AEG: 자동 익스플로잇 생성

Thanassis Avgerinos, 차상길, Brent Lim Tze Hao, David Brumley
카네기 멜론 대학교, 피츠버그, 펜실베니아
{타나시스, 상킬크, 브렌트림, dbrumley}@cmu.edu

추상적인

자동 익스플로잇 생성 챌린지가 주어집니다.
프로그램이 자동으로 취약점을 찾아내고 이에 대한 익스플로잇을 생성합니다.
이 논문에서 우리는 AEG,
완전 자동 익스플로잇 생성을 위한 최초의 종단 간 시스템. AEG를 사용하여
14개의 오픈 소스 프로젝트를 분석했습니다.
16개의 제어 흐름 하이재킹 익스플로잇을 성공적으로 생성했습니다. 생성된
익스플로잇 중 2개(expect-5.43 및
htget-0.93)은 알려지지 않은 취약점에 대한 제로 데이 익스플로잇입니
다. 우리의 기여는 다음과 같습니다: 1) 우리는 방법을 보여줍니다
제어 흐름 하이재킹 공격에 대한 익스플로잇 생성은
형식 검증 문제로 모델링됨, 2) 기호 실행을 대상으로 하는 새로운 기술인 사
전 조건화된 기호 실행을 제안합니다. 3) 다음을 제시합니다.

작업 익스플로잇을 한 번 생성하기 위한 일반적인 접근 방식
버그가 발견되면 4) 취약점을 자동으로 찾고 셸을 생성하는 익스플로잇을 생
성하는 최초의 종단 간 시스템을 구축합니다.

1. 소개

제어 흐름 악용을 통해 공격자는 컴퓨터에서 임의의 코드를 실행할 수 있
습니다. 현재의 최첨단
제어 흐름 익스플로잇 생성은 인간이 생각하는 것입니다.
버그가 악용될 수 있는지 여부에 대해 매우 어렵습니다. 까지
이제 버그가 자동으로 발견되고 익스플로잇이 생성되는 자동화된 익스플로잇
생성이 수행되지 않았습니다.
실제 프로그램에 대해 실용적인 것으로 나타났습니다.

이 논문에서 우리는 새로운 기술을 개발하고
자동 익스플로잇 생성을 위한 종단 간 시스템
(AEG) 실제 프로그램. 우리의 설정에서 우리는 주어진
소스 형식의 잠재적으로 버그가 있는 프로그램. 우리의 AEG
기술은 버그를 찾고, 버그가 악용 가능한지 여부를 결정하고, 악용 가능한 경
우 작동하는 제어 흐름 하이 잭 악용 문자열을 생성합니다. 익스플로잇 문자
열은 직접
취약한 애플리케이션에 공급되어 쉘을 얻습니다. 우리
14개의 오픈 소스 프로젝트를 분석하고 성공적으로
다음을 포함한 16개의 제어 흐름 하이재킹 익스플로잇 생성
이전에 알려지지 않은 취약점에 대한 두 가지 제로 데이 익스플로잇.

우리의 자동 익스플로잇 생성 기술은
몇 가지 즉각적인 보안 영향. 첫째, 실용적인
AEG는 인지 능력을 근본적으로 변화시킵니다.
공격자의. 예를 들어, 이전에는 훈련되지 않은 공격자가 상대적으로 어렵다고
믿어왔습니다.
새로운 취약점을 찾고 제로 데이 익스플로잇을 생성합니다.
우리의 연구는 이 가정이 근거가 없음을 보여줍니다. 공격자의 능력을 이해하
면 다음을 알 수 있습니다.
방어가 적절하다. 둘째, 실용적인 AEG는 방어에 적용됩니다. 예를 들어, 자동
서명
생성 알고리즘은 일련의 익스플로잇을 입력으로 사용하고,
후속 익스플로잇 및 익스플로잇 변종을 인식하는 IDS 서명(입력 필터라고도
함)을 출력합니다 [3, 8, 9].
자동화된 익스플로잇 생성은 서명에 제공될 수 있습니다.
요구하지 않고 방어자에 의한 생성 알고리즘
실제 공격.

도전. 우리가 해결해야 할 몇 가지 과제가 있습니다
AEG를 실용적으로 만들기 위해:

A. 소스코드 분석만으로는 부족하고 충분하다. 소스 코드 분석은 소스 코
드 수준 추상화와 관련하여 오류가 발견되기 때문에 잠재적인 버그가 악용될
수 있는지 여부를 보고하기에 충분하지 않습니다. 그러나 제어 흐름 악용은

스택 프레임과 같은 바이너리 및 런타임 수준 세부 정보,
메모리 주소, 변수 배치 및 할당,
소스 코드에서 사용할 수 없는 기타 많은 세부 정보
수준. 예를 들어 다음 코드 발췌를 고려하십시오.

```
char src [ 1 2 ],          dst [ 1 0 ];
strncpy ( d st      ,   src   ,   크기의 ( src ) ) ;
```

이 예에서는 고전적인 버퍼 오버플로가 있습니다.
더 큰 버퍼(12바이트)가 더 작은 버퍼에 복사되는 경우
버퍼(10바이트). 그러한 진술이 분명하지만
잘못된 [1] 소스에서 버그로 보고됩니다.

1기술적으로 C99 표준은 프로그램이
이 시점에서 정의된 행동.

코드 수준에서 실제로 이 버그는 악용될 가능성이 없습니다. 최신 컴파
일러는 선언된 버퍼를 페이지 정렬하여 두 데이터 구조 모두

16바이트. 대상 버퍼는 16바이트이므로,
12바이트 사본은 문제가 되지 않으며 버그가
악용할 수 없습니다.

소스 코드 분석은 불충분하지만 바이너리 레벨 분석은 확장이 불가
능합니다. 소스 코드에는 변수, 버퍼, 함수 및 사용자 구성 유형과 같
은 추상화가 있어 자동화된 추론을 더 쉽고 확장 가능하게 만듭니다.
그러한 추상화는 현재 존재하지 않습니다.

바이너리 레벨; 스택 프레임, 레지스터, goto만 있습니다.
및 전역적으로 주소가 지정된 메모리 영역.

우리의 접근 방식에서는 소스 코드 수준 분석을 결합하여 버그 및
바이너리 찾기의 확장성을 향상시킵니다.
프로그램을 악용하기 위한 런타임 정보. 최고에게
우리의 지식, 우리는 분석을 결합한 최초의
이 두 가지 매우 다른 코드 추상화 수준에서.

B. 무한한 것 중에서 악용할 수 있는 길 찾기
가능한 경로의 수. AEG를 위한 당사의 기술
프로그램 경로를 탐색하고 각각이

경로를 악용할 수 있습니다. 프로그램에는 루프가 있습니다.
잠재적으로 무한한 수를 의미합니다.
경로. 그러나 모든 경로가 동일한 가능성이 있는 것은 아닙니다.
착취 가능. 어떤 경로를 먼저 확인해야 합니까?

우리의 주요 초점은 악용 가능한 버그를 감지하는 것입니다. 우리의
결과는 (§ 8) 기존의 최첨단 솔루션이
보안에 중요한 버그를 감지하기에 불충분한 것으로 판명됨
실제 프로그램에서.

경로 선택 문제를 해결하기 위해 AEG에서 두 가지 새로운 기여를
개발했습니다. 첫째, 우리는
전제된 상징적 실행, 소설 개발
가능성이 더 높은 경로를 대상으로 하는 기술
착취 가능. 예를 들어 한 가지 선택은
최대 입력 길이가 있는 경로 또는 관련 경로
HTTP GET 요청에. 전제된 상징적인 동안
실행은 일부 경로를 제거하지만 여전히 먼저 탐색해야 하는 경로를 지정
해야 합니다. 주소로
이 챌린지에서 우선 순위 대기열 경로를 개발했습니다.
휴리스틱을 사용하여 선택하는 우선 순위 지정 기술
악용 가능성이 더 높은 경로가 먼저입니다. 예를 들어, 우리는
프로그래머가 경로를 따라 실수(꼭 악용할 필요는 없음)를 하면 다음
을 수행하는 것이 합리적입니다.
경로에 대한 추가 탐색을 우선시합니다.
결국 악용 가능한 상태로 이어질 가능성이 있습니다.

C. 종단 간 시스템. 실제 프로그램에서 AEG를 위한 최초의 실용적
인 end-to-end 시스템을 제공합니다.
종단 간 시스템에는
엄청난 수의 과학적 질문(예: 이진법)
프로그램 분석 및 효율적인 공식 검증, 그러나
또한 엄청난 수의 엔지니어링 문제. 우리의
AEG 구현은 소스 코드 프로그램을 분석하고 기호 실행 공식을 생성하
고 이를 해결하고 이진 분석을 수행하는 단일 명령줄입니다.

