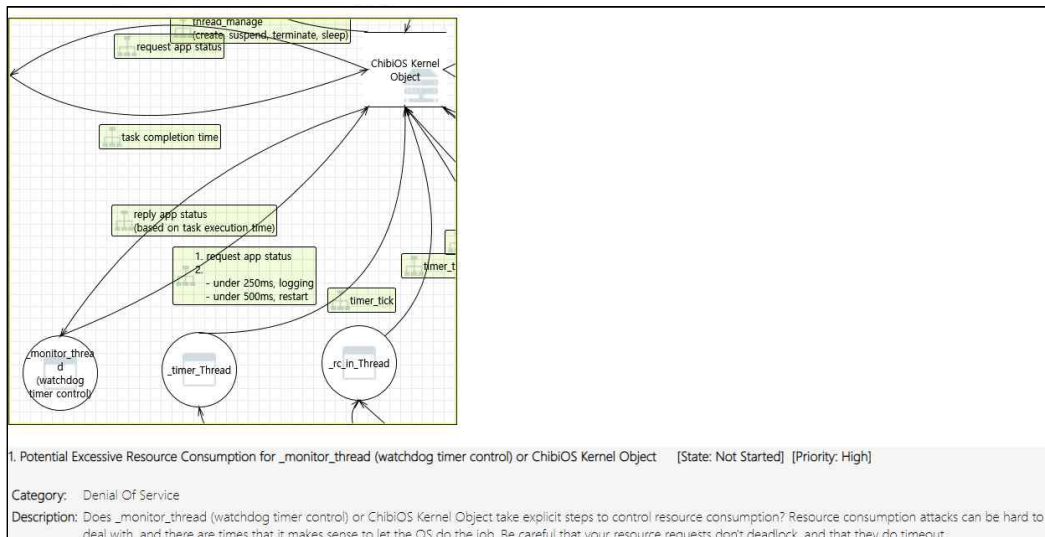
MS TMT는 STRIDE 방법론을 기반으로 분석 대상에서 발생할 수 있는 위협 목록을 아래 그림과 같이 제공한다.



[그림 11] 위협 분석 예시

2.1.2.3. 위협 분석 보고서 생성

MS TMT는 도출된 위협 목록을 기반으로 아래 그림과 같이 세부 구성 요소 별 발생할 수 있는 위협 정보와 이에 대한 상세 정보를 제공한다.



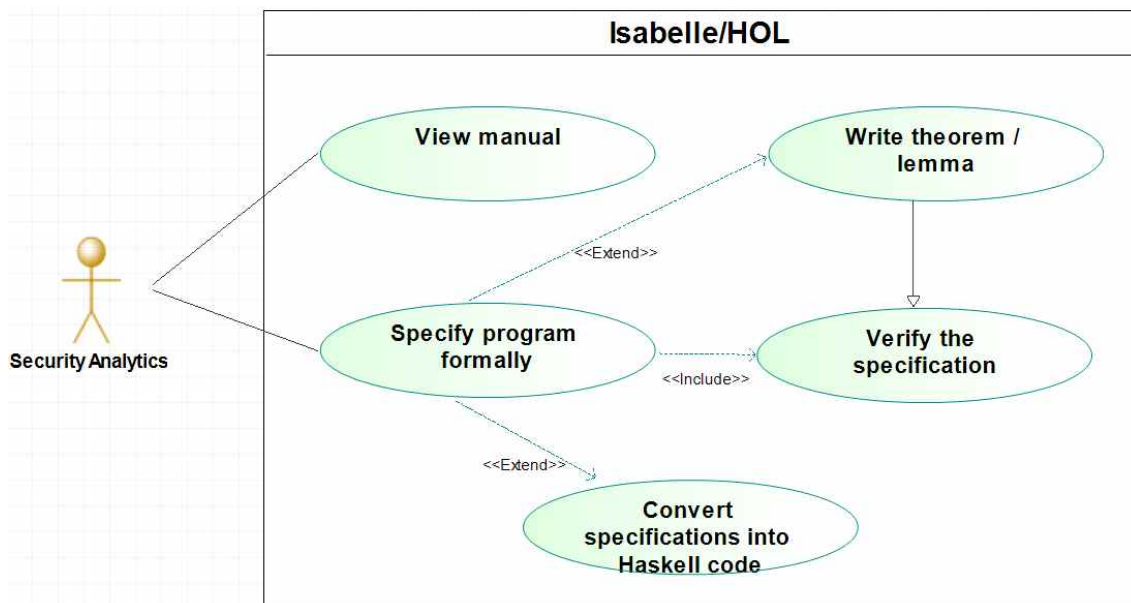
[그림 12] 위협 분석 보고서 생성 예시

3. 정형기법 도구

본 장에서는 요구사항 분석 단계/설계 단계에서 정형 명세 및 검증을 목적으로 사용되는 도구에 대해 서술한다. 정형 기법은 개발 완료된 소프트웨어가 요구사항 간 모순이나 설계 내 의도치 않은 보안 결함이 발생하는 것을 방지하기 위해 활용될 수 있다. 정형 기법은 정형 명세와 정형 검증 두 가지로 구분된다. 정형 명세는 수학과 논리학에 기반을 둔 방법으로 하드웨어나 소프트웨어(프로그램)를 수학적 기호를 사용하여 모델링하는 것을 의미한다. 정형 검증은 모델링된 하드웨어나 소프트웨어가 정형하게 기술된 특정 성질을 만족하는지 여부를 수학적으로 검증하는 것을 의미한다. 따라서 정형기법을 활용할 경우, 자연어로 인해 작성된 요구사항 및 설계가 내포하는 애매모호함이나 불확실성을 최대한 줄일 수 있다.

3.1. Isabelle/HOL

본 절에서는 공통평가기준에서 정형 명세 및 검증 수행 시 활용이 권고되는 도구인 Isabelle/HOL에 대해 서술한다. Isabelle은 정형 언어에 표현된 수학적 공식을 허용하는 환경을 증명하는 대중적인 정리(theorem)로, 이러한 공식을 논리 연산으로 증명하기 위한 증명기이다. Isabelle은 1 차 논리, 고차원 논리 (HOL: High Order Logic)을 바탕으로 프로그램을 명세한다. 이때 Isabelle/HOL은 명세된 프로그램을 프로그래밍 언어와 정형기법의 중간 형태인 함수형 언어인 Haskell 또는 Spark로 변환하는 기능을 제공한다. 명세된 프로그램은 Isabelle/HOL은 커맨드라인 기반 사용자 인터페이스를 활용하거나, Sledgehammer 기반 자동화된 정형 검증 기능을 통해 정형 검증된다. 해당 도구에 대한 유즈케이스 다이어그램은 아래와 같습니다.

