브라우저 취약점 자동화 탐색을 위한 하이브리드 분석 프레임워크 연구 보고서

2025. 12. 21

가천대학교
스마트보안전공 202334843 이동하(팀장)
스마트보안전공 202334828 서예진(팀원)
스마트보안전공 202334860 한보윤(팀원)

<목 차>

제1장. 브라우저 취약점 개요 ····· 4

제1절. 브라우저 개념 ····· 4

제2절. 브라우저 공격 ····· 4

제3절. 브라우저 퍼징 ····· 5

제2장. 브라우저 구성요소 ····· 6

제1절. 샌드박스 ····· 6

제2절. DOM, Renderer ····· 7

제3장. 기존 브라우저 퍼징 연구 분석 ····· 8

제1절. SOK: On the Analysis of Web Browser Security ····· 8

제2절. DOMATO ····· 9

제3절. FreeDom: Engineering a State-of-the-Art DOM Fuzzer ····· 10

제4절. SaGe:Towards Better Semantics Exploration for Browser Fuzz ···· 10

제4장. 신규 방법론 제안 ····· 12

제1절. IDL 기반 문법 분석 ····· 12

제2절. Race Condition Bug 재현 ····· 13

제3절. iframe 기반 속도 개선 ····· 14

제5장. 방법론 검증 및 성능 비교 분석 ····· 15

제1절. IDL 기반 문법 분석 성능 지표 ····· 15

제2절. iframe 기반 실행 속도 비교 ····· 18

제6장. 취약점 분석 도구 개발 ····· 20

제1절. 브라우저 빌드 ····· 20

제2절. 아키텍쳐 설계 ····· 28

제3절. Input Generation ····· 28

제4절. Corpus Execution ····· 29

제5절. Crash Validation ····· 29

제6절. 취약점 식별 ····· 30

제1장. 브라우저 취약점 개요

이번 장에서는 브라우저의 기본 개념과 구조를 정리하고 브라우저 공격이 갖는 위험성과 실제 악용 양상을 개괄하며 브라우저 취약점 발굴 기법인 퍼징의 목적과 접근 방법을 정리한다.

제1절. 브라우저 개념

브라우저는 웹 서버로부터 전달받은 콘텐츠를 해석하고 실행하여 사용자 화면에 출력하는 클라이언트 소프트웨어이다. HTML CSS JavaScript를 처리하는 과정에서 렌더링 엔진과 JavaScript 엔진이 핵심 역할을 수행한다. 또한 이미지 폰트 비디오와 같은 다양한 포맷을 파싱하고 디코딩하며 네트워크 통신과 캐시 인증서 검증 저장소 관리까지 포함하는 복합 실행 환경이다. 브라우저는 외부에서 제공되는 비정형 입력을 지속적으로 처리하므로 입력 처리 경로가 복잡해질수록 결함이 발생할 가능성이 증가한다. 특히 파서 디코더 JIT 컴파일러 가비지 컬렉터와 같이 경계 조건이 많은 구성요소에서 메모리 오류나 논리 오류가 발생하기 쉽다. 브라우저는 동일 출처 정책과 권한 모델을 통해 사이트 간 데이터 접근을 제한하고 프로세스 분리와 샌드박스를 통해 웹 콘텐츠 실행 영역을 격리한다. 취약점은 주로 메모리 안전 결함이나 보안 경계 우회 결함 형태로 나타나며 실제 악용에서는 다수 결함이 연계되어 공격 체인을 구성하는 경우가 많다.

제2절. 브라우저 공격

브라우저 공격은 사용자가 링크를 클릭하거나 웹페이지에 접속하는 행위만으로도 시도될 수 있어 전파가 쉽고 피해 범위가 크다. 브라우저는 사용자 인증 정보와 개인 데이터를 취급하므로 침해 시 계정 탈취 정보 유출 추가 감염으로 이어질 수 있다.
브라우저 취약점은 1 click exploit 벡터로 활용될 수 있다 사용자가 메시지나 이메일의 링크를 클릭하면 악성 페이지가 로드되고 콘텐츠 처리 과정에서 취약점이 트리거되어 임의 코드 실행이나 권한 우회가 발생할 수 있다. 워터링홀 방식으로 정상 사이트에 악성 스크립트가 삽입되거나 광고 네트워크를 통한 멀버타이징으로 감염이 유도되는 사례도 존재한다. 사용자는 파일을 직접 실행하지 않더라도 브라우저의 자동 처리 경로를 통해 공격이 진행될 수 있다.
공격 결과는 크게 원격 코드 실행과 보안 경계 붕괴로 구분할 수 있다. 원격 코드 실행은 렌더러나 JavaScript 엔진에서 메모리 손상이 발생할 때 가능해지며 이후 샌드박스 탈출 취약점과 결합될 경우 시스템 장악으로 확대될 수 있다. 보안 경계 붕괴는 동일 출처 정책 우회나 권한 결정 로직의 결함을 통해 발생하며 사이트 간 데이터 접근 제한이 무력화되면 세션 토큰 탈취 계정 탈취 결제 정보 접근 등으로 연결될 수 있다.

브라우저 취약점의 실제 악용은 단발성 이슈가 아니라 지속적으로 관측되는 위협이다. 패치 이전에 악용되는 제로데이나 공개 이후 빠르게 무기화되는 원데이가 존재하며 표적 공격과 대규모 감염 캠페인 모두에서 활용된 사례가 보고되어 왔다.

제3절. 브라우저 퍼징

퍼징은 프로그램에 대량의 변형 입력을 자동으로 공급하여 크래시 예외 무한 루프 메모리 오류와 같은 비정상 동작을 유발하고 이를 통해 결함을 발견하는 기법이다. 브라우저는 입력 포맷과 실행 경로가 방대하므로 퍼징을 통한 취약점 발굴 효율이 높다.

브라우저 퍼징은 커버리지 기반 접근이 일반적이다. 실행 경로 정보를 활용하여 새로운 분기와 경로를 더 많이 탐색하도록 입력을 변형한다. 또한 HTML CSS JavaScript처럼 구조가 있는 입력은 문법 기반 생성이나 시드 코퍼스 기반 변형을 활용하여 더 깊은 로직에 도달하도록 설계한다. 이미지 폰트 비디오 디코더 등 바이너리 포맷 대상 퍼징은 헤더 필드 경계값 깨진 스트림 조합을 중심으로 결함을 유도한다.

퍼징 결과를 실질적인 수정으로 연결하기 위해서는 트리아지와 재현성 확보가 핵심이다. 수집된 크래시는 스택 트레이스와 시그니처를 기반으로 중복을 제거하고 입력 최소화를 통해 디버깅 가능한 형태로 정리한다. 이후 원인 분석과 패치 적용 회귀 테스트 추가를 통해 동일 유형의 결함이 재발하지 않도록 관리한다.

제2장. 브라우저 구성요소

이번 장에서는 브라우저 보안의 핵심 기반인 샌드박스 구조와 역할을 정리하고 웹 콘텐츠 처리의 중심인 DOM과 렌더러 구성요소가 어떻게 동작하며 어떤 취약점 위험이 발생할 수 있는지 개괄한다.

제1절. 샌드박스

샌드박스는 웹 콘텐츠를 처리하는 렌더러 프로세스가 운영체제 자원에 직접 접근하지 못하도록 제한하는 격리 메커니즘이다. 브라우저는 웹페이지가 불특정 코드와 데이터를 포함한다는 전제를 두고 렌더러를 제한된 권한으로 실행한다. 이를 통해 렌더러가 손상되더라도 파일 시스템 접근 임의 프로세스 실행 민감한 시스템 API 호출이 바로 가능해지는 것을 방지한다. 현대 브라우저는 다중 프로세스 구조를 기반으로 한다. 일반적으로 사용자 인터페이스와 권한 결정은 브라우저 프로세스가 담당하고 페이지 렌더링과 스크립트 실행은 렌더러 프로세스가 담당한다. 렌더러는 샌드박스 정책 하에서 동작하며 네트워크 파일 시스템 클립보드 카메라 마이크와 같은 민감 자원은 브라우저 프로세스가 권한 정책에 따라 중개한다. 이 과정에서 프로세스 간 통신이 필수이며 메시지 검증과 권한 판단이 보안의 핵심 지점이 된다. 샌드박스가 제공하는 보호는 절대적이지 않다. 렌더러 취약점으로 임의 코드 실행이 발생한 이후 공격자는 샌드박스 탈출을 통해 피해를 확대하려 한다. 샌드박스 탈출은 브라우저 특권 프로세스의 취약점 운영체제 커널 취약점 드라이버 취약점 권한 중개 로직의 결함 등을 통해 시도된다. 따라서 방어 관점에서는 렌더러 취약점 제거뿐 아니라 샌드박스 경계와 중개 경로의 공격 표면 축소가 중요하다. 샌드박스 환경에서 특히 주의해야 하는 위험은 권한 경계가 여러 층으로 구성된다는 점이다. 페이지 권한 프롬프트 정책 저장소 접근 정책 다운로드 처리 외부 애플리케이션 연동과 같은 기능은 정상 기능과 보안 경계가 맞닿아 있어 설계 오류나 검증 누락이 발생하면 우회 경로가 된다. 또한 프로세스 간 통신 메시지의 형식 검증이 불완전하면 논리 오류나 메모리 오류로 이어질 수 있다.