바이너리 레벨 런타임 제약 조건 및 형식을 생성합니다.
취약한 프로그램에 직접 공급할 수 있는 실제 익스플로잇 문자열로 출
력합니다. 종단 간 시스템을 시연하는 비디오는 온라인에서 볼 수 있습
니다 [1].

범위. 이 백서에서 우리는 익스플로잇을 강력하게 만듭니다.
지역 환경 변화에 대항하여 우리의 목표는
일반적인 보안 방어에 대해 익스플로잇을 강력하게 만들고,
주소 공간 무작위화 [25] 및 w ⊕ x와 같은
메모리 페이지(예: Windows DEP). 이 작품에서 우리는
항상 소스 코드가 필요합니다. 바이너리 전용의 AEG가 남음
앞으로의 작업으로.

2 AEG 개요

이 섹션에서는 AEG가 단계별로 작동하는 방식을 설명합니다.
버그 발견 및 익스플로잇의 전 과정을 통해
실제 사례에 대한 세대. 대상 응용 프로그램은 다음의 setuid 루트
iwconfig 유틸리티입니다.
Wireless Tools 패키지(버전 26), 프로그램
약 3400줄의 C 소스 코드로 구성됩니다.

AEG가 분석을 시작하기 전에 두 가지 필수 전처리 단계가 있습
니다. 1) 다음을 사용하여 프로젝트를 빌드합니다.
바이너리를 생성하기 위한 GNU C 컴파일러(GCC)
2) LLVM [17] 컴파일러 로 악용하기를 원합니다.
우리의 버그 찾기 인프라가 생성하는 바이트코드
분석에 사용합니다. 빌드 후 도구인 AEG를 실행합니다.
1분 이내에 제어 흐름 하이재킹 익스플로잇을 얻습니다.
초. iwconfig에 익스플로잇 문자열 제공
바이너리는 첫 번째 인수로 루트 셸이 됩니다. 우리
데모 비디오를 온라인에 게시했습니다 [1].

그림 1 은 다음과 관련된 코드 스니펫을 보여줍니다.
생성된 익스플로잇. iwconfig에는 고전적인 strcpy가 있습니다.
정보 가져오기 기능의 버퍼 오버플로 취약성
(15행), AEG가 자동으로 탐지하고 악용하는
1초 미만. 이를 위해 우리 시스템은
다음 분석 단계:

1. AEG는 소스 코드 수준에서 버그를 검색합니다.
   실행 경로를 탐색하여 구체적으로, AEG
   기호 인수를 사용하여 iwconfig를 실행합니다.
   (argv)를 입력 소스로 사용합니다. AEG는 파일, 인수 등과 같
   은 다양한 입력 소스를 고려합니다.
   기본적으로.
2. 메인 → 정보출력 경로를 따라 이동 후
   → 정보를 얻으면 AEG가 15행에 도달하여 변수의 범위를 벗어
   난 메모리 오류를 감지합니다.
   ifr.ifr 이름. AEG는 현재 경로 제약을 해결하고 감지된 버그를
   트리거하는 구체적인 입력을 생성합니다. 예를 들어 첫 번째 인수
   는 다음과 같아야 합니다.

```
int main ( int argc 2 int skfd ; 3 if      , char ** argv ) {
( argc == 2 )                          /* 일반 원시 socketdesc .                    */

4 인쇄 정보(skfd 5 ...   _              ,  인수 [1], NULL, 0) ;

staticint 인쇄 정보 ( int skfd {     _                  ,  char * if 이름        ,  char * 인수 [ ] ,          인트 카운트 )

      구조 무선 정보            _         정보 ;
int 9 rc =                                 rc ;
getinfo(skfd_10 ..      _           ,  만약 이름  ,  정보) ;

staticint getinfo ( int skfd ) {   _              ,  char * if 이름        ,  struct wirelessinfo * 정보     _

      struct iw req wrq ;
      if ( iwgetext ( skfdif 이름 , SIOCGIWNAME , &wrq ) < 0 ) {
            구조체 ifreqifr ;
            strcpy(ifrifrname   ·   _      ,  이면 e) ; /* 버퍼 오버플로 */
. . .
```

그림 1: Wireless Tools' iwconfig의 코드 스니펫.

그림 2: 메모리 다이어그램

```
00000000 02 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 |............|
00000010 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 |............|
00000020 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 |............|
00000030 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 |............|
00000040 01 01 01 01 70 f3 ff bf 31 c0 50 68 2f 2f 73 68 |...p...1.Ph//sh|
00000050 68 2f 62 69 6e 89 e3 50 53 89 e1 31 d2 b0 0b cd |h/bin..PS..1....|
00000060 80 01 01 01 00                                   |.....|
```

그림 3: AEG에서 생성된 iwconfig 익스플로잇.

32바이트 이상이어야 합니다.

3. AEG는 iwconfig에서 동적 분석을 수행합니다. 2단계에서 생성된 구체적인 입력을 사용하여 바이너리. 메모리에 대한 런타임 정보를 추출합니다. 오버플로된 버퍼의 주소와 같은 레이아웃 (ifr.ifr 이름) 및 취약한 기능의 반환 주소 주소(정보 가져오기).

4. AEG는 생성된 런타임 정보를 사용하여 익스플로잇을 설명하는 제약 조건을 생성합니다. 이전 단계에서: 1) 취약한 버퍼 (ifr.ifr 이름)에는 쉘코드가 포함되어야 하며, 2) 덮어쓴 반환 주소는 다음을 포함해야 합니다. 런타임에서 사용할 수 있는 셸 코드의 주소입니다.

   다음으로 AEG는 생성된 제약 조건을 경로 제약 조건 및 쿼리 제약 조건 솔버 만족스러운 답변.

5. 만족스러운 답변은 익스플로잇 문자열을 제공합니다. 그림 3에 나와 있습니다. 마지막으로 AEG는 프로그램을 실행합니다. 생성된 익스플로잇으로 작동하는지 확인합니다. 즉, 쉘을 생성합니다. 제약 조건을 해결할 수 없는 경우 AEG는 프로그램 검색을 재개합니다. 다음 잠재적인 취약점.

도전. 위의 연습은 AEG가 해결해야 하는 여러 문제를 보여줍니다.

• 상태 공간 폭발 문제(단계 1-2). 잠재적으로 무한한 수의 경로가 있습니다. AEG는 악용 가능한 경로가 될 때까지 탐색해야 합니다. 감지됩니다. AEG는 사전 조건부 기호를 사용합니다. 악용 가능한 경로를 대상으로 실행(§ 5.2 참조) .

• 경로 선택 문제(1-2단계). 중에서 무한한 수의 경로, AEG는 선택해야 합니다. 어떤 경로를 먼저 탐색해야 하는지. 이를 위해 AEG 경로 우선 순위 지정 기술을 사용합니다(§ 5.3 참조).

• 환경 모델링 문제(1-3단계). 실제 응용 프로그램은 다음과 집중적으로 상호 작용합니다. 기본 환경. 정확한 AEG가 모델링해야 하는 그러한 프로그램에 대한 분석 명령줄을 포함한 환경 IO 동작 인수, 파일 및 네트워크 패킷(§ 5.4 참조).

• 혼합 분석 챌린지(1-4단계). AEG 바이너리 및 소스 레벨 분석을 혼합하여 수행 할 수 있는 것보다 더 큰 프로그램으로 확장하기 위해 바이너리 전용 접근 방식으로 처리됩니다. 결합 이처럼 근본적으로 다른 추상화 수준에 대한 분석 결과는 소유(6.2절 참조).

• 익스플로잇 검증 문제(5단계). 마지막, AEG는 생성된 익스플로잇이 주어진 시스템에 대한 작업 익스플로잇(§ 6.3 참조).

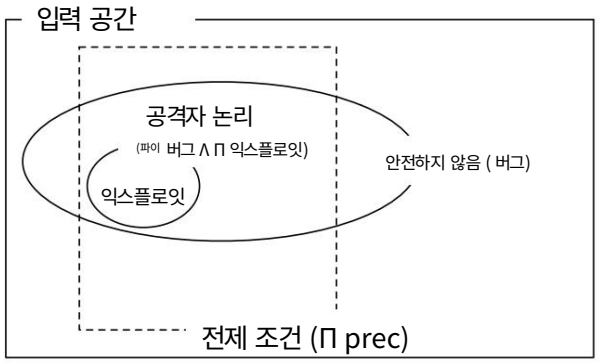

그림 4: 입력 공간 다이어그램은 안전하지 않은 입력과 악용 간의 관계를 보여줍니다. 사전 조건화된 기호 실행은 범위를 좁힙니다.