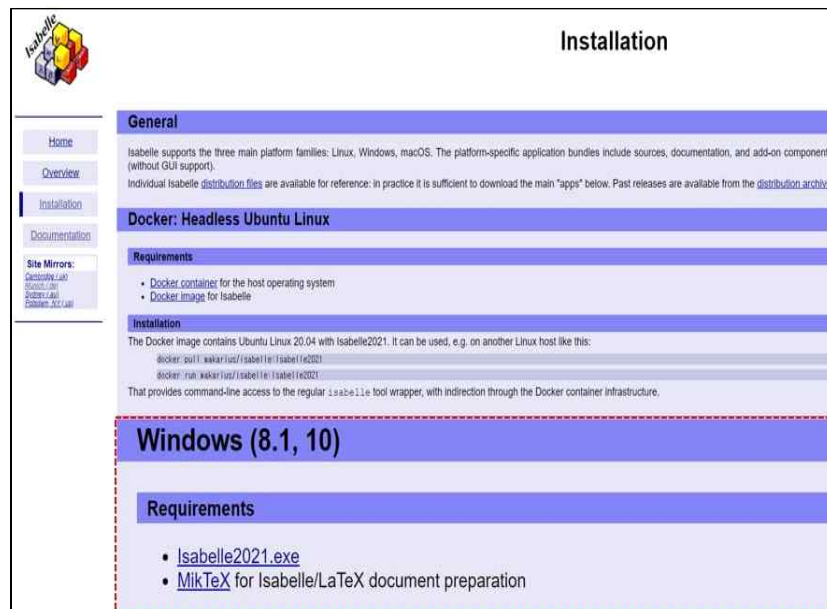


[그림 13] Isabelle/HOL 유즈케이스 다이어그램

3.1.1. 설치

Isabelle/HOL은 아래 URL을 통해 다운로드할 수 있다. 해당 링크에 접속할 경우, 화면 하단에서 다운로드 아이콘을 확인할 수 있다. 해당 아이콘을 클릭하면, PC에 Isabelle/HOL이 자동으로 설치되는 것을 확인할 수 있다. 이때 설치되는 Isabelle/HOL의 버전은 아래 그림에서 확인할 수 있듯이 Isabelle2021이다.

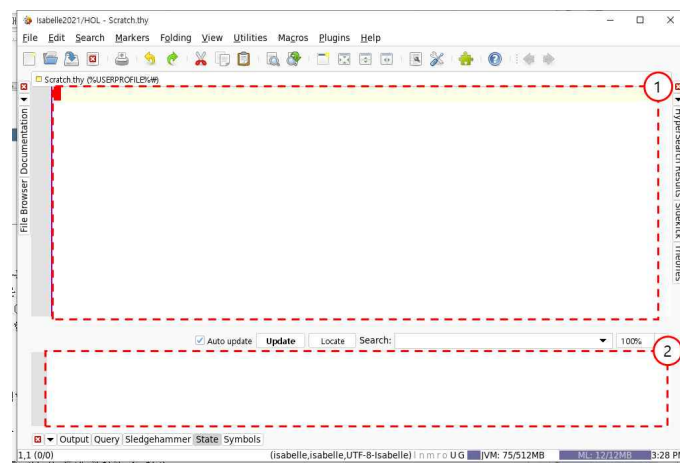
■ URL: <https://isabelle.in.tum.de/installation.html>



[그림 14] Isabelle/HOL 다운로드 화면

3.1.2. 실행

설치된 Isabelle/HOL을 더블 클릭하여 실행하면 아래 그림과 같은 화면이 출력된다. ①은 프로그램에 대한 명세를 작성하는 부분이고, ②에서는 ①에서 명세된 프로그램에 대한 검증 결과를 확인할 수 있다.



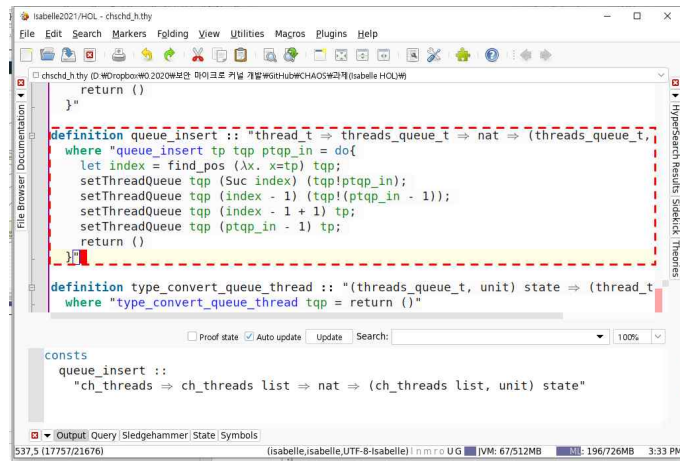
[그림 15] Isabelle2021 실행화면

3.1.2.1. 정형 명세

Isabelle/HOL을 활용하여 정형 명세를 위한 수행하기 위한 기본 문법은 아래 URL을 통해 확인할 수 있다. 해당 URL을 통해 다운로드할 수 있는 “What’s in Main” 문서는 Isabelle/HOL에서 제공하는 기본 함수들에 대한 정보를 제공한다.

상기 문서에서 제시하는 함수는 입력 매개변수나 그 기능에 의해 크게 HOL, Orderings, Set, List, Algebra, Nat, Int 등 총 24개로 구분된다. 해당 함수들을 활용하면 명세하고자 하는 하드웨어나 프로그램에 대한 정형 명세를 수행할 수 있다. 아래 그림은 TOE 내부 자료구조 중 하나인 우선순위 큐에 데이터를 삽입하는 기능에 대해 정형 명세를 수행한 결과이다.