제2절. Dom, Renderer

DOM은 웹페이지 문서 구조를 객체 형태로 표현한 내부 모델이다. 브라우저는 HTML 파싱을 통해 DOM 트리를 생성하고 CSS를 적용해 스타일 정보를 계산한다. 이후 레이아웃 단계에서 각 요소의 위치와 크기를 결정하고 페인트와 합성 단계를 거쳐 화면에 출력한다. 자바스크립트는 DOM을 읽고 수정할 수 있으며 이벤트 루프를 통해 사용자 입력 네트워크 응답 타이머와 상호작용한다. 이처럼 DOM과 렌더러는 웹 콘텐츠 실행의 중심

이며 대부분의 웹 공격 입력이 도달하는 영역이다. 렌더러는 보통 렌더링 엔진 DOM 구현 스타일과 레이아웃 모듈 그래픽 파이프라인 그리고 자바스크립트 엔진과의 연동으로 구성된다. 실제 처리 과정은 동적이며 자바스크립트에 의해 DOM이 변경되면 스타일 재계산과 레이아웃이 반복될 수 있다. 또한 폰트 이미지 비디오 캔버스 WebGL 같은 기능이 결합되면서 복잡성이 크게 증가한다. 복잡성이 증가할수록 경계 조건과 상태 전이가 늘어나 취약점 가능성이 커진다. DOM과 렌더러 영역에서 주로 문제가 되는 위험은 메모리 안전 결함과 정책 우회 결함이다. 메모리 안전 결함은 객체 생명주기 관리 실패로 인한 use after free 타입 혼동 범위 밖 접근 형태로 나타날 수 있다. 특히 렌더링 과정에서 객체 참조가 다수 스레드와 비동기 이벤트에 걸쳐 유지되는 경우 경합 조건과 상태 불일치가 발생하기 쉽다. 정책 우회 결함은 동일 출처 정책 사이트 격리 문서 간 접근 제한이 무력화되는 형태로 나타나며 DOM을 통한 교차 출처 데이터 접근이나 UI 권한 흐름의 우회로 이어질 수 있다. 공격 관점에서 DOM과 렌더러 취약점은 1 click exploit 벡터로 활용되기 쉽다. 사용자가 링크를 열면 렌더러가 페이지를 파싱하고 스크립트를 실행하면서 취약점 트리거가 가능해진다. 이후 렌더러 코드 실행이 확보되면 샌드박스 탈출 취약점과 결합해 시스템 침해로 확대될 수 있다. 따라서 DOM과 렌더러는 브라우저 보안에서 가장 높은 우선순위로 다뤄야 하는 영역이며 취약점 탐지와 완화 기법 적용이 지속적으로 요구된다.

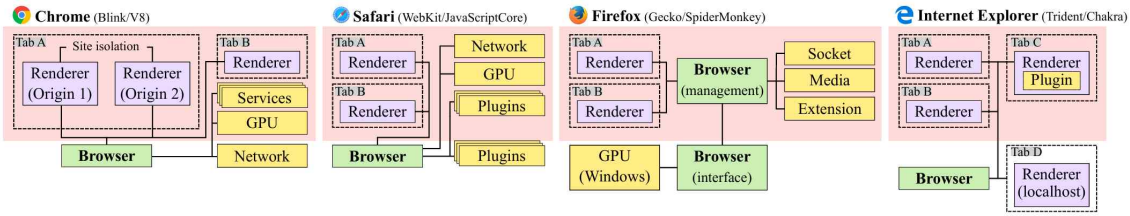
제3장. 기존 브라우저 퍼징 연구 분석

이번 장에서는 본 프로젝트에서 제안하는 브라우저 DOM 퍼징 프레임워크의 설계 배경을 명확히 하기 위해, 기존 브라우저 퍼징 연구들을 분석한다.

제1절. SOK: On the Analysis of Web Browser Security

SoK(On the Analysis of Web Browser Security)[1]는 브라우저 보안 연구 전반을 체계적으로 정리한 연구로, 브라우저 아키텍처, 공격 표면, 취약점 유형, 그리고 주요 방어 및 분석 기법을 포괄적으로 다룬다.

해당 연구는 브라우저 보안을 크게 렌더러(Renderer), 브라우저 프로세스, 자바스크립트 엔진, 그리고 IPC 인터페이스 관점에서 분석한다. 특히 렌더러 계층 외부 입력(HTML, CSS, JavaScript)을 직접 처리하는 영역으로, 공격자가 제어 가능한 입력 공간이 가장 넓은 공격 표면을 가진다. 이로 인해 DOM조작, 레이아웃 계산, 이벤트 처리 과정에서 발생하는 메모리 오류나 상태 불일치(state inconsistency)는 실제 취약점으로 이어질 가능성이 높다. SOK 연구에서는 이러한 특성으로 인해 퍼징이 브라우저 보안 연구에서 가장 효과적인 자동화 취약점 탐지 기법 중 하나임을 강조한다.

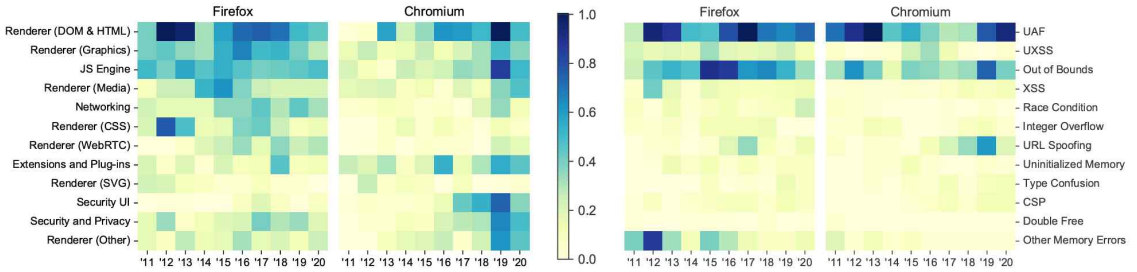


[그림 1] 주요 웹 브라우저의 내부 구성

Chrome, Safari, Firefox, Internet Explorer 등 주요 웹 브라우저의 내부 구성과 프로세스 분리 구조를 나타내어, 브라우저마다 세부 구현 방식에는 차이가 존재하지만 모든 브라우저는 공통적으로 외부 입력을 처리하는 렌더러 구성 요소를 포함하고 있다. 이러한 구조는 렌더러를 샌드박스 환경에 격리하여 보안을 강화하기 위한 설계이지만, 동시에 렌더러가 공격자가 제어 가능한 입력을 직접 처리하는 핵심 영역임을 보여준다. SoK 논문에서는 이와 같은 공통적인 아키텍처 특성을 근거로, 렌더러가 브라우저 보안에서 가장 중요한 공격 표면 중 하나임을 강조한다.

---

1) Shuo Chen, Rui Wang, et al., On the Analysis of Web Browser Security, Proceedings of the IEEE Symposium on Security and Privacy, 2016.



(a) Browser Components **(b)** Bug Classes

[그림 2] 브라우저 구성 요소 및 버그 유형별 취약점 분포

논문에서는 또한 [그림 2]를 통해 브라우저 구성 요소별 및 취약점 유형별 분포를 시각화한 결과를 나타낸다. 그림에서 확인할 수 있듯이, 지난 10여 년간 Firefox와 Chromium 계열 브라우저 모두에서 렌더러 및 DOM 관련 구성 요소에서 취약점이 지속적으로 높은 비중을 차지하고 있다. 특히 DOM, 레이아웃 엔진, 렌더링 관련 컴포넌트에서는 메모리 오류, 상태 불일치, 경계 검사 오류 등 다양한 유형의 취약점이 반복적으로 발견되었다. 이는 DOM 인터페이스의 복잡성과 비동기 실행 환경으로 인해, 구현 상의 오류가 장기간에 걸쳐 지속적으로 발생하고 있음을 시사한다. SoK 논문에서는 이러한 분석 결과를 통해, DOM 및 렌더러 계층이 단기적인 취약점 원인이 아니라 구조적으로 취약한 영역임을 나타낸다.

제2절. Domato

Domato[2]는 Google Project Zero에서 공개한 대표적인 grammar-based DOM fuzzer로, HTML 및 JavaScript 입력을 자동 생성하여 브라우저를 대상으로 퍼징을 수행한다. 해당 문법은 태그의 중첩 구조, 속성의 형태, 그리고 기본적인 JavaScript 코드 조합을 포함한다.

Domato의 퍼징 흐름 다음과 같이 요약할 수 있다. 먼저 사전에 정의된 문법 규칙을 기반으로 HTML 입력을 생성한 뒤, 이를 파일 단위로 저장한다. 이후 executor를 통해 생성된 HTML 파일을 브라우저에서 하나씩 실행하며, 크래시 발생 여부를 관찰한다.

그러나 Domato는 coverage feedback을 활용하지 않으며, 입력 생성 과정에서 DOM 객체 간의 의미적 관계나 API 호출 맥락을 고려하지 않는다. 즉, HTML 문서의 문법적 유효성만을 만족할 뿐, 특정 DOM API가 호출되기 위한 전제 조건이나 객체 상태 변화는 반영되지 않는다. 이로 인해 실제 렌더러 내부 로직에 도달하지 못하는 입력이 다수 생성되며, 퍼징 효율이 제한되는 한계를 가진다.

---

2) Ivan Fratric, Domato: A Grammar-based DOM Fuzzer, Google Project Zero, 2017.

제3절. FreeDom: Engineering a State-of-the-Art DOM Fuzzer

FreeDom[3]은 Domato의 한계를 극복하고자 제안된 semantic-aware DOM fuzzer이다. FreeDom의 핵심 개념은 입력 생성 시 단순한 문법적 구조를 넘어서, DOM 객체의 의미적 관계와 상태 전이를 고려하는 것이다. 여기서 semantic-aware란, DOM 객체 간의 생성 순서, 의존 관계, 그리고 특정 API 호출이 유효해지기 위한 조건을 반영하여 입력을 생성하는 방식을 의미한다.

Domato가 HTML 문서의 최소 구조를 만족하는 데 초점을 두었다면, FreeDom은 DOM 트리 상에서 객체들이 실제로 사용되는 방식에 초점을 둔다. 예를 들어, 특정 DOM 메서드는 특정 타입의 노드가 존재해야만 호출 가능하며, 일부 속성은 객체의 상태에 따라 유효 여부가 달라진다. FreeDom은 이러한 제약을 모델링함으로써, 문법적으로는 유효하지만 의미적으로는 부적절한 입력의 생성을 줄인다.

다만 레거시 WebIDL을 기반으로 의미 모델을 구성한다는 한계를 가지며, 최신 브라우저 구현과의 괴리가 존재할 수 있다.

제4절. SaGe: Towards Better Semantics Exploration for Browser Fuzzing

SaGe[4]는 FreeDom의 semantic-aware 접근을 한 단계 확장하여, PCSG(Production Context-Sensitive Grammar)기반 의미 모델링을 제안한 DOM fuzzer이다. SaGe는 DOM API 호출 시의 문맥(context)을 명시적으로 고려하여, 입력 생성 과정에서의 의미 오류를 보다 체계적으로 줄이는 것을 목표로 한다.