전제 조건을 만족하는 입력에 대한 검색 공간
(Πprec).

3 AEG 챌린지

그 핵심은 자동 익스플로잇 생성(AEG)
문제는 다음과 같은 프로그램 입력을 찾는 문제입니다.
결과적으로 원하는 악용 실행 상태가 됩니다. 이 섹션에서는 AEG 챌린지가 어떻게 표현되는지 보여줍니다.
공식적인 검증 문제로 제안할 뿐만 아니라
AEG를 허용하는 새로운 기호 실행 기술
이전 기술보다 더 큰 프로그램으로 확장합니다. 처럼
결과적으로 이 공식은: 1) 형식 검증을 가능하게 합니다.
익스플로잇을 생성하는 기술을 사용하고 2) AEG가 형식 검증의 발전으로부터 직접적인 이익을 얻을 수 있습니다.

3.1 문제 정의

이 백서에서는 제어 흐름 생성에 중점을 둡니다.
두 가지를 직관적으로 수행하는 하이재킹 익스플로잇 입력
것들. 첫째, 익스플로잇은 프로그램 안전을 위반해야 하며,
예를 들어, 프로그램이 아웃 오브 바운드 메모리에 쓰도록 합니다.
둘째, 익스플로잇은 제어 흐름을 다음으로 리디렉션해야 합니다.
공격자의 논리, 예를 들어 쉘 코드 주입 실행,
return-to-libc 공격 등을 수행합니다.

높은 수준에서 우리의 접근 방식은 프로그램이 악용 가능한지 확인하는 프로그램 확인 기술을 사용합니다(프로그램이 안전한지 확인하는 기존 확인과 반대). 악용된 상태는 버그가 있는 실행 경로 술어 Πbug 와 제어 흐름 하이재킹 익스플로잇 술어 Πexploit 의 두 가지 부울 술어로 특성화 되어 제어 하이재킹 및

코드 인젝션 공격 Πbug 술어는 프로그램이 프로그램의 의미를 위반할 때 충족됩니다 .
안전. 그러나 단순히 안전을 위반하는 것은 일반적으로
부족한. 또한 Πexploit 는 조건을 캡처합니다.
프로그램 제어를 가로채기 위해 필요합니다.

우리 접근 방식의 익스플로잇은 다음을 충족하는 입력 ε입니다.
부울 방정식:

$$\Pi bug(\varepsilon) \wedge \Pi exploit(\varepsilon) = 참 (1)$$

이 공식을 사용하여 AEG의 역학은 다음과 같습니다.
실행의 각 단계에서 Equation 1
만족합니다. 만족스러운 대답은 건설에 의해,
제어 흐름 하이재킹 익스플로잇. 우리는 아래에서 이 두 가지 전제에 대해 더 자세히 논의합니다.

안전하지 않은 경로 술어 버그. Πbug 는 다음을 나타냅니다.
안전을 위반하는 실행의 경로 술어
속성 ϕ. 우리의 구현에서는 범위를 벗어난 쓰기, 안전하지 않은 형식 문자열 등을 확인하는 것과 같이 C 프로그램에 대해 널리 알려진 안전 속성을 사용합니다.
안전하지 않은 경로 술어 Πbug 는 입력 공간을 분할합니다.
술어(안전하지 않음)를 충족하는 입력 및 입력
(안전)하지 않습니다. 경로 술어로 충분하지만
AEG에서는 소스 코드 수준에서 버그를 설명합니다.
매우 구체적으로 설명하기에는 필요하지만 충분하지 않습니다.
우리가 취하고자 하는 행동, 예를 들어 쉘코드 실행.

Exploit 술어 Πexploit. 익스플로잇 술어
공격자가 수행하려는 공격자의 논리를 지정합니다.
eip 탈취 후. 예를 들어 공격자가
프로그램을 충돌시키려는 경우 술어는 "제어권을 얻은 후 eip를 잘못된 주소로 설정"처럼 간단할 수 있습니다. 우리의 실험에서 공격자의 목표는
껍데기. 따라서 Πexploit 는 쉘코드가
메모리에 잘 구성되어 있으며 해당 eip가 전송됩니다.
그것에 대한 통제. 익스플로잇 술어의 결합
(Πexploit) 는 최종 솔루션에 대한 제약을 유발합니다. 만약에
최종 제약 조건(방정식 1)이 충족되지 않으면
버그를 악용할 수 없는 것으로 간주합니다 (§ 6.2).

3.2 사전 조건이 지정된 기호 실행으로 확장

우리의 공식은 우리가 공식 검증을 사용할 수 있도록 합니다.
익스플로잇을 생성하는 기술. 이는 AEG에 잘못된 검증 수단을 사용할 수 있지만, 모델 검사, 가장 약한 전제 조건,

불행히도 상징적 검증을 전달합니다.
소규모 프로그램으로 확장합니다. 예를 들어 KLEE는 최첨단 순방향 기호 실행 엔진 [5]이지만
연습은 /bin/ls와 같은 작은 프로그램으로 제한됩니다.
우리의 실험에서 KLEE는 다음 중 하나만 찾을 수 있었습니다.
우리가 악용한 버그(§ 8).

우리는 확장성이 다음으로 제한되는 한 가지 이유를 관찰합니다.
기존 검증 기법은 전체 프로그램 상태를 고려하여 버그가 없음을 증명하는 것이다.
우주. 예를 들어, KLEE가 다음을 위한 프로그램을 탐색할 때
버퍼 오버플로는 가능한 모든 입력 길이를 고려합니다.
최대 크기, 즉 길이 0의 입력, 길이 1의 입력 등. 우리는 우리가 할 수 있음을 관찰

다음만 포함하도록 상태 공간을 제한하여 AEG 크기 조정
예를 들어 다음을 고려하여 악용될 가능성이 있는 상태
덮어쓰기에 필요한 최소 길이의 입력만
모든 버퍼. 최소 길이를 미리 결정하기 위해 저비용 분석을 수행하여 이를 달성합니다.
이것은 (더 비싼) 검증 단계에서 상태 공간 검색을 제거할 수 있게 해줍니다.

우리는 사전 조건된 기호 실행을 다음과 같이 제안합니다.
일부를 잘라내는 검증 기법
흥미롭지 않은 상태 공간. 조건부 기호 실행은 상태를 점진적으로 탐색한다는 점에서 순방향 기호 실행 [16, 23] 과 유사합니다.

버그를 찾는 공간. 그러나 사전 조건부 기호
실행은 추가 Πprec 매개변수를 받습니다. 사전 조건화된 기호 실행은
Πprec 를 충족하는 프로그램 분기로만 내려가며 순 효과가 있습니다.

충족되지 않은 분기의 후속 단계는 제거됩니다.
AEG에서는 사전 조건부 기호 execution
스플로잇 가능성이 있는 경우에만 탐색을 제한하는[2] ... 를 사용합니다. 익

상태 공간의 영역. 예를 들어, 버퍼 오버플로의 경우 Πprec 는 경량 프로그램 분석을 통해 지정됩니다 .
오버플로할 최소 크기의 입력을 결정합니다.
완충기.

그림 4 는 차이점을 시각적으로 보여줍니다. 일반적인 검증은 안전하지 않고 Πbug 를 만족시키는 인풋을 찾는 것을 목표로 전체 상자로 표시되는 전체 입력 상태 공간을 탐색합니다 . AEG에서 우리는

안전하지 않은 상태의 하위 집합에만 관심이 있습니다.
Πexploit ∧ 로 표시된 원으로 표시되는 악용 가능
익스플로잇. 직관은 전제된 상징적
실행은 검색된 공간을 더 작은 상자로 제한합니다.

논리적으로, Πprec 가 익스플로잇 가능성 조건보다 덜 제한적일 때 가능한 모든 익스플로잇을 찾을 수 있습니다.

$$\Pi bug(x) \wedge \Pi exploit(x) \Rightarrow \Pi prec(x)$$

실제로 이 제한을 완화하여 범위를 좁힐 수 있습니다.
검색 공간을 훨씬 더 멀리,
일부 익스플로잇이 누락되었습니다. 우리는 여러 가능성을 탐구합니다
§ 5.2에서 그 효과를 경험적으로 평가합니다.
§ 8.