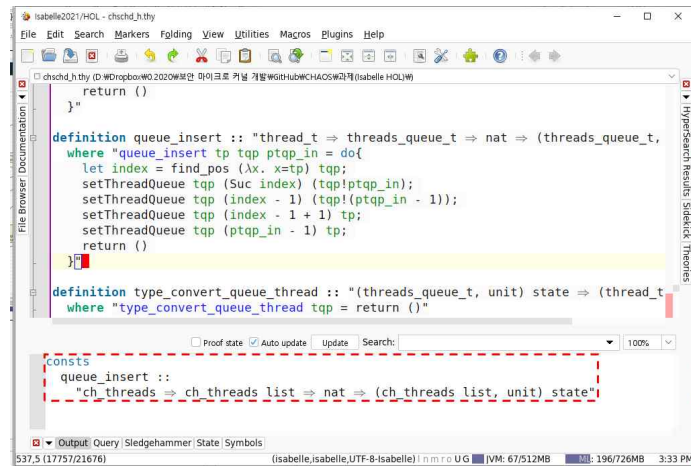
■ URL: <https://isabelle.in.tum.de/dist/Isabelle2021/doc/main.pdf>



[그림 16] 정형 명세 예시

3.1.2.2. 정형 검증

정형 검증의 경우, 정형 명세에 키보드 커서를 위치시키면 자동으로 검증 결과를 아래 그림과 같이 출력한다.

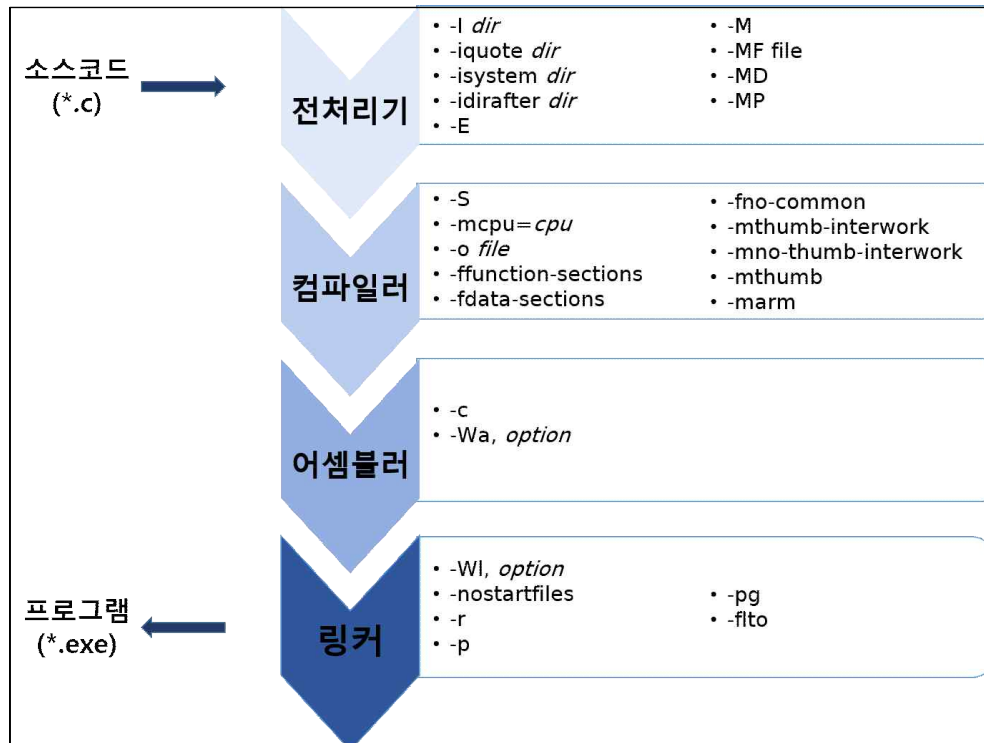


[그림 17] 정형 검증 결과 예시

4. 개발 도구

본 장에서는 구현 단계에서 사용한 프로그래밍 언어와 컴파일러에 대해 서술한다. 본 문서에서 제공하는 프로그래밍 언어와 컴파일러 관련 정보는 향후 구현의 표현(ADV_IMP) 세부 평가활동 수행시, 평가자 TOE의 소스코드를 명확하게 이해하기 위해서 필요하다.

구현 시, 소스코드는 아래 그림과 같이 컴파일러를 포함한 전처리기, 어셈블러, 링커를 거쳐 바이너리 형태의 실제 프로그램으로 변환된다. 이때 소스코드는 Makefile 내 작성된 다양한 컴파일 옵션을 참고하여 변환되기 때문에, 이에 대한 정보를 아래 절에 서술한다.



[그림 18] Gcc 컴파일러를 통한 c언어 기반 소스코드 변환

4.1. C 언어

C는 1972년 켄 톰슨과 데니스 리치가 벨 연구소에서 일할 당시 새로 개발된 유닉스 운영 체제에서 사용하기 위해 개발한 프로그래밍 언어이다. 1980년대에 들어서면서 다양한 C 언어들이 독자적으로 지원하는 확장 기술은 서로 호환되지 않게 되었다. 또한 C 언어가 미국 정부 프로젝트를 위해 사용되기 시작하면서 미국의 표준화 기구인 ANSI에 의해 1980년대 초반에 표준화를 시작되게 된다. 이후 1989년도에 탄생한 C언어 표준이 지금까지 C언어 서적에서 인용되고 있는 ANSI C(정확히는 ANSI X3.159-1989)이다. 발표된 연도를 이용해서 줄여서 C89라고 불리기도 한다. ANSI의 표준화 이후 C 언어 표준이 상대적으로 정적으로 남아 있었던 동안, C++는 표준화를 위하여 계속 진화하고 있었다. 1995년에 1990년의 C 표준에 대한 규약 수정안 1이 출판되었는데, 이는 약간의 세부 사항을 교정하고 국제적 문자 세트에 대한 보다 확장된 지원을 위한 것이었다. C 표준은 1990년대 후반에 더 개정되어, 1999년 ISO/IEC 9899:1999가 출간되었다.

4.2. Gcc 컴파일러

본 절에서는 TOE에 대한 바이너리 형태의 프로그램을 생성하는데 사용한 Gcc 컴파일러를 설명한다. GNU 컴파일러 컬렉션(Gcc)은 다양한 프로그래밍 언어, 하드웨어 아키텍처 및 운영 체제를 지원하는 GNU 프로젝트가 제작한 최적화 컴파일러로 Linux 커널과 관련된 대부분의 프로젝트의 표준 컴파일러이다. Gcc 컴파일러는 TOE의 소스코드를 작성하는데 사용된 프로그래밍 언어인 C/C++ 뿐만 아니라 Objective-C, Objective-C ++, Fortran, Java, Ada, D 및 Go 언어도 컴파일할 수 있으며 전처리기, 컴파일러, 어셈블러, 링커에서 사용될 수 있는 옵션을 모두 일괄 입력받을 수 있다. 현재 Gcc 컴파일러는 GNU 운영 체제의 공식 컴파일러일 뿐만 아니라 Linux를 포함한 많은 컴퓨터 운영 체제의 표준 컴파일러로 채택되었다.