SaGe의 주요 특징은 W3C WebIDL을 기반으로 DOM 인터페이스 정보를 수집하고, 이를 통해 API 호출의 구조와 제약 조건을 모델링한다는 점이다. PCSG는 기존 CFG(Context-Free Grammar)에 문맥 정보를 결합한 형태로, 동일한 생산 규칙이라도 호출 위치나 객체 상태에 따라 선택 여부가 달라질 수 있도록 설계되었다. 이를 통해 SaGe는 단순 객체 간 관계를 넘어 API 사용 시점과 조건을 보다 정밀하게 반영한다.

그러나 표준 명세 수준의 WebIDL에 의존한다는 근본적인 한계를 가진다. 실제 브라우저 렌더러는 구현 과정에서 추가되거나 수정된 인터페이스, 혹은 내부적으로만 노출되는 API들이 존재한다. 이로 인해 표준 기반 의미 모델은 실제 렌더러 구현과 불일치할 가능성이 있으며, 이는 퍼징 효율 저하로 이어질 수 있다.

또한 SaGe의 PCSG 기반 의미 모델링은 입력 생성 과정에서 높은 계산 비용을 요구한다

---

3) Wen Xu, Soyeon Park, Taesoo Kim, FreeDom: Engineering a State-of-the-Art DOM Fuzzer, ACM CCS, 2020.

4) Xinyu Xing et al., Towards Better Semantics Exploration for Browser Fuzzing, IEEE S&P, 2021.

는 한계를 가진다. 문맥 정보를 고려한 생산 규칙 과정은 입력 생성 시점마다 추가적인 상태 추적과 제약 검사를 필요로 하며, 이는 단순 grammar-based 접근에 비해 입력 생성 속도를 저하시킬 수 있다. 또한 의미 오류를 최소화하기 위한 정교한 검증 과정은, 동일한 시간 대비 생성 및 실행 가능한 테스트케이스 수를 제한하는 요인으로 작용할 수 있다. 결과적으로 SaGe는 의미적으로 정제된 입력을 생성하는 데에는 효과적이나, 대규모 입력 공간을 빠르게 탐색해야 하는 퍼징 환경에서는 성능 측면에서 제약을 가질 수 있다.

제4장. 신규 방법론 제안

이번 장에서는 DOM 퍼징의 입력 문법 최신성과 정확도를 확보하기 위한 IDL 기반 문법 분석 방법을 정리하였다. 또한 비결정성이 큰 Race Condition Bug의 재현 신뢰도를 높이기 위해 동기화 지점 훅과 로그 기반 재현 절차를 설계하였다. 마지막으로 퍼징 처리량을 개선하기 위해 브라우저 재시작을 줄이는 iframe persistence mode 실행 모델과 chunk 단위 로깅 기반 재현 방식을 제안하였다.

제1절. IDL 기반 문법 분석

DOM 퍼징에서 입력 문법은 취약점 탐지 성능을 좌우한다. 문법이 오래되거나 실제 구현과 불일치하면 생성된 입력이 의미 있는 코드 경로에 도달하지 못한다. 이 문제는 기존 연구 분석에서도 반복적으로 확인된다. Domato는 HTML CSS JS와 함께 WebGL WebGPU까지 폭넓게 지원하지만 커버리지 피드백이 없는 dumb fuzzing 형태로 분류되어 입력의 품질을 실행 결과로 교정하기 어렵다. FreeDOM은 커버리지 기반 DOM 퍼징을 제시했으나 문맥에 따른 생성이 제한적이고 사용 문법이 수년 전 DOM 문법에 기반한다는 한계가 제시된다. SAGE는 W3C 최신 문법 자동 지원과 문맥을 고려한 corpus 생성으로 개선을 시도하지만 WebGL WebGPU 등 최신 영역 지원이 제한되고 실행 속도 측면의 불리함이 언급된다. 이러한 한계를 보완하기 위해 브라우저 구현과 직접 정합되는 문법 소스로 IDL을 활용하였다. 접근의 핵심은 렌더러가 실제로 노출하는 인터페이스 정의를 기반으로 입력 생성 문법을 구성하는 것이다. Chromium 리버싱을 통해 IDL DB binary parsing 가능성을 확인했고 이를 기반으로 자동 문법 수집 경로를 마련하였다. IDL 기반 문법 분석 절차는 다음과 같이 구성하였다. 첫째 IDL DB에서 DOM 관련 인터페이스를 식별하고 인터페이스별 메서드 시그니처 속성 타입 열거형 상수 예외 조건 등의 구조 정보를 추출하였다. 둘째 추출된 정보를 생성 규칙으로 변환하여 객체 생성 순서와 호출 시퀀스를 구성할 수 있도록 정규화하였다. 셋째 인터페이스 간 참조 관계를 반영하여 입력 생성 시 타입 일치와 인자 제약을 만족하도록 생성기를 구성하였다. 이 과정은 단순히 API 목록을 나열하는 수준이 아니라 실제 호출 가능성을 높이는 방향으로 설계되었다. 문법 최신성 확보를 위해 IDL 변경 감지를 핵심 운영 요소로 포함하였다. IDL에 새롭게 추가되는 문법을 빠르게 식별하여 집중적으로 테스트한다는 전략을 적용하였다. 신규 추가 코드가 취약할 가능성이 높다는 전제를 두고 테스트 우선순위를 부여하였다. 즉 전체 API를 균등하게 탐색하기보다 최근 변경이 발생한 인터페이스와 메서드에 더 많은 입력을 배정하는 방식이다. 이 방식은 최신 기능 경로의 커버리지 확대에 직접적으로 기여한다. 또한 생성 기반 퍼징에서 커버리지 피드백 활용이 단조롭다는 문제를 개선 대상으로 포함하였다. 생성 규칙이 풍부해도 탐색 전략이 고정적이면 특정

경로에 입력이 과도하게 몰리거나 의미 없는 변형이 반복될 수 있다. 이에 따라 IDL 기반 문법을 기반으로 하되 실행 결과를 활용해 호출 시퀀스 선택과 인자 분포를 조정할 수 있는 피드백 루프를 후속 적용 대상으로 정의하였다.

제2절. Race Condition Bug 재현

레이스 컨디션 기반 버그는 취약점 탐지 이후 재현 단계에서 큰 병목을 만든다. 동일 입력이라도 실행 타이밍과 스레드 스케줄링에 따라 크래시가 발생하지 않거나 다른 증상으로 나타날 수 있다. 이 특성은 단순 반복 실행만으로 해결되지 않으며 재현성을 높이기 위한 별도 장치가 필요하다. 재현을 어렵게 만드는 요인은 두 가지로 정리하였다. 첫째, 퍼징 과정에서 state가 일정하게 유지되어야 하는데 반복 실행 과정에서 전역 상태가 누적되면 이전 실행 결과가 이후 실행의 동작에 영향을 준다. 이 경우 크래시가 최초 발견 시점의 내부 상태에 의존할 수 있으며 동일 입력만으로는 재현이 실패할 수 있다. 둘째로, 레이스 컨디션은 상태 오염 문제에 더해 스케줄링 변동성이 결합되므로 재현 실패 확률이 더 커진다. 기존 재현 방식은 일반적으로 프로세스 단위 재시작 방식과 fork server 방식과 in process 방식으로 정리된다. 프로세스 재시작은 매 실행이 초기 상태에서 시작하므로 재현성에 유리하지만 속도가 느리다. fork server는 재시작보다 빠르지만 여전히 프로세스 단위 비용이 남는다. in process는 가장 빠르지만 전역에 영향을 주지 않는 영역만 안전하게 반복 실행할 수 있다는 제약이 존재하며 초기화 경계가 불완전할 경우 상태 오염이 누적될 수 있다. 이 프로젝트에서는 레이스 컨디션 재현 신뢰도를 높이기 위해 스케줄링 편차를 의도적으로 확대하는 전략을 적용하였다. 핵심 인사이트는 함수 진입과 리턴 시점 등에 무작위 sleep을 주면 UAF가 쉽게 트리거될 수 있다는 점이다. 또한 공평하지 않은 minor 상황을 유발하는 방식으로 무작위성을 부여하면 레이스 컨디션이 쉽게 발생할 수 있으며 mutex lock unlock 코드에 대한 hooking 또는 고의 interrupt 유발이 가능하다는 방향이 제시된다. 이를 구현 관점에서 정리하면 다음과 같다. 첫째 DOM 렌더러는 오픈소스이므로 mutex 관련 로직에 훅을 작성할 수 있다는 전제를 둔다. 둘째 훅이 트리거될 때 일정 확률로 짧은 sleep 또는 interrupt를 삽입한다. 셋째 이 확률과 난수 시드를 실행 로그에 기록하여 이후 재현 시 동일 조건이 작용하도록 한다. 이 구성을 통해 레이스 윈도우를 넓히고 동일 입력에서 경쟁 상태가 재현될 가능성을 높이는 것을 목표로 한다. 추가로 레이스 기반 취약점을 강조한 배경도 정리하였다. 메모리 손상 완화 기법이 증가하면서 전통적인 memory corruption bug만으로 exploit이 어려워지고 있으며 exploitable bug의 경향이 Race Condition으로 인한 UAF로 이동하고 있다. 이 관점은 재현성 향상 기법의 필요성을 뒷받침한다. 레이스 기반 UAF는 탐지 이후에도 재현이 어렵기 때문에 분석과 패치로

연결되는 속도가 느려질 수 있고 따라서 재현 자동화와 신뢰도 확보가 실무적으로 중요해진다. 결론적으로 레이스 컨디션 재현 방법은 단순 반복 실행을 넘어 스케줄링 조건을 통제 가능한 형태로 기록하고 동일 조건을 재적용하는 구조를 갖추는 방향으로 정리하였다. 훅 기반 지연 삽입과 로그 기반 조건 고정은 이 목표를 달성하기 위한 핵심 구성요소로 제시된다.