4 우리의 접근 방식

이 섹션에서는 자동 익스플로잇 생성을 위한 시스템인 AEG의 구성 요소에 대한 개요를 제공합니다. 그림 5 는 AEG에서 익스플로잇을 생성하는 전체 흐름을 보여줍니다. AEG 찰에 대한 우리의 접근

2참고 사전 조건 순방향 기호 실행은 다음과 다릅니다.
가장 약한 전제 조건. 가장 약한 전제 조건은 정적으로 계산
원하는 사후 조건을 달성하기 위한 가장 약한 사전 조건. 여기서 우리는
가지치기를 위한 필요하지 않은 가장 약한 전제 조건을 동적으로 확인합니다.

lenge는 PRE-PROCESS, SRC ANALYSIS, BUG-FIND, DBA, EXPLOIT-GEN 및[3]
확인하다.

전처리 : src → (Bgcc, Bllvm).
AEG는 2입력 단일 출력 시스템입니다.
대상 바이너리 및 LLVM 바이트 코드 제공
AEG가 성공하면 동일한 프로그램의
주어진 바이너리에 대한 작업 익스플로잇을 되돌립니다.
프로그램 분석 부분이 시작되기 전에,
필요한 수동 전처리 단계: 소스
프로그램(src)은 1) 바이너리 Bgcc 로 컴파일됩니다.
AEG는 작업 익스플로잇과 2) LLVM 바이트코드 파일 Bllvm 을 생성하려고 시도합니다.
버그 찾기 인프라에서 사용됩니다.

SRC- 분석: Bllvm → 최대.
AEG는 소스 코드를 분석하여 기호 데이터 max의 최대 크기를 생성해야 합니다.
프로그램에 제공됩니다. AEG는 다음으로 최대값을 결정합니다.
가장 큰 정적으로 할당된 버퍼 검색
대상 프로그램의. AEG는 다음과 같은 휴리스틱을 사용합니다.
최대값은 최대값보다 10% 이상 커야 합니다.
버퍼 크기.

BUG-FIND (Bllvm, φ, max) → (Πbug, V).
BUG-FIND 는 LLVM 바이트코드 Bllvm 과
안전 속성 φ, 튜플 hΠbug,Vi 출력
탐지된 각 취약점에 대해 버그 는 다음을 포함합니다.
경로 술어, 즉 안전 속성 φ의 위반까지 모든 경로 제약 조건의 결합.

V에는 개체 이름과 같이 탐지된 취약점에 대한 소스 수준 정보가 포함되어 있습니다.
덮어쓰기되고 취약한 기능. 에게
경로 제약 조건을 생성하고 AEG는 기호를 사용합니다.
집행자. 기호 실행자는 버그를 보고합니다.
φ 속성을 위반할 때마다 AEG. AEG는 악용 가능한 버그를 감지하기 위해 몇 가지 새로운 버그 찾기 기술을 활용합니다(§ 5 참조).

DBA: (Bgcc, (Πbug, V)) → R.
DBA 는 구체적인 버그 입력으로 tar get 바이너리 Bgcc 에 대해 동적 바이너리 분석을 수행하고 런타임 정보 R을 추출합니다. 구체적인 입력
경로 제약 Πbug를 해결하여 생성됩니다.
취약한 기능을 실행하는 동안(지정된
소스 코드 수준에서 V), DBA는
저수준 런타임 정보(R)를 추출하는 바이너리,
스택에 있는 취약한 버퍼의 주소와 같은
취약한 함수의 반환 광고 드레스 주소와 바로 직전의 스택 메모리 내용
취약점이 발동됩니다. DBA는 다음을 보장해야 합니다.
이 단계에서 수집된 모든 데이터는 정확합니다.

3Dynamic Binary Analysis

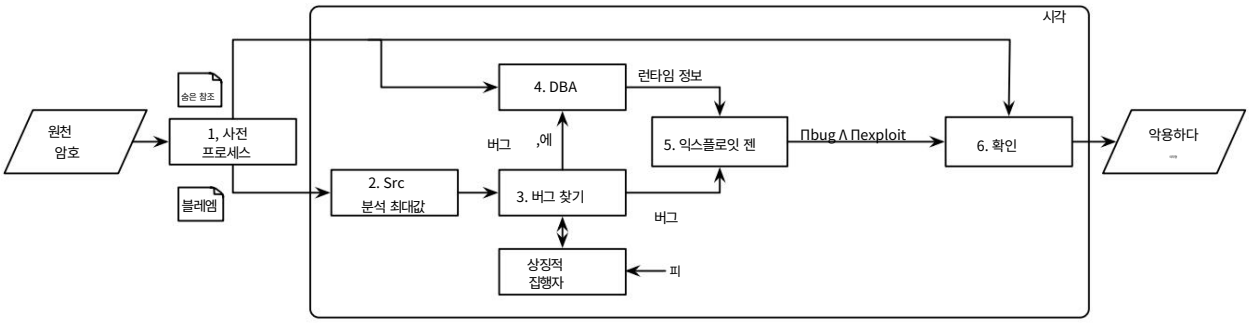

그림 5: AEG 설계.

AEG는 작업을 생성하기 위해 AEG에 의존하기 때문에
익스플로잇(§ 6.1 참조).

EXPLOIT -GEN: (Πbug, R) → Πbug ∧Πexploit.

EXPLOIT -GEN 은 버그의 경로 술어 (Πbug) 와 런타임 정보(R)가
포함된 튜플을 수신하고,
제어 흐름 하이재킹에 대한 공식을 구성합니다.
악용하다. 출력 공식에는 제약 조건이 포함됩니다.
1) 가능한 프로그램 카운터 포인트
사용자가 지정한 위치, 2) 위치
셸코드 포함(공격자의 논리 지정
익스플로잇). 결과적인 익스플로잇 공식은 두 술어의 결합입니다
(6.2절 참조).

확인 : (Bgcc, Πbug ∧Πexploit) → {ε, ⊥}.

VERIFY 는 대상 바이너리 실행 파일 Bgcc 를 가져옵니다.
및 익스플로잇 공식 Πbug ∧ Πexploit, 그리고 반환
만족스러운 답변이 있는 경우에만 ε을 익스플로잇합니다.
그렇지 않으면 ⊥를 반환합니다. 우리의 구현에서,
AEG는 VERIFY에서 추가 단계를 수행합니다. 실행
ε을 입력으로 사용 하는 이진 Bgcc 이고 다음 을 확인합니다.
적대적 목표가 충족되었는지 여부, 즉 다음과 같은 경우
프로그램은 쉘을 생성합니다(6.3절 참조).

알고리즘 1 은 해결을 위한 고급 알고리즘을 보여줍니다.
AEG 챌린지.

알고리즘 1: AEG 익스플로잇 생성 알고리즘

입력 : src: 프로그램의 소스 코드
출력: {ε, ⊥}: 작동 중인 익스플로잇 또는 ⊥

```
1 (Bgcc, Bllvm) = 전처리(src);
최대 2개 = Src-분석(Bllvm);
3 동안 (Πbug,V ) = Bug-Find(Bllvm, ϕ, max) do
4 R = DBA(Bgcc, (Πbug, V ));
5       Πbug ∧Πexploit = Exploit-Gen(Πbug, R);
6       ε = 검증(Bgcc, Πbug ∧Πexploit);
7       ε 6= ⊥ 이면
8           리턴 ε;
9 반환 ⊥;
```

5 BUG-FIND: 익스플로잇 생성을 위한 프로그램 분석

BUG-FIND 는 대상 프로그램을 입력으로 사용합니다.
LLVM 바이트코드 형식, 버그 및 각 버그 확인
나머지 익스플로잇 생성 단계를 시도했습니다.
성공할 때까지. BUG-FIND 는 기호로 버그를 찾습니다.
프로그램 상태를 탐색하는 프로그램 실행
한 번에 한 경로를 띄웁니다. 그러나 잠재적으로 탐색할 수 있는 경로는 무한합니다. AEG 광고는 두 가지 새로운 알고리즘으로 이 문제를 해결합니다. 첫 번째,
우리는 다음으로 간주되는 경로를 제한하는 사전 조건화된 기호 실행이라는 새로운 기술을 제시합니다.
악용 가능한 버그를 포함할 가능성이 가장 높은 것들.
둘째, 새로운 경로 우선순위화 휴리스틱을 제안합니다.
미리 정의된 기호 실행으로 먼저 탐색할 경로를 선택합니다.