4.2.1. 컴파일 옵션 설명

TOE에 대한 바이너리 형태의 프로그램을 생성하기 위해 사용한 옵션의 세부 내용은 아래와 같다.

옵션	설명
-Wa, <i>option</i>	■ -Wa, <i>option</i> 은 어셈블러에 <i>option</i> 을 옵션값으로 전달한다. <i>option</i> 이 여러 옵션들로 구성되어 있는 경우에는 각 옵션이 쉼표 단위로 구분된다.
-Wl, <i>option</i>	■ -Wl, <i>option</i> 은 링커에 <i>option</i> 을 옵션값으로 전달한다. <i>option</i> 이 여러 옵션들로 구성되어 있는 경우에는 각 옵션이 쉼표 단위로 구분된다. 이 문법으로 특정 인자를 링커에 전달할 수 있다. 예를 들어, -Wl,-Map,output.map는 -Map output.map를 링커에 전달한다.
-nostartfiles	■ -nostartfiles는 링크 과정에서 표준 시스템 스타트업 파일 (standard system startup file)을 사용하지 않도록 한다. -nostdlib, -nolibc, -nodefaultlibs가 사용되지 않는 경우에 표준 시스템 라이브러리(standard system library)가 사용된다.
-mcpu= <i>cpu</i>	■ -mcpu는 어떤 <i>cpu</i> 를 기준으로 컴파일할 것인지를 설정한다. 구체적으로 이 옵션을 통해 아키텍처 유형, 레지스터 사용량, 그리고 명령어 스케줄링 매개변수(instruction scheduling parameter)를 설정할 수 있다.
-o <i>file</i>	■ -o <i>file</i> 은 컴파일 결과에 대한 이름을 지정하는 옵션이다. 이 옵션은 컴파일 결과에 대한 기본 출력을 파일명 <i>file</i> 로 저장하도록 한다. 만약 -o 옵션이 지정되지 않는다면 출력 결과는 정해진 규칙을 따라 이름이 부여된다. 출력되는 실행 파일의 기본값은 a.out이다. source.suffix에 대한 목적 파일의 기본값은 source.o이다. 어셈블러 파일의 기본값은 source.s이다. 미리 컴파일된 헤더 파일의 기본값은 source.suffix.gch이다.
-p -pg	■ 분석 프로그램 prof (-p 옵션)이나 gprof (-pg 옵션)에 적합한 프로파일 정보를 작성하기 위한 추가적인 코드를 생성한다.

-c	<ul style="list-style-type: none"> ■ -c는 소스 파일들에 대한 목적 파일(object file)을 생성한다. 이 옵션이 있는 경우 입력으로 주어진 소스 파일은 링커에 의해 처리되지 않는다. 즉 최종 출력은 각 소스 파일에 대한 목적 파일의 형태이다.
-I <i>dir</i> -include <i>dir</i> -isystem <i>dir</i> -idirafter <i>dir</i>	<ul style="list-style-type: none"> ■ -I <i>dir</i>, -include <i>dir</i>, -isystem <i>dir</i>, -idirafter <i>dir</i>는 경로 <i>dir</i>를 소스 코드에서 사용하는 헤더 파일들의 검색 경로에 추가한다. 즉, 이 옵션은 새로운 헤더 파일을 사용하는 경우, 해당 헤더 파일의 경로를 컴파일러에게 알려줄 때 사용한다. 구체적으로는 전처리 과정에서 이 옵션들로 주어진 경로 <i>dir</i>을 포함하여 헤더 파일들을 검색하게 된다. ■ -include로 지정된 경로는 #include "file" 형식의 include 지시자(directive)만 적용된다. -I, -isystem, 또는 idirafter로 지정된 경로는 #include "file" 과 #include <file> 형식에 include 지시자 모두에 적용된다. ■ 여러 경로에 위치할 수 있는 헤더 파일들을 검색 순서를 지정하기 위하여 -I <i>dir</i>, -include <i>dir</i>, -isystem <i>dir</i>, -idirafter <i>dir</i>의 옵션을 통해 검색 순서를 명시할 수 있다. <ul style="list-style-type: none"> ■ -include 옵션으로 지정된 경로들이 왼쪽에서 오른쪽 순서로 검색된다. ■ -I 옵션으로 지정된 경로들이 왼쪽에서 오른쪽 순서로 검색된다. ■ -isystem 옵션으로 지정된 경로들이 왼쪽에서 오른쪽 순서로 탐색 된다. ■ -idirafter 옵션으로 지정된 경로들이 왼쪽에서 오른쪽 순서로 검색된다.
-S	<ul style="list-style-type: none"> ■ -S는 컴파일까지만 처리하고, 나머지는 수행하지 않는 옵션이다. 이 옵션이 있는 경우 어셈블 단계를 거치지 않게 된다.
-E	<ul style="list-style-type: none"> ■ -E는 전처리 과정의 결과를 화면에 보여주도록 하는 옵션이다. 이 옵션이 있는 경우 컴파일 과정으로 넘어가기 전에 전처리 단계가 끝나는 즉시 종료된다. 해당 옵션을 통해 전처리 결과는 표준 출력으로 사용자에게 제공되며, 출력 값은 전처리된 소스 코드의 형태로 존재한다.
-r	<ul style="list-style-type: none"> ■ -r은 나중에 추가적인 링크가 가능한 목적 파일을 출력으로 생성한다.
-ffunction-sections -fdata-sections	<ul style="list-style-type: none"> ■ -ffunction-sections과 -fdata-sections은 gcc의 최적화 옵션에 해당한다. 이 옵션들은 컴파일러에게 함수(-ffunction-sections)나 코드 데이터(-fdata-sections)를 해당 섹션에 위치시키도록 한다. 이 옵션을 사용하지 않는 경우에는 임의의 섹션에 코드나 데이터가 들어갈 수 있지만, 이 옵션들을 사용하게 되면 해당 섹션에 강제로 위치시킬 수 있다. 즉, 이 옵션을 통해 사용하지 않는 함수나 데이터가 제거할 수 있다.