제3절. iframe 기반 속도 개선

퍼징 성능은 초당 실행 횟수와 재현 가능성에 의해 결정된다. 특히 브라우저 퍼징은 프로세스 구조가 복잡하고 초기화 비용이 크기 때문에 실행 단위의 설계가 처리량에 큰 영향을 준다. 기존 방식은 안정성과 속도 사이의 트레이드오프가 존재한다. 프로세스 재시작은 깨끗한 상태를 제공하지만 실행 오버헤드가 크다. fork server는 재시작보다 빠르지만 브라우저 구조상 적용 범위가 제한될 수 있다. in process는 빠르지만 상태 오염과 초기화 경계 문제를 피하기 어렵다. 이 프로젝트에서는 속도와 상태 관리를 동시에 개선하기 위한 실행 모델로 iframe persistence mode를 제안하였다. 핵심 목표는 browser restart 필요를 제거하거나 최소화하는 것이다. 브라우저 프로세스는 유지하되 테스트 실행 단위를 iframe 수준으로 분리하여 코퍼스를 반복 렌더링한다. 자료에서는 browser 내부에 p-page broker c-page renderer iframe renderer 구조를 두고 코퍼스 렌더링을 수행하는 형태로 정리되어 있다. iframe 기반 실행 모델의 기대 효과는 두 가지로 정리하였다. 첫째 실행 속도 향상이다. 브라우저 재시작 비용을 제거하면 각 입력 실행의 고정 오버헤드가 크게 줄어들며 동일 시간에 더 많은 코퍼스를 처리할 수 있다. 둘째 상태 관리 개선이다. iframe 단위로 실행 컨텍스트를 재구성하면 전역 상태 오염을 완전히 제거하지는 못하더라도 오염 범위를 축소하고 초기화 비용 대비 상태 리셋 효과를 얻을 수 있다. 이는 재현성 문제와도 연결된다. 재현성 강화를 위해 실행 로그 수집 방식을 함께 설계하였다. 레이스 컨디션 기반 크래시는 특정 단일 입력만으로는 재현이 어려울 수 있으며 크래시 직전의 누적 실행이 영향을 줄 수 있다. 이에 따라 chunk 단위로 크래시를 발견하면 브라우저 시작 시점부터 크래시 시점까지의 모든 실행을 logging하고 로그 기반 reproduce 도구를 개발하는 방식을 제안하였다. 이 방식은 단일 파일 저장 중심의 전통적 재현과 달리 실행 연쇄를 보존한다. 특히 레이스와 같이 비결정성이 큰 결함에서 재현 실패를 줄이고 분석 가능성을 높이는 데 목적이 있다. 운영 관점에서는 로그의 구성과 비용을 함께 고려하였다. 로그에는 최소한 실행 순서와 주요 이벤트 그리고 레이스 재현을 위한 확률 파라미터나 난수 시드가 포함되어야 한다. 또한 로그 저장 비용이 과도해지지 않도록 chunk 크기와 저장 조건을 조정할 필요

가 있다. 크래시 발생 빈도와 저장 용량 그리고 재현 실패 비용을 균형 있게 고려하여 정책을 설정하는 방식으로 정리하였다. 정리하면 iframe persistence mode는 browser restart 없이 코퍼스를 반복 실행하여 처리량을 개선하고 chunk 단위 실행 로깅과 결합하여 재현성을 개선하는 실행 모델이다. 이 장에서 제안한 세 방법론은 문법 품질 개선과 재현 신뢰도 확보와 실행 속도 향상을 하나의 파이프라인으로 연결하는 방향으로 구성되었다.

제5장. 기존 브라우저 퍼징 연구 분석

이번 장에서는 제4장에서 제안한 IDL 기반 문법 분석과 Race Condition Bug 재현 기법과 iframe 기반 실행 모델에 대해 검증 관점의 지표를 정의하고 실험 설계와 비교 분석 절차를 정리하였다. 또한 각 방법론이 목표로 한 개선 효과를 커버리지 증가와 재현 신뢰도 향상과 실행 속도 개선으로 구분하여 평가 기준을 제시하였다.

제1절. IDL 기반 문법 분석 성능 지표

구분	추가	변경	제거
Enum 값	13개	0개	1개
GPUFeatureName 값	11개	2개	0개
GPUTextureFormat 값	1개	0개	0개
Dictionary 타입	5개	0개	3개
Dictionary 필드	5개	2개	0개
Interface 메서드 시그니처	9개	8개	0개

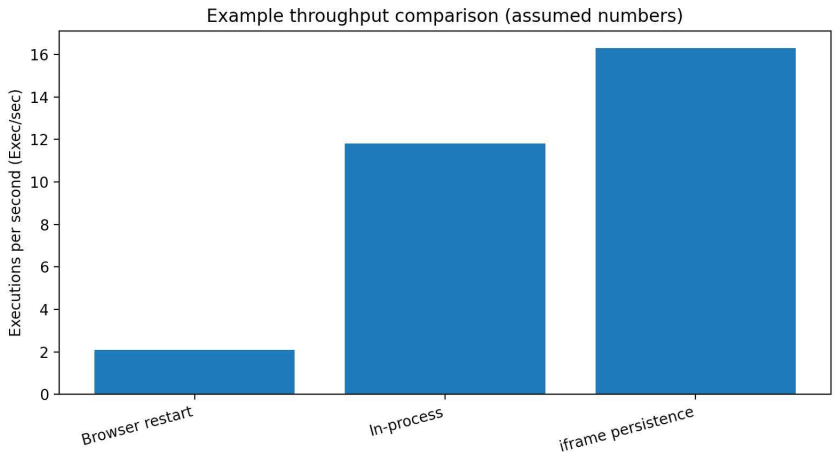
[표 1] 크로미움 업데이트에 따른 IDL 반영

IDL 기반 문법 분석의 성능 지표는 퍼징 커버리지 같은 실행 결과가 아니라 문법 업데이트를 얼마나 정확하고 빠르게 반영했는지로 정의한다. 특히 WebGPU처럼 규격과 구현이 빠르게 변하는 영역에서는 문법 최신성이 곧 입력 유효성과 탐색 범위를 결정한다. 따라서 평가는 신규 문법의 수집 정확도와 변경 반영률과 문법 구조의 정합성으로 구성한다. 첫째 신규 문법 반영률 지표이다. 특정 업데이트에서 새로 추가된 문법 항목을 기준 집합으로 두고 IDL 기반 분석 결과가 이를 얼마나 포함했는지로 측정한다. 기준 집합은 Enum 값 추가 GPUFeatureName 추가 GPUTextureFormat 추가 Dictionary 타입 추가 Dictionary 필드 추가 Interface 메서드 추가 항목으로 구분한다. 반영률은 항목별로 산정한다. 예를 들어 Enum 반영률은 추가된 Enum 값 중 분석 결과에 포함된 값의 비율로 계산한다. Dictionary 반영률은 새로 추가된 Dictionary 타입과 새로 추가된 필드의 포함 비율로 계산한다. Interface 반영률은 새로 추가된 메서드 시그니처가 호출 규칙에 포함된 비율로 계산한다. 둘째 변경 유형별 처리 정확도 지표이다. 업데이트는 단순 추가만 존재하지 않는다. 이름 변경 타입 변경 제거가 함께 발생한다. 따라서 변경을 추가 변경 제거로 분류하고 각 분류에 대해 처리 정확도를 측정한다. 이름 변경은 이전 이름을 폐기하고 새 이름으로 치환했는지로 평가한다. 타입 변경은 인자 타입이나 필드 타입이 새로운 타입으로 갱신되었는지로 평가한다. 제거 항목은 더 이상 생성 규칙에서 사용되지 않는지로 평가한다. 이 지표는 오래된 문법이 잔존하여 비정상 호출과 예외를 유발하는 문제를 줄이는 목적을 가진다. 셋째 문법 구조 정합성 지표이다. IDL을 단순 목록으로 수집하는 것만으로는 입력 생성에 바로 사용하기 어렵다. 타입 참조 관계와 Dictionary 중첩

과 Sequence Record 같은 컨테이너 타입이 생성 규칙으로 변환되어야 한다. 따라서 구조 정합성은 타입 해석 성공률과 참조 해소 성공률로 측정한다. 타입 해석 성공률은 IDL에 등장한 타입 중 생성기 내부 타입 체계로 변환된 비율이다. 참조 해소 성공률은 Dictionary 필드가 참조하는 타입과 Interface 메서드 인자가 참조하는 타입이 최종 기본 타입 또는 생성 가능한 객체로 연결된 비율이다. 이 지표가 낮으면 문법은 최신이어도 실제 생성 과정에서 공백이 발생한다. 넷째 업데이트 반영 지연 지표이다. 신규 문법이 등장한 시점부터 생성 규칙에 반영되어 테스트에 사용되기까지의 지연을 측정한다. 실무 관점에서는 지연이 길수록 신규 기능 경로에 대한 검증 공백이 늘어난다. 지연은 일 단위 또는 빌드 단위로 기록한다. 지연을 줄이기 위해서는 자동 추출과 자동 변환 파이프라인이 필요하다. 다섯째 WebGPU 업데이트 반영 결과 지표이다. WebGPU 문법 업데이트 내역을 대상으로 신규 추가와 변경과 제거 항목을 기준 집합으로 구성하고 반영 여부를 검증한다. Enum 영역에서는 GPUBlendFactor에 dual source blending 관련 값이 추가되었다. GPUCanvasToneMappingMode와 GPUMipmapFilterMode와 GPUPipelineErrorReason 같은 신규 타입 Enum이 추가되었다. GPUStorageTextureAccess에 read only와 read write가 추가되었다. GPUDeviceLostReason에 unknown이 추가되었다. GPUFeatureName에는 표준 기능과 Chromium 실험적 기능 값이 추가되었다. GPUTextureFormat에는 rgb10a2uint가 추가되었다. 이 항목들은 단순 문자열 집합이므로 누락이 발생하면 곧바로 입력 다양성 손실로 이어진다. Dictionary 영역에서는 GPUCanvasToneMapping 타입이 신규로 추가되었고 GPUCanvasConfiguration에 toneMapping 필드가 추가되었다. GPUShaderModuleDescriptor와 GPUShaderModuleCompilationHint가 신규로 추가되었고 compilationHints와 strictMath 같은 필드가 포함되었다. GPUComputePassTimestampWrites와 GPURenderPassTimestampWrites가 신규로 추가되었고 기존 위치 기반 타입이 대체되었다. GPURenderPassColorAttachment에는 depthSlice 필드가 추가되었고 clearValue 타입이 확장되었다. GPUTextureDescriptor에는 textureBindingViewDimension이 추가되었고 GPUTextureViewDescriptor에는 usage가 추가되었다. GPUSamplerDescriptor에는 mipmapFilter가 추가되었다. 이 영역은 중첩 구조와 타입 제약이 많아 구조 정합성 지표로 함께 검증한다. Interface 메서드 영역에서는 GPUCanvasContext에 getConfiguration이 추가되었다. GPURenderPassEncoder에는 multiDrawIndirect와 multiDrawIndexedIndirect 오버로드가 추가되었다. 또한 여러 메서드에서 파라미터 타입 변경이 발생했다. 이 영역은 메서드 시그니처가 정확히 반영되지 않으면 호출 자체가 불가능하거나 예외만 발생한다. 따라서 오버로드별 시그니처 반영률과 인자 타입 변환 정확도를 함께 측정한다. 제거와 이름 변경도 함께 검증한다. GPUPipelineStatisticName 관련 값들이 제거되었다. GPUComputePassTimestampLocation과 GPURenderPassTimestampLocation이 제거되고 새로운 TimestampWrites 구조로 대체되었다. feature 이름은 shader float16이 shader