5.1 버그에 대한 전통적인 상징적 실행 발견

높은 수준에서 기호 실행은 개념적으로
제공하는 대신 새로운 기호 변수를 제공한다는 점을 제외하고는 일반적인 구체적인 실행과 유사합니다.
입력에 대한 구체적인 값. 프로그램이 실행되면서 각각의
기호 실행 단계는 다음과 같은 식을 구축합니다.
프로그램 용어를 기호 입력으로 대체합니다.
프로그램 브랜치에서 인터프리터는 개념적으로 "forks
off" 두 명의 통역사, 진정한 지점 경비원을
트루 브랜치 인터프리터를 위한 조건, 그리고 유사하게
거짓 가지를 위해. 인터프리터가 실행할 때 부과되는 조건을 주어진 경로를 실행하기 위한 경로 술어라고 합니다. 포크 후 인터프리터는 다음을 확인합니다.

결정을 쿼리하여 경로 술어를 만족할 수 있는 경우
절차. 그렇지 않으면 어떤 입력으로도 경로를 실현할 수 없습니다.
그래서 인터프리터가 종료됩니다. 경로 술어가 충족될 수 있으면 인터프리터는 계속 실행하고 탐색합니다.
프로그램 상태 공간 보다 정확한 의미론은
Schwartz et al. [23].

기호 실행은 다음을 추가하여 버그를 찾는 데 사용됩니다.

ϕ를 사용한 안전 점검. 예를 들어 포인터를 사용하여 버퍼에 액세스할 때마
다 인터프리터는 포인터가 버퍼 범위 내에 있는지 확인해야 합니다. 그만큼

bounds-check는 안전을 의미하는 true를 반환합니다.
재산 보유 또는 거짓, 위반이 있음을 의미,
따라서 버그. 안전 위반이 발견될 때마다,
기호 실행이 중지되고 현재 버그가 있는 경로
술어 (Πbug) 가 보고됩니다.

5.2 전제된 기호 실행

기호 실행(및
다른 검증 기술)이 상태를 관리하고 있습니다.
우주 폭발 문제. 상징적 실행 이후
모든 지점에서 새로운 인터프리터를 분기합니다.
통역사의 수는 기하급수적으로 증가합니다.
가지.

우리는 사전 조건된 기호 실행을 다음과 같이 제안합니다.
기호 실행을 대상으로 하는 새로운 방법
입력 상태 공간의 특정 하위 집합(그림 4 참조). 상태 공간 부분집합은 다음
과 같이 결정됩니다.
전제 조건 술어 (Πprec); Πprec 를 만족하지 않는 입력은 탐색되지 않습니
다. 사전 조건된 기호 실행에 대한 직관은 다음과 같이 좁힐 수 있다는 것입
니다.
공격 가능성 조건을 전제 조건으로 지정하여 탐색 중인 상태 공간 아래로, 예
를 들어 모든 기호 입력은 트리거할 최대 크기를 가져야 합니다.

버퍼 오버플로 버그. 사전 정의된 기호 실행의 주요 이점은 간단합니다. 크기
를 제한함으로써
기호 실행이 시작되기 전에 입력 상태 공간 중 프로그램 경로를 잘라내어 탐
색할 수 있습니다.
대상 프로그램을 보다 효율적으로

전제 조건은 무작위로 선택할 수 없습니다.
전제 조건이 너무 구체적이면 악용을 감지하지 않습니다(악용 가능성이

전제 조건); 너무 일반적이면 거의 전체 상태 공간을 탐색해야 합니다. 따라
서 전제조건
익스플로잇 간의 공통 특성을 설명해야 함
(가능한 한 많이 캡처하기 위해) 동시에
악용할 수 없는 상당 부분을 제거해야 합니다.
입력.

사전 조건이 지정된 기호 실행은 사전 조건 제약 조건을 추가하여 사전
조건을 적용합니다.
초기화 중 경로 술어. 제약 조건 추가
기호 실행 중에 분기 지점에서 수행할 더 많은 검사가 있기 때문에 이상하게
보일 수 있습니다. 그러나 상태 공간의 축소는 다음과 같이 부과됩니다.

전제 조건 제약 - 분기 지점에서 의사 결정 절차 오버헤드보다 큽니다. 지점에
대한 전제 조건이 충족되지 않으면 더 이상 수행하지 않습니다.

확인하고 통역사를 중단하지 마십시오.
나뭇가지. 우리는 악용 가능한 것에만 초점을 맞추는 반면
경로, 전체 기술이 더 일반적으로 적용됩니다.

```
1 int 처리 입력 ( 문자 입력 [ 4 2 ] )
2         문자 버퍼 [20] ;
삼         w hil e (입력 [ i ] != ' \0 ' )
4                 buf [ 나는 ++ ] = 입력 [ 나는 ] ;
```

그림 6: 촘촘한 기호 루프. 일반적인 패턴
대부분의 버퍼 오버플로에 대해.

가 되었다.

사전 조건이 지정된 기호 실행의 이점은 예제를 통해 가장 잘 설명됩니다.
고려하다
그림 6에 표시된 프로그램. 입력이
버퍼는 42개의 기호 바이트를 포함합니다. 4-5행은 다음을 나타냅니다.
strcpy와 동일한 엄격한 기호 루프
결국 각각 다른 경로 술어를 갖는 전통적인 기호 실행을 사용하는 42개의 다
른 인터프리터를 생성합니다. 첫 번째 인터프리터가 실행되지 않습니다 .

루프는 (input[0] = 0), 두 번째
인터프리터는 루프를 한 번 실행하고 다음을 가정합니다.
(입력[0] 6= 0)∧(입력[1] = 0) 등. 따라서 각각의
경로 술어는 다음에 대한 다른 조건을 설명합니다.
기호 입력 버퍼의 문자열 길이입니다. [4]

사전 조건이 지정된 기호 실행은 검사를 피합니다.
버퍼 오버플로로 이어지지 않는 루프 반복
길이 전제 조건을 부과하여:

$$패 = \forall^{나는}_{나는=0} (입력[i] \; 6= 0) \wedge (입력[n] = 0)$$

이 술어는 경로 술어(Π)에 추가됩니다.
프로그램의 상징적 실행을 시작하기 전에,
따라서 전제 조건을 충족하지 않는 경로를 제거합니다. 이전 예(그림 6)에서
실행자는
도달할 때마다 다음 검사를 수행합니다.
루프 분기점:

거짓 분기: Π∧L ⇒ input[i] = 0, 제거된 ∀i < n
실제 분기: Π∧L ⇒ 입력[i] 6= 0, 만족스러운 ∀i < n

두 가지 검사 모두 매우 빠르게 수행됩니다.
분기 조건의 (또는 무효)는 전제 조건 제약 L(in

사실 이 특정 예에서는 솔버가 필요하지 않습니다.
쿼리에 의해 유효성 또는 무효가 결정될 수 있기 때문에
우리의 가정 집합 Π ∧ L)을 통한 간단한 반복.
따라서 길이 전제 조건을 적용하면
단일 인터프리터가 전체 루프를 탐색합니다. 나머지에서
섹션에서 우리는 다른
검색 공간을 줄이기 위한 전제 조건 유형.

4문자열의 길이 전제 조건은 null을 기반으로 생성됩니다.
모든 문자열이 모두 null로 종료되기 때문입니다.

5.2.1 전제조건

AEG에서는 4가지 서로 다른
효율적인 익스플로잇 생성을 위한 전제 조건:

없음 전제 조건이 없으며 상태 공간은 다음과 같습니다.
정상적으로 탐색되었습니다.
알려진 길이 전제 조건은 입력이
이전 예에서와 같이 알려진 최대 길이. 정적 분석을 사용하여 이 전
제 조건을 자동으로 결정합니다.

알려진 접두사 전제 조건은 기호가
puts에는 알려진 접두사가 있습니다.
Concolic 실행 Concolic 실행 [24] 은 다음과 같습니다.
전제 조건이 지정된 경우 사전 조건이 지정된 기호 실행의 특정 형태
로 간주됩니다.
예제 입력에 의해 실현되는 단일 프로그램 경로에 의해. 예를 들어,
우리는 이미
프로그램을 충돌시키는 입력, 그리고 우리는 그것을
실행된 경로가 다음과 같은지 확인하기 위한 전제 조건
착취 가능.

위의 전제 조건은 다양한 양의
정적 분석 또는 사용자 입력. 다음에서 우리는 더
이러한 전제 조건에 대해 논의하고 사전 조건화된 기호 실행이 제공하는 상
태 공간의 감소를 설명합니다. 전제 조건의 효과 요약

분기에 대한 것은 그림 7에 나와 있습니다.