-fno-common	■ -fno-common은 컴파일러가 초기화되지 않은 전역 변수들을 목적 파일의 BSS 영역에 위치시키도록 한다.
-flto	■ -flto는 표준 링크 타임(standard link-time)을 최적화 한다.
-mthumb-interwork	■ -mthumb-interwork는 ARM을 위한 옵션 중에 하나로, ARM과 Thumb 명령어 세트 사이의 호출을 지원하는 코드를 생성한다.
-mno-thumb-interwork	■ -mno-thumb-interwork는 -mthumb-interwork 옵션을 종료한다. 즉, ARM과 Thumb 명령어 세트 사이의 호출을 지원하는 코드를 생성하지 않는다.
-mthumb -marm	■ -mthumb 또는 -marm은 ARM 상태에서 실행되는 코드를 생성하는 것(-marm)과 Thumb 상태에서 실행되는 코드를 생성하는 것(-mthumb) 중에 하나를 선택한다. 기본 값은 ARM 상태에서 실행되는 코드를 생성하는 것이지만, --with-mode=state 옵션을 통해 다른 상태로 변경할 수 있다.
-M	■ -M 옵션은 의존 관계(dependency)를 설명하는 파일을 자동으로 생성하는 옵션이다. 이 -M 옵션은 암묵적으로 -E 옵션을 포함하여, 컴파일 단계 이전에 전처리 과정에서 중단되도록 한다.
-MF <i>file</i>	■ -MF <i>file</i> 은 -MD 옵션과 함께 사용되면, 기본 의존성 파일(default dependency output file)을 <i>file</i> 로 오버라이딩한다.
-MD	■ -MD는 -E 옵션의 기능이 포함되지 않는 점을 제외하고는 -M -MF <i>file</i> 과 동일하다. 의존 관계에 대한 정보는 출력 파일명의 접미사 '.o'를 '.d'로 옮겨놓은 파일명의 파일로 출력된다. -E의 기능이 포함되지 않기에 컴파일 과정을 거치게 되는데, -MD 옵션은 컴파일 과정의 부수 효과(side effect)로서의 의존성 파일(dependency output file)을 생성하는데 사용될 수 있다.
-MP	■ -MP 옵션은 CPP가 기본 파일 이외에 각 의존 관계에 대한 가짜 대상(phony target)을 추가하여 각 항목이 아무것도 의존하지 않도록 지시한다. 만약에 소스파일에 #include 라인을 제거하고 해당 헤더 파일을 삭제했다면, 의존 관계 출력 파일인 .d 파일의 목록에는 해당 파일이 남아있기 때문에 다음 make 명령 실행시에는 의존 파일이 없다는 오류가 발생할 수 있다. 이 때 -MP 옵션을 사용하게 되면, 헤더 파일의 더미 규칙(dummy rule)을 생성하여 make 실행이 종료되지 않게 한다.

5. 테스트 도구

본 장에서는 검증 단계에서 활용할 수 있는 테스트 도구에 대해 서술한다. 테스트 도구는 테스트 과정에서 소모되는 비용(시간과 인력)을 절약하기 위해 활용한다. 테스트 도구들은 프로그램 실행 유무에 따라 크게 정적, 동적 테스트 도구로 구분된다.

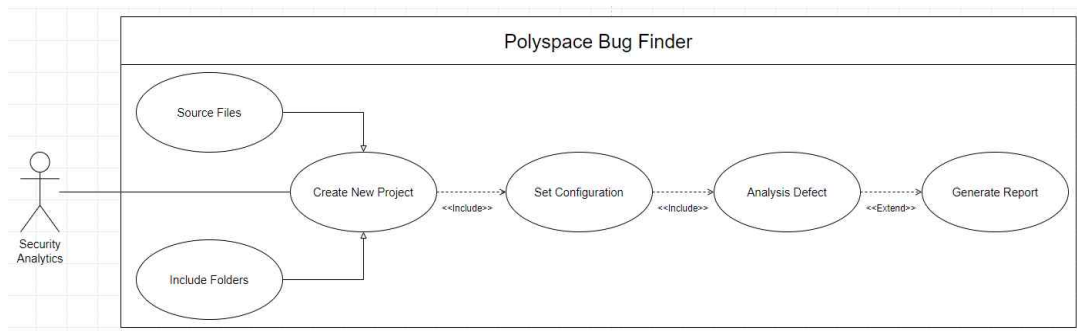
정적 테스트는 실제 프로그램을 실행하지 않고 소스코드만을 활용하여 소프트웨어를 분석하는 방법을 의미한다. 일반적으로 소스코드를 정적 테스트 도구에 입력하면 해당 도구는 소스코드 동작을 추상화하는 모델링을 수행한다. 모델링 수행 결과 도출된 소스코드에 대한 모델은 소스코드 내 선언된 변수의 범위나 실행 경로를 추적할 수 있는 다양한 정보들이 포함된다. 정적 테스트의 경우 해당 모델에 대한 분석을 수행함으로써 그 중 취약성을 유발할 수 있는 보안 약점을 식별할 수 있다.

동적 테스트는 실제 프로그램을 실행하면서 소프트웨어를 분석하는 것을 의미한다. 해당 기법은 프로그램 실행 과정 중 발생할 수 있는 다양한 입/출력 데이터의 변화 및 사용자 상호 작용에 따른 보안 약점을 식별한다. 동적 테스트의 수행 목적은 정적 테스트를 통해 확인할 수 없는 보안 약점을 보완하기 위함이다.

TOE를 개발하기 위해 본 연구진은 정적 테스트와 동적 테스트 모두 수행하였으며, 이를 위해 ▲Polyspace Bug Finder, ▲ American Fuzzy Lop(이하 AFL)을 개발도구로 선정하였다.

5.1. Polyspace Bug Finder

Polyspace Bug Finder는 미 국립표준기술연구소에서 활용을 권고하는 정적 분석 도구 중 하나로 C/C++ 임베디드 소프트웨어에서 발생할 수 있는 런타임 오류, 동시성 등의 보안 결함들을 탐지한다. 해당 도구는 정적 분석을 수행하기 위해 아래 그림과 같이 프로젝트를 생성하고 분석 대상에 대한 세부 정보를 분석 전에 설정한다. 이후 소프트웨어 제어, 데이터 흐름, 실행 기능 간 관계 등을 분석한 결과를 바탕으로 소프트웨어 결함을 식별한 후, 보고서를 자동 생성하여 사용자에게 제공한다.



[그림 19] Polyspace Bug Finder 유즈케이스 다이어그램

5.1.1. 설치

Polyspace Bug Finder는 아래 URL에서 다운로드 링크를 제공한다. 해당 도구는 상용 도구이기 때문에 다운로드 후 설치 시, 라이선스가 요구된다.