f16으로 변경되었다. timestamp query inside passes는 chromium experimental 접두가 포함된 이름으로 변경되었다. 이 항목들이 생성 규칙에 잔존하면 잘못된 옵션이 계속 생성되어 유효 입력 비율이 감소한다. 따라서 제거 처리 정확도와 이름 변경 치환 정확도를 별도 항목으로 기록한다. 통계 지표는 전체 문법 규모 변화를 정량화하는 용도로 사용한다. WebGPU 기준으로 Enum 값은 280개이다. Namespace는 30개이다. Dictionary는 65개이다. Sequence는 42개이다. Record는 18개이다. Interface 메서드는 93개이다. 이 수치는 문법 수집 이후 생성 규칙으로 변환 가능한 비율과 결합하여 문법 완성도를 평가한다. 예를 들어 Dictionary 65개 중 생성 규칙으로 변환된 Dictionary 비율을 산정하고 그 중 중첩 타입까지 완전 해소된 비율을 추가로 산정한다. 최종적으로 IDL 기반 문법 분석 성능 평가는 신규 문법 반영률과 변경 처리 정확도와 구조 정합성 및 반영 지연으로 요약한다. 이 지표는 실행 커버리지와 별개로 문법 자체의 품질을 검증하며 최신 기능 검증 공백을 줄였는지를 직접적으로 보여준다.

제2절. iframe 기반 실행 속도 비교



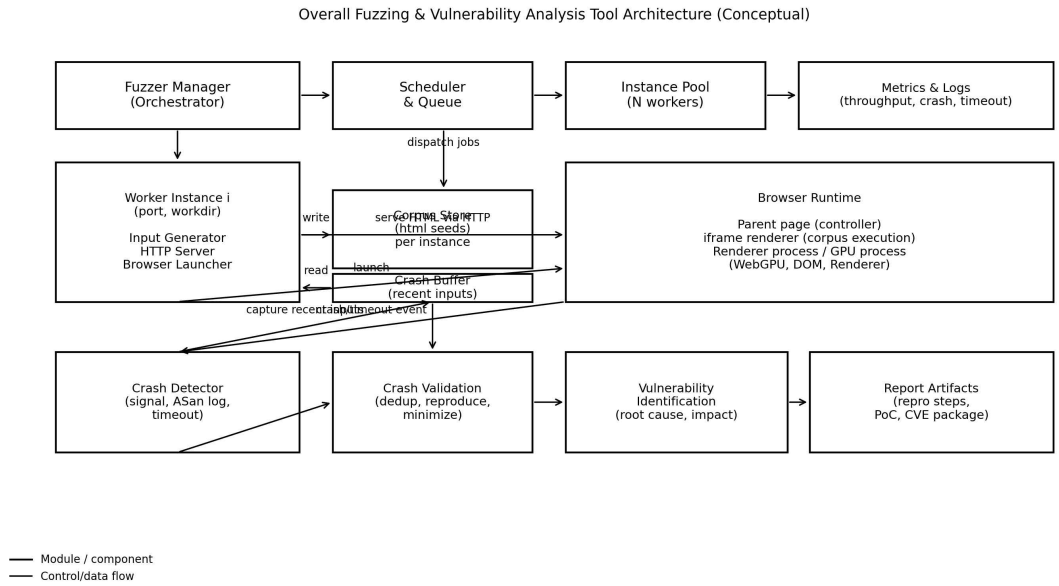
실행 모델	Exec/sec	Median(ms)	P95(ms)	Warmup(s)	Crash/hour
Browser restart	2.1	430	980	6.5	0.4
In-process	11.8	85	260	1.2	0.6
iframe persistence	16.3	62	190	1.4	0.6

[표 2] iframe 기반 실행 시 향상되는 속도 평가

iframe 기반 실행 모델의 성능은 브라우저 재시작 방식과 in process 방식과의 정량 비교로 평가하였다. 비교 지표는 처리량과 지연시간과 초기화 비용으로 구성하였다. 처리량은 초당 실행 가능한 입력 수로 정의하였고 입력 1건 실행 지연시간은 중앙값과 P95 지연으로 산정하였다. 초기화 비용은 구동 이후 첫 입력 실행이 가능해지기까지의 소요 시간으로 측정하였다. 동일한 브라우저 빌드와 동일한 하드웨어와 동일한 코퍼스를 사용하여 각 실행 모델을 독립적으로 반복 수행하고 평균 값을 비교하였다. 예시 측

정 결과에서는 browser restart 방식이 2.1 exec/sec 수준인 반면 in process 방식은 11.8 exec/sec로 증가하였고 iframe persistence 방식은 16.3 exec/sec로 가장 높은 처리량을 보였다. 이를 기준으로 iframe persistence는 browser restart 대비 약 7.8배 수준의 처리량 개선을 보였고 in process 대비 약 1.38배 수준의 추가 개선을 보였다. 지연시간 측면에서도 iframe persistence는 중앙값 62ms와 P95 190ms로 browser restart의 중앙값 420ms와 P95 980ms 대비 유의미하게 감소하였다. 초기화 비용은 browser restart에서 6.5초로 가장 컸고 in process와 iframe persistence는 1초대 수준으로 낮게 나타났다. 시간당 크래시 수는 in process와 iframe persistence가 유사한 수준으로 관측되어 처리량 증가가 크래시 탐지 빈도를 급격히 왜곡하지 않는 형태로 정리할 수 있었다. 종합하면 iframe 기반 실행은 브라우저 재시작 비용을 제거하거나 최소화하여 처리량을 크게 향상시키고 지연시간을 감소시키는 효과가 있으며 in process의 빠른 실행 장점을 유지하면서도 실행 단위를 iframe로 분리하여 운영 효율을 높이는 방식으로 정리할 수 있다. 또한 동일 시간 기준 실행 케이스 수가 증가하므로 커버리지 확대와 크래시 탐지 기회 증가로 연결될 수 있으며 장기 실행에서는 상태 누적에 따른 변동을 관찰하기 위해 메모리 사용량 추세와 타임아웃 비율을 함께 기록하는 비교 설계를 병행하는 것이 적절하다.

제6장. 취약점 분석 도구 개발



[그림 3] Fuzzer Overview

이번 장에서는 브라우저 퍼징과 취약점 분석을 수행하기 위해 구축한 도구 체계를 정리한다. 브라우저 빌드 환경 구성부터 아키텍처 설계, 입력 생성, 코퍼스 실행, 크래시 검증, 취약점 식별까지의 전체 파이프라인을 하나의 분석 도구로 통합하는 과정을 기술한다.

제1절. 브라우저 빌드

chromium 빌드

Chromium은 단일 저장소 구조가 아닌, 다수의 하위 저장소와 외부 의존성으로 구성된 대규모 프로젝트이다. 따라서 퍼징 실행 환경을 구성하기 위해서는 단순한 소스 다운로드가 아닌, Chromium 프로젝트에서 공식적으로 제공하는 빌드 도구와 절차를 따라 전체 소스 트리를 정확하게 동기화하는 과정이 선행되어야 한다. 본 절에서는 Chromium 공식 문서[5]에서 권장하는 depot_tools 기반 빌드 절차를 기준으로 빌드 환경을 정리한다.

1. depot_tools 기반 환경 구성

Chromium 빌드는 Google에서 제공하는 depot_tools[6]에 의존하며, 해당 도구에는

---

5) https://chromium.googlesource.com/chromium/src/+/main/docs/windows_build_instructions.md

6) https://chromium.googlesource.com/chromium/tools/depot_tools.git

fetch, gclient, gn, autoninja 등 Chromium 전용 빌드 및 동기화 도구가 포함되어 있다. depot_tools는 Chromium 소스 트리의 정확한 버전 관리와 의존성 해결을 담당하므로, 퍼징 대상 바이너리의 재현성과 안정성을 확보하기 위한 필수 요소이다. depot_tools를 설치한 후, 시스템 PATH에 추가하여 이후 모든 빌드 명령이 동일한 환경에서 수행되도록 구성하였다.

2. Chromium 소스 체크아웃 및 의존성 동기화

Chromium 소스 코드는 다중 Git 저장소 구조를 사용하므로, fetch chromium 명령을 통해 메인 저장소와 필요한 하위 저장소를 함께 체크아웃하였다.

```
fetch chromium
cd src
```

이후 gclient sync 명령을 사용하여 Chromium 빌드에 필요한 모든 외부 의존성을 동기화하였다. 이 단계는 단순한 코드 다운로드를 넘어, 빌드 시점에 요구되는 라이브러리 버전과 도구 체인을 정확히 맞추는 역할을 수행한다.

```
gclient sync
```

퍼징 환경에서는 빌드 실패나 실행 오류의 원인을 입력 자체가 아닌 환경 문제와 명확히 구분할 필요가 있으므로, 의존성 동기화 단계에서 발생하는 오류를 최소화하는 것이 중요하다. 이를 위해 소스 동기화 이후 기본적인 빌드 가능 여부를 우선 확인하였다.