없음. 사전 조건화된 기호 실행은 표준 기호 실행과 동일합니다. 입력 전제
조건은 참(전체 상태 공간)입니다. 입력 공간: S용

기호 입력 바이트의 경우 입력 공간의 크기는 256S 입니다.
입력 공간: 그림 7 의 예 에는 N + M이 포함 됩니다.
기호 분기 및 S 최대 반복이 있는 기호 루프, 따라서 최악의 경우(가지치기
없음),
2N ·S·2 가 필요하다 중 상태 공간을 탐색하는 인터프리터.

알려진 길이. 전제 조건은 모든 입력이
최대 길이여야 합니다. 예를 들어 입력 데이터가 문자열 유형이면 다음과 같
은 전제 조건을 추가합니다.
최대 입력 길이까지 입력의 각 바이트
NULL이 아닙니다. 즉, (strlen(input) = len) 또는 논리적으
로 동등합니다(input[0] 6= 0) ∧ (input[1] 6= 0) ∧ ... ∧
(입력[len-1] 6= 0)∧(입력[len] = 0). 입력 공간:
길이가 len인 문자열의 입력 공간은 255len이 됩니다. 메모
len = S의 경우 이는 각 바이트에 대한 입력 공간의 0.4% 감소를 의미합니
다. 절약: 길이 조건은 N + M 기호 분기에 영향을 주지 않습니다.

그림 7의 예입니다. 그러나 기호 strcpy
직선 콘크리트 사본으로 변환됩니다.
길이와 가지 치기가 활성화되어 있음을 알기 때문에
가능한 모든 길이의 문자열 복사를 고려할 필요는 없습니다.
따라서 전체를 탐색 하려면 2N+M 인터프리터가 필요합니다.
상태 공간. 전체적으로 길이 전제 조건이 감소합니다.

입력 공간이 약간 있지만 strcpy를 루프처럼 구체화할 수 있습니다.
이는 버퍼 오버플로를 감지하기 위한 일반적인 패턴입니다.

알려진 접두사. 전제 조건은 접두사를 제한합니다.
입력 바이트에서, 예를 들어 HTTP GET 요청은 항상 시작됩니다.
"GET"을 사용하거나 특정 헤더 필드가
특정 범위의 값 내(예: 프로토콜 필드)
IP 헤더에서. 접두사 전제 조건을 사용하여 특정 항목으로 시작하는 입력에
대한 검색을 tar
접두사. 예를 들어,
이미지 처리 유틸리티의 PNG 이미지만 가능합니다. 그만큼
PNG 표준은 모든 이미지가
표준 8바이트 헤더 PNG H, 따라서 단순히 접두사 전제 조건을 지정함으로
써(input[0] = PNG H[0]) ∧ –
... ∧(input[7] = PNG H[7]), 검색에 집중할 수 있습니다.
PNG 이미지만. 접두사 전제 조건이 필요합니다.
정확한 평등으로 구성될 뿐만 아니라; 그들은 또한 기호에 대한 값의 범위
또는 열거를 지정할 수 있습니다.
바이트.
입력 공간: S 기호 바이트 및 정확한 접두사
길이 P(P < N < S)의 경우, 입력 공간의 크기는
256S P 입니다. 절감액: 그림 7에 표시된 예의 경우 ,
접두사 전제 조건은 첫 번째 P를 효과적으로 구체화합니다.
분기와 기호의 첫 번째 P 반복
strcpy, 따라서 필요한 통역사의 수를 S · 2 N+M P로 줄입니다.
· 접두사 전제 조건은 다음을 가질 수 있습니다.
상태 공간에 급진적인 영향을 미치지만 만병 통치약은 아닙니다. 을 위한
예를 들어 유효한 접두사만 고려하면 잘못된 헤더로 인해 발생할 수 있는
익스플로잇이 누락될 수 있습니다.

복통 실행. 전제 조건을 지정하지 않는 이중은 모든 입력 바이트가 특정 값
을 갖는다는 전제 조건을 지정하는 것입니다. 모든 입력 바이트에 특정 값
을 지정하는 것은 연쇄 실행과 동일합니다 [24]. 수학적으로 ∀i 를 지정합
니다 .

V (입력[i] = 구체적인 입력[i]).
입력 공간: 하나의 구체적인 입력이 있습니다. 저금:
프로그램을 탐색하려면 단일 통역사가 필요합니다.
그리고 state pruning 때문에 주어진 입력에 대한 실행 경로를 구체적으
로 실행하고 있습니다. 따라서 특히 concolic 실행의 경우 훨씬 더 유용합
니다.
상태 가지치기를 비활성화하고 새 인터프리터를 분기할 때마다 전제 조건
제약 조건을 삭제합니다. 참고로,
이 경우 AEG는 concolic fuzzer로 동작합니다.
구체적인 제약 조건은 초기 시드를 설명합니다. 조차
concolic 실행은 모든 방법 중에서 가장 제한적인 것처럼 보이지만 실제로
는 매우 유용할 수 있습니다.
예를 들어, 공격자는 이미 개념 증명(PoC—프로그램을 충돌시키는 입력)을
가지고 있을 수 있지만
작동하는 익스플로잇을 만들 수 없습니다. AEG는 해당 PoC를 사용할 수 있습니다.
프로그램이 AEG 지원 ex에서 악용될 수 있는 한 시드로 사용하고 악용을
생성합니다.

(a) 미리 정의된 기호 실행의 이점을 보여주는 예입니다.

		N 중
		N 중
		N P 중

(b) 왼쪽 예제 프로그램의 상태 공간 탐색에 필요한 입력 공간의 크기와 인터프리터의 수,
AEG가 지원하는 4가지 전제 조건 각각에 대해. 기호 입력 바이트 수를 나타내기 위해 S를 사용하고 길이에 대해 P를 사용합니다.
알려진 접두사(P < N < S).

5.3 경로 우선 순위 지정: 검색 휴리스틱

[redacted]

5.4 환경 모델링: 실제 세계의 취약점 탐지

AEG는 대부분의 시스템 환경을 모델링합니다.
공격자는 입력 소스로 사용할 수 있습니다. 그러므로,
AEG는 실제 프로그램에서 대부분의 보안 관련 버그를 감지할 수 있습니다. 환경 모델링에 대한 지원에는 다음이 포함됩니다.
파일 시스템, 네트워크 소켓, 표준 입력, 프로그램
인수 및 환경 변수. 추가적으로,
AEG는 가장 일반적인 시스템 및 라이브러리 기능을 처리합니다.
전화.

기호 파일. AEG는 다음과 유사한 접근 방식을 사용합니다.
기호 파일에 대한 KLEE의 [5] : 열기, 읽기 및 쓰기와 같은 기본 시스템 호출 기능을 모델링합니다.
AEG는 KLEE의 파일 시스템 모델을 단순화하여 속도 향상
우리의 주요 초점은 코드 적용 기간이 아니라 효율적으로 악용 가능한 버그 감지에 있기 때문에 분석합니다. 예를 들어, AEG는 추가 경로 생성을 피하기 위해 임무별과 같은 기호 파일 속성을 무시합니다.

심볼릭 소켓. 원격 익스플로잇을 생성할 수 있도록 AEG는 네트워킹 코드를 분석하기 위해 네트워크 지원을 제공합니다. 기호 소켓 설명자는 다음과 같습니다.
심볼릭 파일 디스크립터와 유사하게 처리되며 심볼릭 네트워크 패킷과 페이로드가 처리됩니다.
기호 파일 및 그 내용과 유사합니다. AEG는 현재 다음을 포함한 모든 네트워크 관련 기능을 처리합니다.
소켓, 바인드, 수락, 보내기 등

환경 변수. 여러 취약점은
특정 환경 변수로 인해 트리거됩니다.
따라서 AEG는 get env의 완전한 요약을 지원합니다.
가능한 모든 결과를 나타냄(구체적인 값, 완전히
상징과 실패).

라이브러리 함수 호출 및 시스템 호출. AEG는 약 70개의 시스템 호출을 지원합니다. AEG 지원
모든 기본 네트워크 시스템 호출, 스레드 관련 시스템
포크와 같은 호출 및 모든 일반적인 형식
printf 및 syslog를 포함한 기능. 스레드는
표준 방식으로 처리됩니다. 즉, 각 프로세스/스레드 생성 함수 호출에 대해 새로운 심볼릭 인터프리터를 생성합니다. 또한 AEG는 (완전히 또는 부분적으로) 상징적인

인수가 형식화 함수에 전달됩니다. 예를 들어 AEG는 형식 문자열 취약점을 감지합니다.
"fprintf(stdout, 사용자 입력)".