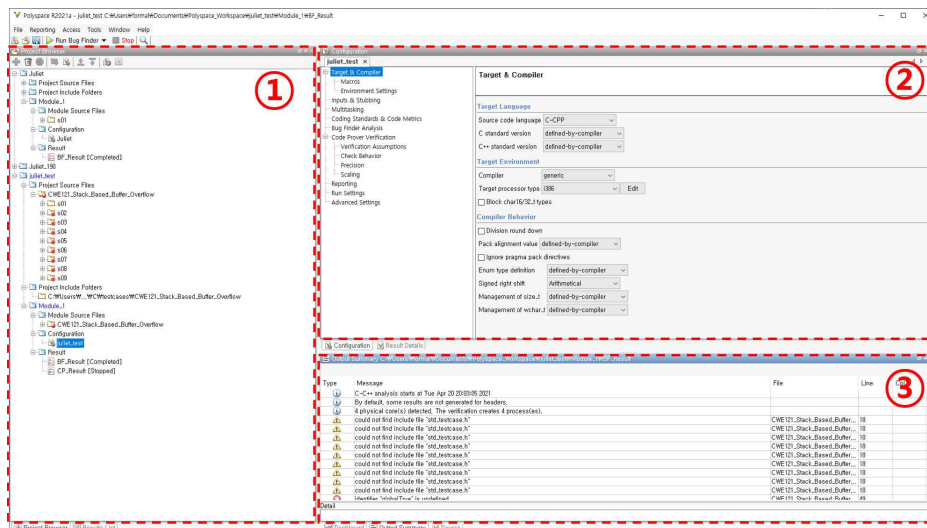
- URL: <https://kr.mathworks.com/products/polyspace-bug-finder.html>



[그림 20] Polyspace Bug Finder 다운로드 화면

5.1.2. 실행

Polyspace Bug Finder 실행 후 새로운 프로젝트를 생성하면 아래 그림과 같은 화면이 출력된다.

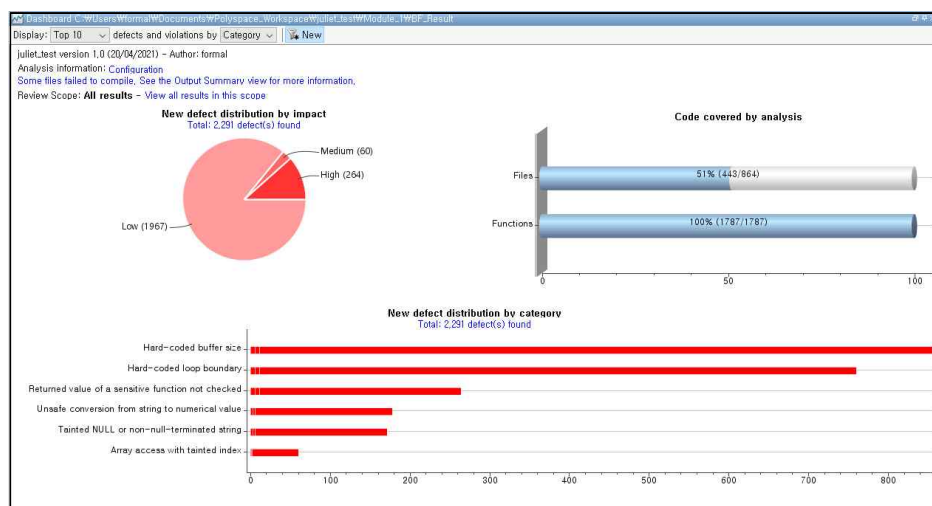


[그림 21] Polyspace 실행 화면

Polyspace Bug Finder에 새로운 프로젝트를 생성하면 ①과 같이 Project Browser 창에 정적 분석을 원하는 시스템의 소스 파일을 추가할 수 있다. 그리고 정적 분석을 수행하기 위해 필요한 정보는 ②와 같은 Configuration 창을 통해 ▲개발 언어, ▲컴파일러 정보, ▲프로세서 정보와 같은 분석 대상의 세부 정보를 설정하고 분석 옵션(코딩 표준, 결함 종류)을 설정할 수 있다. 분석이 시작하면 ③에 위치한 창을 통해 분석 과정 중 발생하는 예러나 경고 메시지를 확인할 수 있다.

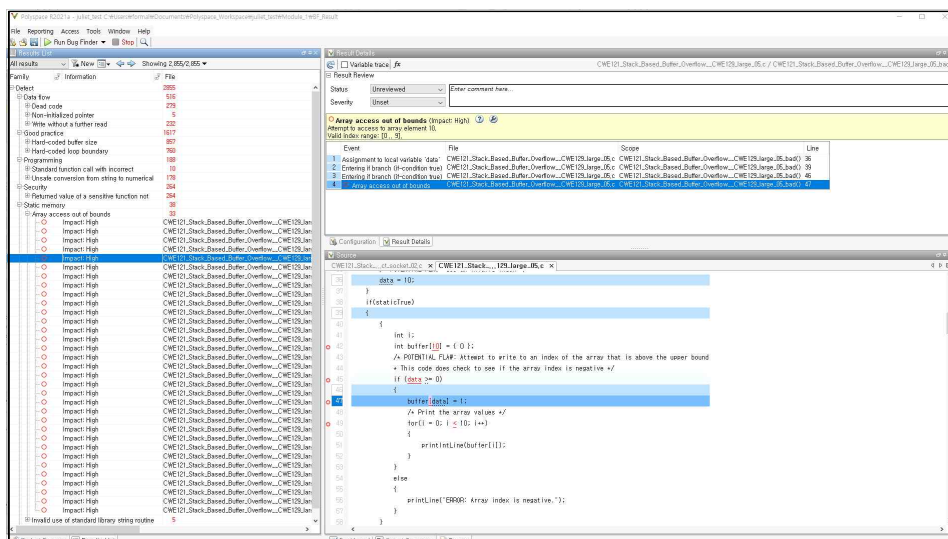
5.1.2.1. 정적 분석

Polyspace Bug Finder는 다양한 종류의 버그를 탐지하고 요약한 결과를 아래 그림과 같은 대쉬보드 형태로 제공한다.



[그림 22] 분석 결과에 대한 대쉬보드

그리고 탐지한 버그에 대한 이유와 경로, 위치를 세부 결과에서 확인이 가능하다.



[그림 23] 정적 분석 세부 결과

5.1.2.2. 정적 분석 보고서 생성

Poyspace Bug Finder는 아래 그림과 같이 탐지한 버그에 대한 목록과 작성한 리뷰에 대해 자동으로 보고서를 생성하여 제공한다.