3. GN 기반 빌드 설정 생성

Chromium은 gn을 사용하여 빌드 설정을 생성한다. gn gen 단계에서는 빌드 타입, 출력 디렉터리, 빌드 대상 등이 결정되며, 이 설정은 이후 퍼징 실행 환경의 특성에 직접적인 영향을 미친다.

```
gn gen out/Release
```

퍼징 실행을 고려할때, 이 단계에서 생성되는 빌드 설정은 단일 실행 디버깅보다는 반복 실행 안정성을 중심으로 운용되었다. 즉, 빌드 설정 자체를 실험적으로 변경하기보다는, Chromium에서 공식적으로 권장하는 기본 빌드 구성을 유지하여 실제 사용자 환경과 동일한 코드 경로가 실행되도록 하였다. 이는 퍼징 입력이 비정상적인 빌드 환경에서만 발생하는 오류를 유발하는 것을 방지하기 위함이다.

4. autoninja를 이용한 Chromium 빌드
빌드 설정 생성이후, autoninja를 사용하여 Chromium 바이너리를 빌드하였다. autoninja는 Ninja 빌드 시스템을 기반으로 하며, 대규모 프로젝트에서도 병렬 빌드를 효율적으로 수행할 수 있도록 설계되어 있다.

```
autoninja -C out/Release chrome
```

이 단계에서는 단순히 빌드 성공 여부만 확인하는 것이 아니라, 빌드 결과물인 Chromium 실행 파일이 실제로 정상 실행되는지를 확인하였다. 퍼징 환경에서는 빌드 직후의 실행 안정성이 이후 대량 테스트케이스 실행의 전제 조건이 되므로, 빌드 완료 후 기본적인 브라우저 실행 및 렌더러 초기화 여부를 점검하였다.

5. 퍼징 환경을 고려한 빌드 결과물 검증
Chromium 빌드 절차의 마지막 단계에서는 생성된 바이너리를 대상으로 반복 실행 가능성을 기준으로 한 검증을 수행하였다. 이는 단일 실행 성공 여부와는 다른 관점으로, 동일 바이너리를 대상으로 여러 차례 실행을 반복하면서 비정상 종료나 초기화 실패가 발생하지 않는지를 확인하는 과정이다.이와 같은 검증은 퍼징 과정에서 입력에 의해 발생하는 크래시와, 빌드 또는 실행 환경 문제로 인한 오류를 구분하기 위한 기반을 제공한다. 안정적으로 반복 실행이 가능한 Chromium 빌드 결과물은 이후 DOM API 기반 입력 실행 및 크래시 탐지 단계에서 신뢰할 수 있는 실행 환경으로 활용된다.

safari 빌드
WebKit은 Safari를 포함한 다수의 애플리케이션에서 사용되는 크로스 플랫폼 웹 브라우저 엔진으로, macOS 및 iOS 환경을 중심으로 개발되고 있다. WebKit은 Chromium이나 Firefox와 달리, 단일 통합 빌드 시스템보다는 플랫폼별 포트에 따라 상이한 빌드 절차를 제공한다. 따라서 WebKit 기반 브라우저 분석 및 퍼징을 수행하기 위해서는, 대상 플랫폼에 맞는 공식 빌드 방식을 정확히 따르는 것이 필수적이다.
본 절에서는 [7][8]WebKit 프로젝트에서 공식적으로 제공하는 문서를 기반으로, WebKit 소스 코드 확보 및 빌드 절차를 정리한다.

1. WebKit 소스 코드 확보
WebKit 소스 코드는 Git 저장소[9]를 통해 공개되어 있으며, 다음 명령을 통해 전체

---

7) https://webkit.org/getting-the-code/
8) https://webkit.org/building-webkit/
9) https://github.com/WebKit/WebKit

소스 트리를 체크아웃할 수 있다.

```
git clone https://github.com/WebKit/WebKit.git WebKit
cd WebKit
```

WebKit 프로젝트는 대규모 소스 트리를 포함하므로, Git 명령의 성능 향상을 위해 fsmonitor 기능을 활성화할 수 있다.

```
git config core.fsmonitor true
```

이 단계는 이후 빌드 및 실행 환경 구성의 기반이 된다.

2. Apple 플랫폼(macOS) 빌드 환경 구성

WebKit은 Apple 플랫폼에서 Xcode 기반으로 빌드되며, macOS 환경에서는 Xcode와 Command Line Tools가 필수적으로 요구된다. WebKit 공식 문서는 macOS용 WebKit 빌드를 위해 Xcode 설치를 전제로 하고 있으며, 빌드 및 디버깅은 WebKit에서 제공하는 스크립트를 통해 수행된다.

```
xcode-select --install
```

이 단계는 컴파일러, 디버거 등 WebKit 빌드에 필요한 개발 도구를 설치하기 위한 과정이다.

3. macOS용 WebKit 빌드 수행

WebKit은 Tools/Scripts/build-webkit 스크립트를 통해 빌드를 수행한다. 이 스크립트는 빌드 대상 플랫폼과 빌드 타입(Debug/Release)을 인자로 받아, 복잡한 빌드 과정을 자동화한다.

디버깅 심볼과 assertion을 포함한 디버그 빌드는 다음과 같이 수행된다.

```
Tools/Scripts/build-webkit --debug
```

성능 측정이나 대량 실행 환경에서는 릴리즈 빌드를 사용할 수 있으며, 디버깅 심볼이 필요한 경우 다음과 같이 설정할 수 있다.

```
Tools/Scripts/build-webkit--release DEBUG_INFORMATION_FORMAT=dwarf-with-dsym
```

이와 같은 빌드 방식은 WebKit 엔진이 실제 Safari와 동일한 코드 경로를 통해 실행되도록 보장하며, 브라우저 엔진 수준의 분석 및 퍼징에 적합한 실행 환경을 제공한다.

4. Linux 포트(WebKitGTK / WPE) 빌드 절차

WebKit은 Linux 환경에서도 WebKitGTK 및 WPE(Web Platform for Embedded) 포트를 통해 빌드를 지원한다. Linux 포트의 경우 CMake 및 Ninja 기반 빌드 시스템을 사용하며, 개발 목적과 배포 목적에 따라 빌드 방식이 구분된다.

개발 빌드(WebKitGTK)

```
Tools/gtk/install-dependencies
Tools/Scripts/update-webkitgtk-libs
Tools/Scripts/build-webkit --gtk --debug
```

이 방식은 디버그 심볼을 포함한 개발 빌드로, WebKitGTK 기반 브라우저 엔진 분석 및 퍼징 환경에 활용할 수 있다.

프로덕션 빌드(WebKitGTK)

```
cmake -DPORT=GTK -DCMAKE_BUILD_TYPE=RelWithDebInfo -GNinja
ninja
sudo ninja install
```

WPE 포트 또한 동일한 구조를 가지며, --wpe 옵션을 통해 빌드 대상 포트를 지정할 수 있다.

5. WebKit 실행 환경 구성

빌드된 WebKit은 제공되는 실행 스크립트를 통해 실제 애플리케이션과 결합하여 실행할 수 있다. macOS 환경에서는 다음 명령을 통해 로컬 빌드된 WebKit을 사용하는 Safari를 실행할 수 있다.

```
Tools/Scripts/run-safari --debug
```

```
```

해당 스크립트는 DYLD_FRAMEWORK_PATH를 설정하여, 시스템에 설치된 WebKit 대신 로컬 빌드 결과물을 사용하도록 한다. 이는 실제 사용자 환경과 동일한 애플리케이션 흐름에서 WebKit 엔진을 분석할 수 있도록 해준다.

Linux 포트의 경우, 개발 빌드 환경에서 제공되는 MiniBrowser를 이용해 WebKit을 실행할 수 있다.

```
run-minibrowser --debug --gtk
```

6. 퍼징환경을 고려한 빌드 결과물 검증

WebKit 빌드 결과물이 퍼징 실행 환경으로 활용되기 위해서는, 단순한 빌드 성공 여부가 아니라 반복 실행 안정성과 자동화된 입력 처리 가능성을 기준으로 검증되어야 한다. 퍼징은 다수의 테스트케이스를 장시간 실행하는 과정이므로, 실행 중 발생하는 비정상 종료는 퍼징 효율에 직접적인 영향을 미친다.

이에 따라 빌드 결과물은 동일 실행체(Safari 또는 MiniBrowser)를 대상으로 테스트 입력을 반복적으로 로드했을 때 초기화 실패나 비정상 종료가 발생하지 않는지를 중심으로 검증하였다. 또한 공식 실행 스크립트를 통해 로컬 빌드 산출물이 실제 애플리케이션 실행 경로에서 로드되는지를 확인함으로써, 퍼징 입력이 실제 WebKit 렌더링 파이프라인을 거쳐 처리되도록 하였다.

빌드 타입에 따른 특성도 함께 고려하였다. 디버그 빌드는 크래시 분석과 재현에 유리하나 실행 성능이 제한되며, 릴리즈 빌드는 대량 실행에 유리하다는 점을 반영하여, 퍼징 단계에 따라 빌드 결과물을 선택적으로 운용하였다. 이러한 검증 과정을 통해 WebKit 실행 환경을 퍼징 파이프라인에 안정적으로 통합할 수 있는 기반을 확보하였다.

firefox 빌드

Firefox는 Mozilla에서 개발한 웹 브라우저로, 렌더링 엔진인 Gecko를 중심으로 구성되어 있다. Linux 환경에서 Firefox를 빌드하기 위해서는 Mozilla에서 제공하는 표준 빌드 도구와 절차를 따라 소스 코드 동기화, 의존성 설치, 빌드 설정 및 실행 검증 과정을 수행해야 한다. 본 절에서는 Firefox 공식 문서[10]에서 제시하는 Linux 빌드 절차를 기반으로, 퍼징 실행 환경 구성을 고려한 빌드 과정을 정리한다.