6 DBA, EXPLOIT-GEN 및 검증 : 익스플로잇 생성

높은 수준에서 AEG의 세 가지 구성 요소(DBA,
EXPLOIT-GEN 및 VERIFY) 가 함께 작동하여 변환
BUG-FIND 에 의한 안전하지 않은 술어 (Πbug) 출력
작업 익스플로잇 ε.

```
1 char * ptr = m all oc (1 0 0 ) ;
2 자 버퍼 [1 0 0 ] ;
ᵹ       strcpy(bu       ,   입력 ) ; // 오버플로
4       fstrcpy(pt r 리   ,   버프) ;        // ptrdereference
5       턴 );
```

그림 8: 스택 내용이 스택에 의해 왜곡된 경우
오버플로가 발생하면 프로그램이 반환 명령 전에 실패할 수 있습니다.

6.1 DBA: 동적 이진 분석

DBA는 동적 이진 분석(계측)입니다.
단계. 3가지 입력을 받습니다: 1) 대상 실행 파일
(Bgcc) 우리가 악용하려는; 2) 경로 제약
버그 (Πbug)로 이어집니다. 및 3) 스택 오버플로 공격 또는 버퍼에서 오버플로에 취약한 버퍼와 같은 취약한 기능 및 버퍼의 이름

형식 문자열에 악성 형식 문자열을 보유하는
공격. 그런 다음 런타임 정보 세트를 출력합니다. 1)
덮어쓸 주소(우리 구현에서 이것은
함수의 반환 주소 주소이지만 우리는
함수 포인터나 GOT의 항목을 포함하도록 이것을 쉽게 확장할 수 있습니다.
2) 우리가 쓰는 시작 주소,
3) 스택을 설명하는 추가 제약 조건
버그가 트리거되기 직전의 메모리 내용.

AEG가 버그를 찾으면 구체적인 입력을 사용하여 동일한 버그가 있는 실행 경로를 재생합니다. 구체적인 투입
경로 제약 Πbug를 해결하여 생성됩니다. DBA 동안 AEG는 주어진

실행 가능한 바이너리 Bgcc. 취약성을 감지했을 때
함수를 호출하면 실행을 중지하고 스택을 검사합니다.
특히, AEG는 취약한 함수(&retaddr)의 반환 주소 주소, 주소

덮어쓰기가 시작되는 취약한 버퍼(bu faddr)와 이들 사이의 스택 메모리 내용(μ).

형식 문자열 취약점의 경우 취약한 함수는 다음을 취하는 가변 형식화 함수입니다.
형식 인수로 사용자 입력. 따라서 주소는
반환 주소(&retaddr)의 주소는 취약한 서식 기능의 반환 주소가 됩니다. 예를 들어, 프로그램에 취약한 printf 함수가 있는 경우 AEG는 printf의 반환 주소를 덮어씁니다.

형식 문자열 취약점을 악용하는 기능 자체.
이런 식으로 공격자는 프로그램의 제어권을 가로챌 수 있습니다.
취약한 함수가 반환된 직후. 다음과 같은 추가 형식 문자열 공격을 적용하는 것은 간단합니다.
AEG에서 납치를 당했습니다.

스택 복원. AEG는 스택 내용을 검사합니다.
익스플로잇 술어를 생성하기 위해 DBA 동안
로컬 스택을 손상시키지 않는 (Πbug ∧ Πexploit)

EXPLOIT-GEN의 변수 (§ 6.2). 예를 들어
취약한 함수가 반환되기 전에 스택에서 역참조가 있습니다. 단순히 스택을 덮
어쓰면
항상 유효한 익스플로잇을 생성하지는 않습니다. 공격자를 가정해보자
다음을 사용하여 그림 8 에 표시된 프로그램을 악용하려고 시도합니다 .
strcpy 버퍼 오버플로 취약점. 이 경우 ptr
반환 주소와 버퍼 버퍼 사이에 있습니다.
스택 오버플로 후에 ptr이 역참조된다는 점에 유의하십시오.
공격. ptr도 스택에 있으므로
ptr은 스택 오버플로로 인해 왜곡되어 다음을 유발할 수 있습니다.
반환 명령 전에 프로그램이 충돌합니다. 따라서,
정교한 공격은 위의 경우를 고려해야 합니다.
스택에 대한 유효한 메모리 포인터를 덮어씁니다. AEG
전체를 검사하여 이 상황을 적절하게 처리합니다.
DBA 중 스택 공간 및 정보 전달
(μ) EXPLOIT-GEN.

6.2 익스플로잇 젠

EXPLOIT-GEN 은 익스플로잇을 생성하기 위해 두 가지 입력을 취합니
다. 경로 제약 (Πbug) 을 포함하는 안전하지 않은 프로그램 상태 와 저수준
런타임 정보 R, 즉,
취약한 버퍼의 주소(bufaddr), 주소
취약한 함수의 반환 주소(&retaddr),
및 런타임 스택 메모리 내용(μ). 사용
그 정보, EXPLOIT -GEN 은 4가지 유형의 익스플로잇에 대한
mulas (Πbug ∧ Πexploit) 에 대한 익스플로잇을 생성 합니다. 1)
스택 오버플로 스택으로 반환, 2) 스택 오버플로 libc로 반환, 3) 형식 문자
열 스택으로 반환, 4) 형식 문자열
libc로 반환. 이 논문에서는 1에 대해서만 전체 알고리즘을 제시합니다.

악용 기술은 당사 웹사이트에서 찾을 수 있습니다 [2].

익스플로잇을 생성하기 위해 AEG는 두 가지 주요 단계를 수행합니다. 첫
째, AEG는 공격 등급을 결정합니다.
제어 하이재킹을 위해 Πexploit 를 수행하고 공식화 합니다.
예를 들어 스택 오버플로 스택 반환 공격에서
Πexploit 는
리턴 주소(&retaddr)는 DBA에서 제공한 대로 쉘코드의 주소를 포함하도
록 덮어써야 합니다.
또한, 익스플로잇 술어 Πexploit 는 다음 을 포함해야 합니다 .
쉘코드가 타겟에 작성되어야 하는 제약조건
완충기. 생성된 술어는 함께 사용됩니다.
Πbug 를 사용하여 최종 제약 조건을 생성합니다(익스플로잇
공식 ∧ ∧ ∧ ∧ ∧ ∧ ∧ ∧ ∧ ∧
익스플로잇. 알고리즘 2 는 익스플로잇 술어가 어떻게
(Πexploit) 는 스택 오버플로 반환 스택에 대해 생성됩니다.
공격.

6.2.1 익스플로잇

AEG는 두 가지 유형의 익스플로잇을 생성합니다: return-to-stack [21]
및 return-to-libc [10], 둘 다 가장 인기 있는 고전적인 제어 하이재킹 공
격 기술입니다. AEG는 현재 최첨단 보호 체계를 처리할 수 없으며,

**알고리즘 2: 스택 오버플로 스택으로 반환 Exploit 술어 생성 알고리
즘**

입력 : (bufaddr, &retaddr, μ) = R
출력: Πexploit

```
1 for i = 1 to len(μ) do
3 오프셋 복구 &retaddr μ[i] ; 2                    // 스택 복원
bufaddr;
4 jmp 대상 ← 오프셋 + 8 ; // 이전 ebp + retaddr = 8
5 exp str[오프셋] ← jmp 대상 ; // eip 하이재킹
6 for i = 1 to len(shellcode) do
        exp str[오프셋 + i] ← 쉘코드[i];
7 8 return (Mem[bu f addr] == exp str[1]) ∧ ... ∧
   (Mem [bu f addr + len (μ) -1] == exp str [len (μ)]);
   // Πexploit
```

그러나 우리는 § 9 에서 가능한 방향에 대해 논의합니다. 또한,
우리의 return-to-libc 공격은 고전적인 공격과 다릅니다.
"/bin/sh"의 주소를 알 필요가 없다는 점에서
바이너리의 문자열. 이 기술은 우회를 허용합니다
스택 무작위화(libc 무작위화 아님).

스택으로 돌아가기 익스플로잇. 리턴 투 스택 익스플로잇
함수의 반환 주소를 덮어씁니다.
프로그램 카운터는 주입된 입력을 다시 가리킵니다. 예:
사용자 제공 쉘코드. 익스플로잇을 생성하기 위해 AEG
취약한 버퍼(bufaddr)의 주소를 찾아
복사할 수 있는 입력 문자열 및 주소
취약한 기능의 반환 주소가 있는 곳. 두 주소를 사용하여 AEG는 다음을 계
산합니다.
쉘코드가 위치한 점프 타겟 주소. 알고리즘 2 는 익스플로잇 조건자를 생성하
는 방법을 설명합니다.
쉘 코드가 뒤에 배치되는 스택으로 반환(return to stack) 익스플로잇의 경
우 스택 오버플로 취약성
반송 주소.