Table 1.1. Project Summary

	Count	Reviewed	Unreviewed	Pass/Fail
Defects	2855	0	2855	NA
Total	2855	0	2855	NA

Table 1.2. Summary By File

File	Defects (Reviewed)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_01.c	15 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_02.c	24 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_03.c	24 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_04.c	24 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_05.c	24 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_06.c	24 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_07.c	24 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_08.c	24 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_09.c	0 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_10.c	0 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_11.c	20 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_12.c	25 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_13.c	0 (0)
C:\Users\formal\Desktop\juliet_test\Ctestcases\CWE121_Stack_Based_Buffer_Overflows\01\CWE121_Stack_Based_Buffer_Overflow_CWE129_connect_socket_14.c	0 (0)

1

[그림 24] Polyspace Bug Finder 보고서

5.1.3. 적용된 구현 표준

Polyspace Bug Finder는 C/C++ 임베디드 소프트웨어에서 발생 가능한 런타임 오류, 동시성 문제, 보안 취약성 등 결함들을 탐지한다. 정적 분석을 활용하여 소프트웨어의 제어, 데이터 흐름 및 프로시저 간 동작을 분석하고 MISRA C/C++, JSF++, Cert C/C++ 같은 다양한 표준에 따라 구현되었는지 확인한다.

MISRA C/C++은 차량용 소프트웨어의 안전성을 높이기 위해 출판되었고 MISRA(Motor Industry Software Reliability Association)에서 개발된 코딩 가이드라인이다. 현재는 차량용 소프트웨어뿐만 아니라 우주/항공, 의료장비, 국방, 철도 등의 다양한 분야에서 적용되고 있다. 2004년에 출판된 MISRA-C:2004는 개발에 적합하지 않은 규칙들이 제거되고 여러 의미를 가질 수 있는 규칙이 정확하게 하나의 의미를 가질 수 있도록 분리되었다. 특히 2016년에 출판된 MISRA-C:2012는 소프트웨어 보안성을 위해 몇 가지 새로운 규칙들이 추가되었고 이를 준수함으로써 코딩 오류 및 보안 결함을 방지하는 것이 가능하게 되었다.

JSF++는 항공 소프트웨어가 준수해야 하는 C++ 코딩 표준으로써 2005년에 JSF사에서 개발되었다. 해당 표준은 안정성을 최우선적으로 고려하고 코딩 스타일과 코드 메트릭에 대한 가이드라인도 제시한다.

Cert C/C++ 코딩 표준은 Carnegie Mellon University에서 운영하는 SEI(Software Engineering Institute) 협회에서 제안하였고 시스템의 안전성, 신뢰성, 보안성을 목표로 가진 코딩 표준이다. 해당 표준은 SEI 및 그 밖의 협력 기관에서 지원하고 있고 GIAC와 GSSP-C에서 진행되는 시험이나 인증 과정의 기초로 사용되고 공식적인 검사 기술을 통해 해당 표준의 규칙 준수 여부를 확인할 수 있다.

5.2. AFL Fuzzer

AFL Fuzzer는 프로그램 내 조건문에 의해 발생하는 분기점을 기준으로 코드 커버리지를 측정하며, 테스트 커버리지를 극대화하기 위해 유전 알고리즘을 활용하여 입력 값을 변조하는 방식으로 동작하는 퍼저이다.

5.1.1. 설치

AFL Fuzzer를 설치하는 방법과 대상 프로그램을 AFL Fuzzer가 작동할 수 있도록 빌드하는 방법은 다음과 같다. 먼저 AFL Fuzzer, preeny, ChibiOS가 필요하며 의존 라이브러리는 총 5개로 build에 필요한 라이브러리 묶음인 gcc-multilib g++multilib와 preeny 빌드에 필요한 libini-config인 libini-config-dev:i386, seccomp:i386, libseccomp-dev:i386, libtool로 나뉜다.

```
# 1. build에 필요한 라이브러리 묶음
$ sudo apt install -y build-essential gcc-multilib g++-multilib

# 2. preeny 빌드에 필요한 libini-config
$ sudo apt install -y libini-config-dev libini-config-dev:i386 seccomp:i386 libseccomp-dev:i386 libtool
```

[그림 25] 의존 라이브러리 설치

AFL Fuzzer 설치는 크게 5가지 과정으로 나뉘며 AFL 다운로드, 압축해제, AFL 폴더로 이동, AFL 빌드, AFL 설치로 이루어지며 각 과정은 아래 그림과 같다.

```
# 1. AFL 다운로드
$ wget http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

# 2. 압축해제
$ tar -xzf afl-latest.tgz

# 3. afl 폴더로 이동
$ cd afl-2.52b # 또는 새로운 버전

# 4. afl 빌드
$ make

# 5. afl 설치
# 제대로 설치되었으면 afl-fuzz 라는 명령어를 터미널에서 사용할 수 있게됨
$ sudo make install
```

[그림 26] AFL Fuzzer 설치

AFL 퍼저는 Standard Input/Output (Std I/O) 채널을 통해 분석 대상에게 명령어를 전송한다. 하지만 TOE는 전용 에뮬레이터를 통해서만 동적 테스트를 수행할 수 있어 선행 작업이 필요하다. 해당 에뮬레이터의 경우, AFL 퍼저와 달리 TCP 소켓을 통해 명령어를 전송하기 때문에 Std I/O를 TCP 소켓 통신으로 연결해 주기 위해 preeny 라이브러리를 설치한다. 해당 라이브러리는 desock.so 파일을 AFL에게 제공하여 Socket API를 후킹할 수 있다. 이를 통해 Std I/O로 송신된 퍼징 관련 명령어는 TCP Socket을 통해 오류 없이 ChibiOS로 수신된다. preeny를 빌드하는 방법은 아래 그림과 같다. 이때 퍼징을 수행할 타겟의 아키텍처에 따라 ARCH 값을 별도로 수정해야 한다.


```
# 1. preeny 다운로드
$ git clone https://github.com/zardus/preeny.git

# 2. src/desock.c 에서
# __attribute__((destructor)) void preeny_desock_shutdown()
# 부분 삭제 (또는 주석처리)

# 3. 빌드
$ make ARCH=i386
```

[그림 27] preeny 설치

빌드가 정상적으로 수행되면 preeny/i386-linux-gnu 폴더가 아래 그림과 같이 생성된다.

```
taegeun@taegeun-VirtualBox:~/preeny$ ls
cmake-build-multiarch.sh  CMakeLists.txt  LICENSE  README.md  test
cmake-build.sh            i386-linux-gnu  Makefile  src
taegeun@taegeun-VirtualBox:~/preeny$ ls i386-linux-gnu/
crazyrealloc.so  deptrace.so  desock_dup.so  devid.so      logging.so      setstdin.so
dealarm.so       derand.so    desock.so      ensock.so     mallocwatch.so  startstop.so
deexec.so        desigact.so  desrand.so     eofkiller.so  patch.so        writeout.so
defork.so        desleep.so   detime.so      getcanary.so  setcanary.so
```