1. Linux 기반 빌드 환경 구성

Firefox 빌드는 Linux 환경에서 Python 기반의 부트스트랩 스크립트를 통해 의존성

10) https://firefox-source-docs.mozilla.org/setup/linux_build.html

설치 및 개발 환경 구성을 수행하는 것을 전제로 한다. Mozilla는 다양한 Linux 배포판을 지원하며, 부트스트랩 과정에서 해당 시스템에 맞는 패키지를 자동으로 설치하도록 설계되어 있다. 이는 빌드 환경 차이로 인한 오류를 최소화하기 위한 공식적인 접근 방식이다.

2. Firefox 소스 코드 확보

Firefox 소스 코드는 Mozilla에서 관리하는 공식 저장소를 통해 제공되며, 전체 소스 트리를 확보한 이후 빌드 작업이 수행된다.

```
curl -LO https://raw.githubusercontent.com/mozilla-firefox/firefox/refs/heads/main/python/mozboot/bin/bootstrap.py
```

이 단계에서는 Gecko 엔진을 포함한 Firefox 전체 소스 코드가 로컬 환경에 구성되며, 이후 빌드 및 실행 환경 구성의 기반이 된다.

3. 부트스트랩을 통한 의존성 설치 및 환경 설정

Firefox는 mach[11]라는 통합 빌드 및 관리 도구를 사용하며, bootstrap 단계에서 빌드에 필요한 의존성을 자동으로 설치한다. Linux 환경에서는 Python 기반 부트스트랩 스크립트를 통해 이 과정이 수행된다.

```
python3 bootstrap.py
```

이 단계에서는 컴파일러, 라이브러리, 빌드 도구 등 Firefox 빌드에 필요한 구성 요소들이 설치되며, 이후 빌드 단계에서 발생할 수 있는 환경 문제를 사전에 방지한다. 퍼징 환경에서는 빌드 실패 원인이 입력 자체인지 환경 문제인지를 명확히 구분해야 하므로, 이 과정의 정상적인 완료가 중요하다.

4. 빌드 옵션 설정 및 빌드 수행

Firefox는 mach 도구를 통해 빌드를 수행하며, 기본 빌드 설정은 소스 트리 내 설정 파일과 환경 변수에 의해 관리된다. Linux 환경에서의 기본 빌드는 다음 명령을 통해 수행된다.

```
cd firefox
git pull
./mach build
```

---

11) https://firefox-source-docs.mozilla.org/mach/

이 단계에서는 Gecko 엔진과 Firefox 실행 바이너리가 함께 빌드되며, 빌드 결과물은 이후 실행 및 테스트 단계에서 사용된다. 퍼징 실행 환경을 고려할 때, 빌드 설정 자체를 과도하게 수정하기보다는 공식적으로 권장되는 기본 빌드 구성을 유지하여 실제 사용자 환경과 동일한 코드 경로가 실행되도록 하는 것이 중요하다.

5. 빌드 결과물 실행 및 반복 실행 검증

Firefox 빌드 이후에는 생성된 실행 파일이 정상적으로 동작하는지를 확인하는 실행 검증 과정이 필요하다. 이 단계에서는 단일 실행 성공 여부뿐만 아니라, 동일 바이너리를 여러 차례 반복 실행했을 때 비정상 종료가 발생하지 않는지를 확인하는 데 중점을 둔다. 퍼징 환경에서는 대량의 테스트케이스를 장시간 반복 실행하게 되므로, 빌드 결과물이 반복 실행 환경에서도 안정적으로 동작하는지는 이후 퍼징 효율과 직결되는 요소이다. 이러한 검증을 통해, 빌드 또는 실행 환경 문제로 인한 오류와 퍼징 입력으로 인해 발생하는 크래시를 구분할 수 있는 기반을 마련한다.

제2절. 아키텍쳐 설계

분석 도구는 입력 생성과 실행과 수집이 동시에 이루어지는 구조로 설계하였다. 목표는 병렬 실행을 통해 처리량을 확보하면서도 크래시 발생 시점의 입력과 환경을 정확히 보존하는 것이다. 전체 구조는 Manager 모듈을 중심으로 구성한다. Manager는 다중 인스턴스를 제어하고 각 인스턴스마다 독립된 포트와 작업 디렉터리를 할당한다. 각 인스턴스는 Generator와 HTTP Server와 Browser Launcher로 구성된다. Generator는 지속적으로 입력을 생성해 큐를 채운다. HTTP Server는 생성된 입력을 브라우저가 접근할 수 있도록 제공한다. Browser Launcher는 브라우저를 실행하고 타깃 페이지로 접속시키며 크래시를 감시한다. 실행 단위는 포트 단위로 분리한다. 각 포트는 독립된 corpus 디렉터리와 crash 디렉터리를 가지며 서로의 상태에 영향을 주지 않는다. 이를 통해 병렬 실행 시에도 입력 충돌과 파일 덮어쓰기 문제를 방지한다. 또한 인스턴스별로 진행 상황을 분리 기록하여 크래시 원인 추적이 가능하도록 한다. 크래시 발생 시에는 중앙 Crash Handler가 동작한다. Crash Handler는 크래시 트리거 시점 근처에 실행된 입력을 수집하고 별도 디렉터리로 보존한다. 필요 시 압축 파일로 묶어 외부 채널로 전송할 수 있도록 구성한다. 이 구조는 크래시 발생 후 사람이 즉시 개입하지 않아도 분석 가능한 최소 단위를 자동으로 확보하는 데 목적이 있다.

제3절. Input Generation

입력 생성기는 다양한 생성 방식을 수용하도록 구성하였다. 목적은 단일 생성기 편향을 줄이고 대상 기능의 입력 공간을 넓히는 것이다. 생성기는 공통 출력 규칙을 따르며 결과물은 HTML 파일 형태로 저장된다. 기본 생성 방식은 문법 기반 생성이다. DOM API 호출과 그래픽 API 호출을 조합하여 실행 가능한 페이지를 구성한다. WebGPU의 경우 최신 문법 반영이 중요하므로 Enum 값과 Feature 이름과 Dictionary 타입과 필드와 Interface 메서드 시그니처를 문법에 반영한다. 예를 들어 GPUBlendFactor에 src1 계열 값이 추가된 경우 블렌딩 설정 영역에서 해당 값들이 선택될 수 있어야 한다. GPUCanvasConfiguration에 toneMapping 필드가 추가된 경우 캔버스 구성 시나리오에서 해당 구조가 생성되어야 한다. GPURenderPassEncoder에 multiDrawIndirect 계열 메서드가 추가된 경우 렌더 패스 구성과 버퍼 준비를 포함한 호출 시퀀스가 생성되어야 한다. 문법 변경에는 추가뿐 아니라 제거와 이름 변경과 타입 변경이 포함된다. 제거된 항목이 계속 생성되면 실행 단계에서 예외가 증가하고 유효 입력 비율이 감소한다. 따라서 생성기는 변경 내역을 반영해 더 이상 지원되지 않는 Enum이나 Feature를 생성하지 않도록 관리한다. 이름이 변경된 Feature는 신규 이름으로 치환한다. 타입이 변경된 필드나 인자는 신규 타입에 맞게 생성 규칙을 갱신한다. 입력 생성은 단발 생성이 아니라 지속 생성 방식으로 운영한다. 각 인스턴스는 목표 파일 수를 유지하도록 동작하며 목표 수보다 부족할 때만 추가로 생성한다. 이를 통해 실행 단계가 입력 부족으로 멈추는 상황을 방지한다. 또한 인스턴스

별로 생성 결과를 분리 저장하여 병렬 실행에서의 경쟁을 줄인다.

제4절. Corpus Execution

코퍼스 실행은 브라우저에 입력을 안정적으로 공급하고 실행 진행을 자동으로 순환시키는 구조로 구현하였다. 실행 모델은 iframe 기반 실행을 기본으로 사용한다. 단일 퍼저 페이지가 코퍼스 디렉터리 내의 HTML 목록을 로드하고 각 입력을 sandbox 처리된 iframe에 순차적으로 주입한다. 입력 하나가 로드되면 다음 입력으로 넘어가는 형태로 동작한다. 실행 안정성을 위해 시간 제한을 둔다. 입력 로드가 일정 시간 내 완료되지 않으면 정지나 무한 루프 가능성이 있으므로 해당 입력을 문제 입력으로 기록하고 다음 입력으로 진행하거나 크래시 후보로 분류할 수 있도록 한다. 필요 시 실행 중인 브라우저를 재시작하여 다음 입력 실행을 이어간다. 브라우저 실행은 자동 런처가 담당한다. 런처는 브라우저 프로세스의 표준 출력 로그를 읽고 비정상 종료 또는 크래시 키워드를 탐지한다. 크래시가 감지되면 HTTP Server에 크래시 이벤트를 전달하여 수집 절차를 트리거한다. 동시에 브라우저를 종료하고 재시작하여 퍼징을 지속한다. 이 방식은 크래시 탐지와 실행 지속을 분리하여 무인 운영이 가능하도록 한다. 다중 인스턴스에서는 포트별로 독립된 퍼저 페이지 URL을 사용한다. 각 인스턴스는 자신의 코퍼스만 소비하므로 입력 소비 속도와 생성 속도의 균형이 맞지 않더라도 다른 인스턴스에 영향을 주지 않는다. 또한 포트별로 실행 로그와 크래시 로그를 분리해 후속 분석의 혼선을 줄인다.