[그림 28] preeny 빌드 완료

또한 AFL로 퍼징을 수행하기 위해서는 AFL Fuzzer와 호환되는 컴파일러를 활용해야 한다. 따라서 아래 그림처럼 Makefile에서 사용하는 컴파일러를 변경해야 한다.

```
taegeun@taegeun-VirtualBox: ~/ardupilot/modules/ChibiOS/demos/various/RT-Posix-Simu...
File Edit View Search Terminal Help

# List all user libraries here
ULIBS =

#
# End of user defines
#####

#####
# Compiler settings
#
TRGT =
CC = /usr/local/bin/afl-gcc
CXX = /usr/local/bin/afl-g++
# Enable loading with g++ only if you need C++ runtime support.
# NOTE: You can use C++ even without C++ support if you are careful. C++
# runtime support makes code size explode.
LD = $(TRGT)gcc
#LD = $(TRGT)g++
CP = $(TRGT)objcopy
AS = $(TRGT)gcc -x assembler-with-cpp
AR = $(TRGT)ar
OD = $(TRGT)objdump
SZ = $(TRGT)size
HEX = $(CP) -O ihex
BIN = $(CP) -O binary
COV = gcov

# Define C warning options here
CWARN = -Wall -Wextra -Wundef -Wstrict-prototypes

# Define C++ warning options here
```

[그림 29] 대상 컴파일러 변경

5.1.2. 실행

AFL Fuzzer로 퍼징을 수행하기 위해서는 실행 환경인 커널과 관련 라이브러리인 preeny에 대한 설정 값을 아래 그림과 같이 입력해야 한다.

```
# 1. 커널 설정
$ sudo sysctl -w kernel.core_pattern=core

# 2. preeny를 AFL에 반영
$ export AFL_PRELOAD=<path-to-preeny>/i386-linux-gnu/desock.so
```

[그림 30] 사전 설정

AFL 퍼저는 앞서 언급하였듯이 유전 알고리즘으로 변이된 값을 바탕으로 퍼징을 수행하기 때문에 해당 값을 아래 그림과 같이 생성한다.

```
# 1. ChibiOS demo 디렉토리로 이동
$ cd ~/ardupilot/modules/ChibiOS/demos/various/RT-Posix-Simulator/build

# 2. input 폴더 생성
$ mkdir input

# 3. 인풋 파일 생성
$ cd input
$ echo "echo message" > echo_message.txt
$ echo "mem" > mem.txt
$ echo "test oslib" > test_oslib.txt
$ echo "test rt" > test_rt.txt
$ echo "threads" > threads.txt
$ echo "sysitime" > sysitime.txt
$ echo "info" > info.txt
```

[그림 31] Input 파일 생성

이후 아래 그림과 같이 명령어를 입력함으로써 퍼징을 수행한다.

```
# 1. ChibiOS demo 디렉토리로 이동
$ cd ~/ardupilot/modules/ChibiOS/demos/various/RT-Posix-Simulator/build

# 2. 퍼징 시작
$ afl-fuzz -i input -o output -- ./ch
```

[그림 32] 퍼징 시작

5.1.2.1. 동적분석

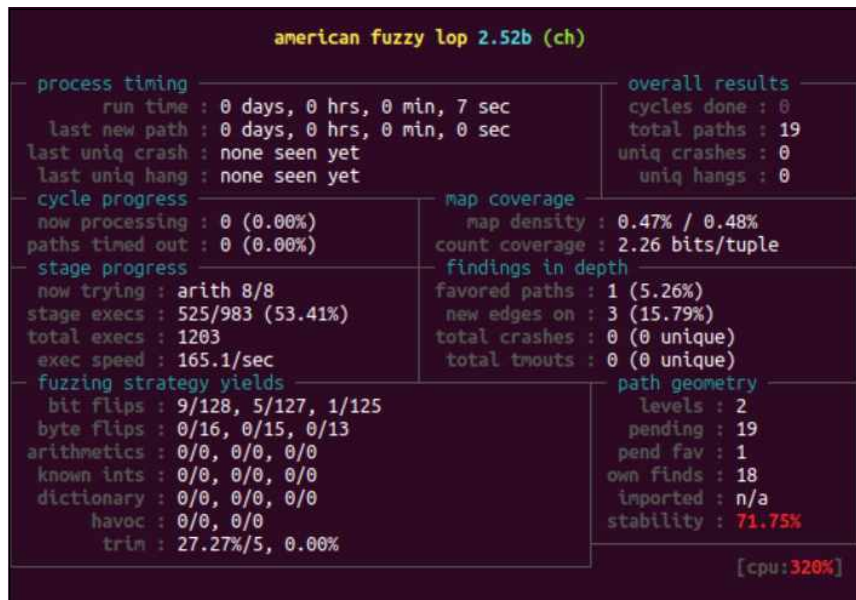
Fuzzing은 기본 입력값(seed)를 바탕으로 이를 변이(mutation)시켜 가며 다양한 인풋을 생성하여 에러 및 실패를 유발하여 보안상의 취약점을 찾아내고자하는 과정으로, Fuzzing 기본 입력 값을 생성해야한다. 분석 대상과 입력 값이 저장된 디렉토리에서 아래 명령어를 입력함으로써 아래 그림과 같이 퍼징을 수행한다.

```
# 1. ChibiOS demo 디렉토리로 이동
$ cd ~/ardupilot/modules/ChibiOS/demos/various/RT-Posix-Simulator/build

# 2. 퍼징 시작
$ afl-fuzz -i input -o output -- ./ch
```

[그림 33] Fuzzing 시작

퍼징이 완료되면 대상 바이너리에 대한 분석결과를 [그림 31]과 같은 상태 진행창에 나타나며, 수행 결과가 기록된 파일은 위 명령을 수행한 디렉토리의 output 폴더에 저장된다. 퍼징 결과에 포함되는 정보로는 아래 그림과 같이 경과 시간, 탐색한 경로(path) 수, 퍼징 싸이클 수, 발견한 크래쉬와 교착상태(hangs), 인풋 변이 기법 사용 통계, 발견한 경로에 대한 정보 등이 있다.



[그림 34] Fuzzing 상태 창