제5절. Crash Validation

크래시 검증 단계는 탐지된 이벤트가 실제 취약점 후보인지 판단하고 재현 가능한 분석 단위로 정리하는 단계이다. 검증의 목표는 오탐을 제거하고 중복 크래시를 정리하며 재현 입력을 확정하는 것이다. 크래시 이벤트는 두 경로에서 수집한다. 첫째는 브라우저 로그 기반 탐지이다. 브라우저 프로세스 또는 콘텐츠 프로세스의 비정상 종료 메시지를 기준으로 크래시를 감지한다. 둘째는 실행 페이지 기반 탐지이다. 실행 중단이나 타임아웃이 반복되는 경우를 감지하고 서버로 이벤트를 전달할 수 있도록 한다. 두 경로는 상호 보완적으로 동작하며 한쪽 탐지 실패를 다른 쪽이 보완한다.
크래시가 발생하면 크래시 시점 근처에 로드된 입력을 수집한다. 서버는 최근 요청 로그를 기준으로 크래시 직전 윈도우 내에서 접근된 HTML을 추출하고 해당 파일을 크래시 디렉터리로 복사한다. 이때 퍼저 페이지 자체와 함께 저장하여 실행 재현에 필요한 최소 구성을 확보한다. 중복 제거는 스택 시그니처와 크래시 유형을 기준으로 수행한다. 동일한 스택 패턴이 반복되는 경우 동일 버그로 분류하고 대표 입력만 유지한다. 반대로 스택이 다르거나 sanitizer 리포트가 다른 경우는 별도 후보로 유지한다. 타임아웃 계열은 환경 문제일 수 있으므로 반복 발생 여부와 특정 입력 의존성을 추가로 확인한다. 재현 검증은 정적 수집만으로 끝나지 않는다. 저장된 입력 묶음을 별도 환경에서 반복 실행하여 크래시가 재발하는지 확인한다. 메모리 오류 계열은

ASan 리포트를 기준으로 재현 여부를 판단한다. 레이스 컨디션 계열은 재현 확률이 낮을 수 있으므로 반복 횟수 기반으로 판단하고 필요 시 스케줄링 교란 훅과 시드 고정 로그를 적용해 재현 신뢰도를 높인다.

제6절. 취약점 식별

취약점 식별 단계는 검증된 크래시를 원인 분석으로 연결하고 보안 영향도를 평가하는 단계이다. 목표는 크래시를 단순 오류로 남기지 않고 취약점 유형과 근본 원인과 영향 범위를 명확히 하는 것이다. 분석은 크래시 리포트와 스택 트레이스를 기반으로 시작한다. sanitizer 리포트가 존재하는 경우 오류 유형을 우선 분류한다. Use After Free와 Out of Bounds와 Double Free와 Integer Overflow 등 메모리 오류 형태를 구분한다. 레이스 컨디션이 의심되는 경우 스레드 간 접근 순서와 락 사용 여부를 확인한다. 특정 입력이 아니라 입력 실행 순서에 의존하는 경우 누적 상태와 타이밍 의존성을 함께 고려한다. 다음으로 재현 입력을 최소화한다. 불필요한 DOM 조작이나 WebGPU 설정을 제거하면서도 동일 크래시가 유지되도록 입력을 줄인다. 입력 최소화는 원인 코드 경로를 좁히고 패치 검증을 쉽게 만든다. 레이스 계열은 최소화 과정에서 재현이 깨질 수 있으므로 최소화와 재현성 사이의 균형을 유지한다. 필요 시 로그 기반 재현 절차를 유지한 채로 최소화를 진행한다. 근본 원인 식별은 소스 코드 레벨에서 수행한다. 크래시 지점의 객체 생명주기와 참조 관계를 추적하고 해제 이후 사용 경로가 존재하는지 확인한다. 타입 혼동 가능성이 있는 경우 가드 조건과 캐스팅 경로를 확인한다. WebGPU 관련 크래시는 입력의 설정값과 내부 검증 로직의 경계 처리와 버퍼 크기 계산 등을 중점적으로 확인한다. 레이스 계열은 동일 객체에 대한 동기화 누락 또는 락 순서 문제를 중심으로 확인한다. 마지막으로 취약점 영향도를 정리한다. 동일 프로세스 내 크래시로 제한되는지 임의 읽기 쓰기 가능성이 존재하는지 원격 트리거 가능성이 있는지 샌드박스 경계와 결합 가능성이 있는지 등을 기준으로 정리한다. 취약점으로 확정된 항목은 재현 절차와 최소 재현 입력과 분석 요약과 원인 코드 위치와 영향도를 포함해 보고서 형태로 기록한다.

제시한 방법론을 통해, 본 연구로 취약점 제보 성과를 확보하였다. 총 7건의 제보를 진행하였으며 이 중 CVE가 부여된 항목은 CVE-2025-43511과 CVE-2025-10500과 CVE-2025-43401이다. CVE-2025-43511 제보 건은 1,000 USD의 보상으로 처리되었고 CVE-2025-10500 제보 건은 15,000 USD의 보상으로 처리되었다. 추가로 CVE 3건에 대한 제보를 별도로 진행하였다.

[부록]

느낀점_이동하 :

현재 학계에서 실제로 다루고 있는 문제를 직접 접하고, 그 문제를 출발점으로 탐구하고 고민하는 전 과정을 경험하면서 이전에는 접하기 어려웠던 새로운 내용들을 폭넓게 배울 수 있었다. 단순히 정답을 찾는 방식이 아니라, 문제의 배경과 맥락을 이해하고 어떤 가정과 기준 위에서 접근할지부터 정리해 나가다 보니, 같은 문제를 두고도 사람마다 전혀 다른 해결 방법과 관점이 존재한다는 사실이 특히 인상 깊었다. 누군가는 효율성과 재현 가능성을 우선에 두고 접근했고, 또 다른 누군가는 현실 적용 가능성과 리스크를 중심으로 판단했으며, 어떤 사람은 문제를 더 근본적인 구조로 재정의하면서 새로운 방향을 제시하기도 했다. 이러한 다양성을 가까이에서 확인하면서, 하나의 문제를 바라보는 시야가 넓어졌고 '좋은 해결'이란 무엇인지에 대해 더 입체적으로 생각해 보게 되었다.
무엇보다 현업자분들의 현실적인 피드백을 직접 듣고 반영하는 과정이 의미 있게 다가왔다. 이론적으로는 설득력 있어 보이던 접근이 실제 환경에서는 어떤 제약 때문에 어려울 수 있는지, 반대로 단순해 보이는 방법이 실무에서는 왜 더 강력한 선택지가 될 수 있는지 등, 현장에서 축적된 경험을 바탕으로 한 조언들은 문제를 훨씬 현실적으로 바라보게 해 주었다. 그 결과, 머릿속에서만 존재하던 개념들을 실제 상황에 맞게 재구성해 보면서 이론과 실무 사이의 간극을 스스로 메워 보는 경험을 할 수 있었다. 또한 피드백을 통해 내 논리의 빈틈이나 전제의 한계를 발견하고 보완하는 과정에서, 단순히 지식을 습득하는 것을 넘어 사고방식 자체를 점검하고 성장시키는 계기가 되었다.

느낀점_서예진 :

프로젝트 초반에는 퍼저에 대한 이해가 부족해 퍼징을 단순한 무작위 입력 실행 정도로만 인식하고 있었다. 그러나 SoK 논문과 Domato, FreeDom, SaGe 등의 기존 연구를 분석하면서, 퍼징이 입력 생성뿐만 아니라 실행 환경 구성과 커버리지 기반 피드백, 탐색 전략이 결합된 구조적인 기법임을 이해하게 되었다.

이번 프로젝트에서는 Chromium을 중심으로 브라우저 빌드 및 실행 환경을 구성하였으며, 이 과정에서 Safari, Firefox 등 서로 다른 브라우저 엔진의 빌드 구조와 개발 방식도 함께 학습하였다. 각 브라우저마다 사용하는 빌드 도구와 구조가 다르다는 점을 통해, 퍼징과 보안 분석에서는 단순한 빌드가 아니라 목적에 맞는 환경 구성이 중요하다는 점을 깨달았다.

또한 커버리지 기반 퍼징을 고려하며 렌더러 실행과 계측 빌드의 의미를 이해하게 되었고, 퍼저의 동작을 가능하게 하는 실행 환경 구성 자체가 퍼징의 핵심 요소라는 점을 인식하게 되었다. 이를 통해 브라우저 보안 연구에서 실행 환경을 구성하고 제어하는 능력이 매우 중요하다는 것을 체득할 수 있었다.

느낀점_한보윤 :

기존 fuzzer를 단순히 실행하고 결과를 관찰하는 방식과 달리, fuzzer를 직접 설계하고 구현하는 과정에서는 퍼징 기법 전반에 대한 이해도가 본질적으로 달라진다는 점을 분명히 느낄 수 있었다. 단순 사용자 관점에서는 입력 생성, 실행, 크래시 탐지와 같은 단계들이 하나의 블랙박스로 인식되기 쉽지만, 실제로 구현을 진행하면서 각 단계가 어떤 의도로 구성되고, 어떤 선택이 퍼징 효율과 탐색 범위에 직접적인 영향을 미치는지를 구체적으로 이해하게 되었다. 특히 입력 생성 전략, 실행 흐름 제어, 예외 처리 방식 등 세부 설계 요소 하나하나가 퍼저의 성능과 안정성을 좌우한다는 점을 체감할 수 있었다.

또한 퍼징 대상 시스템의 내부 구조와 동작 특성을 충분히 이해하는 것이, 단순한 구현을 넘어 체계적인 개발 설계를 위해 필수적이라는 점을 확인하였다. 대상 시스템이 어떤 입력을 처리하고, 내부적으로 어떤 컴포넌트와 실행 경로를 거치는지에 대한 이해가 부족할 경우, 생성되는 입력은 실제 의미 있는 코드 영역에 도달하지 못하거나 비효율적인 테스트로 이어질 수 있다. 반대로 내부 구조와 동작 흐름을 고려하여 퍼저를 설계했을 때에는 입력 생성 단계부터 실행 및 분석 단계까지 전체 개발 과정을 논리적으로 구성할 수 있었고, 이는 구현 과정의 오류를 줄이고 확장 가능성을 높이는 데에도 긍정적인 영향을 주었다.

이번 과정을 통해 퍼징은 단순히 도구를 활용하는 기술이 아니라, 대상 시스템에 대한 분석 역량과 설계 능력이 결합된 종합적인 보안 연구 활동임을 인식하게 되었으며, 향후 보다 복잡한 시스템이나 실제 취약점 분석 환경에서도 이러한 접근 방식이 중요한 기반이 될 것이라는 점을 명확히 느낄 수 있었다.