Return-to-libc 익스플로잇. 고전적인 return-to-libc에서
공격, 공격자는 일반적으로 반환 주소를 변경
libc의 execve 함수를 가리킵니다. 그러나
쉘을 생성하려면 공격자가 쉘의 주소를 알아야 합니다.
바이너리에서 "/bin/sh" 문자열은 일반적이지 않습니다.
대부분의 프로그램. return-to-libc 공격에서 우리는 다음을 생성합니다.
/bin/sh에 대한 심볼릭 링크와 링크 이름에 대해 우리는
libc에 있는 임의의 문자열을 사용하십시오. 시험 5 의 경우
ple, 5바이트 문자열 패턴 e8..00....16 매우 일반적입니다
libc에서는 x86에 대한 호출 명령을 나타내기 때문입니다.
따라서 AEG는 libc에서 특정 문자열 패턴을 찾고 대상 프로그램과 동일한
디렉토리에 /bin/sh에 대한 심볼릭 링크를 생성합니다. 문자열의 주소는

execve(execute에 대한 파일)의 첫 번째 인수로 전달되고 null 문자열
0000000016의 주소 가 사용됩니다 .
두 번째 및 세 번째 인수에 대해. 공격은 유효하다
로컬 공격 시나리오에만 해당하지만 더 안정적입니다.

도 5a의 점(.)은 4비트 문자열을 16진법으로 나타낸다.

스택 주소 무작위화를 우회합니다.

위의 익스플로잇 기술(return to stack 및 return-to-libc)은 생성 방법을 결정합니다.
제어 하이재킹 공격용 쉘이지만 하이재킹 방법은 아님
제어 흐름. 따라서 위의 기술은 다양한 유형의 제어 하이재킹 공격에 적용될 수 있습니다.
형식 문자열 공격 및 스택 오버플로. 예를 들어,
형식 문자열 공격은 위의 기술 중 하나를 사용하여 쉘을 생성할 수 있습니다.
AEG는 현재 위의 공격 공격 패턴의 가능한 모든 조합을 처리합니다.

6.2.2 착취 기법

다양한 쉘코드. return-to-stack 익스플로잇은 스택에 쉘코드를 주입해야 합니다. 지원하기 위해
다양한 유형의 익스플로잇, AEG에는 셸 코드 데이터베이스가 있습니다.
로컬 익스플로잇을 위한 표준 셸코드와 바인딩 및 역 바인딩 셸코드의 두 가지 셸코드 클래스 포함
원격 공격용. 또한 이 공격은
런타임 정보 μ를 사용하여 스택 내용
(§ 6.1).

익스플로잇의 유형. AEG는 현재 4가지 유형을 지원합니다.
익스플로잇: 스택 오버플로 스택으로 반환, 스택 오버플로 libc로 반환, 형식 문자열 스택으로 반환,
및 format-string return-to-libc 익스플로잇. 알고리즘
위의 각 익스플로잇에 대해 exp str을 생성하려면
알고리즘 2의 간단한 확장 . 관심 있는 독자
전체 알고리즘은 당사 웹사이트 [2] 를 참조하십시오.

셸코드 형식 및 위치 지정. 코드 인젝션에서
공격 시나리오에는 두 가지 매개변수가 있습니다.
항상 다음을 고려하십시오. 1) 형식(예: 크기 및 허용됨)
문자 및 2) 주입된 쉘 코드의 위치. 고급 공격이 있기 때문에 둘 다 중요합니다.
주입된 페이로드에 대한 복잡한 요구 사항, 예를 들어
익스플로잇 문자열은 제한된 바이트 수에 맞습니다.
또는 영숫자 문자만 포함합니다. 에게
위치 찾기, AEG는 무차별 대입 접근 방식을 적용합니다.
가능한 모든 사용자 제어 메모리 위치를 시도합니다.
쉘 코드를 배치하십시오. 예를 들어 AEG는
덮어쓴 반환 아래 또는 위의 쉘코드
주소. 특별한 포맷팅 문제를 해결하기 위해,
AEG에는 표준 및 영숫자를 포함하여 약 20개의 다른 셸 코드가 포함된 셸 코드 데이터베이스가 있습니다.
다시 말하지만, AEG는 신뢰성을 높이기 위해 가능한 모든 쉘코드를 시도합니다. AEG에는 VERIFY 단계가 있으므로 모든
생성된 제어 하이재킹이 실제가 되는 것으로 확인됨
착취.

6.2.3 익스플로잇의 신뢰성

특히 제어 흐름 하이재킹을 수행하는 익스플로잇은 섬세합니다. 길 등 작은 변화에도
프로그램은 ./a.out 또는 ../../../a.out을 통해 실행됩니다.
프로세스의 다른 메모리 레이아웃이 발생합니다.
이 문제는 ASLR이 꺼진 경우에도 지속됩니다.
같은 이유로 대중 자문에서 대부분의 개념 증명 익스플로잇은 일부(사소하거나 주요) 수정 없이는 실제로 작동하지 않습니다. 이 하위 섹션에서는 AEG에서 사용하는 기술에 대해 설명합니다.

주어진 시스템 구성에 대해 안정적인 익스플로잇을 생성하려면 a) 환경 변수의 차이를 상쇄하고 b) NOP-sled를 사용합니다.

환경 변수의 차이 상쇄.
환경 변수는 터미널, 길이가 다른 프로그램 인수 등에 따라 다릅니다.

프로그램이 먼저 로드되고 환경 변수는
프로그램의 스택에 복사됩니다. 스택이 커지기 때문에
메모리 주소가 낮을수록 더 많은 환경
변수가 있으면 스택에 있는 실제 프로그램 데이터의 주소가 더 낮아집니다.
OLDPWD 및 (밑줄)과 같은 환경 변수

프로그램이 호출되는 방식이 중요하기 때문에 동일한 시스템에서도 변경이 가능합니다. 또한, 주장
(argv)도 스택에 복사됩니다. 따라서 길이
명령줄 인수 중 메모리 레이아웃에 영향을 줍니다. 따라서 AEG는 크기의 차이를 기반으로 스택의 로컬 변수 주소를 계산합니다.

이진 분석 사이의 환경 변수
그리고 정상적인 실행. 이 기술은 일반적으로 다음과 같은 경우에 사용됩니다.
우리는 기계에서 익스플로잇을 만들고 실행해야 합니다.
다른 시스템에 대한 익스플로잇.

NOP-썰매. AEG는 선택적으로 NOP 썰매를 사용합니다. 단순성을 위해 알고리즘 2 는 NOP-sled 옵션을 사용하지 않습니다.
계정에. 일반적으로 큰 NOP 썰매는
특히 ASLR 보호에 대해 보다 안정적인 익스플로잇. 반면에 NOP-sled는 크기를 증가시킵니다.
잠재적으로 익스플로잇을 더 어렵게 만들거나 불가능하게 만듭니다. AEG의 경우 NOP 썰매 옵션
명령줄 옵션으로 켜거나 끌 수 있습니다.

6.3 확인

VERIFY 는 두 가지 입력을 받습니다: 1) 익스플로잇 제약 Πbug ∧Πexploit 및 2) 대상 바이너리. 다음 중 하나를 출력합니다.
구체적인 작업 익스플로잇, 즉 생성되는 익스플로잇
AEG가 익스플로잇을 생성하지 못한 경우 셸 또는 ⊥. VERIFY 는 먼저 익스플로잇 제약 조건을 해결하여 구체적인
악용하다. 익스플로잇이 로컬 공격인 경우 익스플로잇을 입력으로 하여 exe cutable을 실행하고 쉘이
산란되었습니다. 익스플로잇이 원격 공격인 경우 AEG
세 가지 프로세스를 생성합니다. 첫 번째 프로세스는 exe cutable을 실행합니다. 두 번째 프로세스는 nc를 실행하여 익스플로잇을 보냅니다.
실행 파일에. 세 번째 프로세스는 원격
셸이 포트 31337에서 생성되었습니다.

그림 5에서 직선을 보여주었다는 점에 유의하십시오.

단순성을 위해 PRE-PROCESS 에서 VERIFY 로의 라인 흐름 . 그러나 실제 시스템에서 VERIFY 는
제약 조건을 충족할 수 없는 경우 EXPLOIT-GEN 에 피드백
해결. 이것은 EXPLOIT-GEN 이 dif를 선택하기 위한 큐입니다.
참조 쉘코드.

7 구현

AEG는 C++와 Python을 혼합하여 작성되었습니다.
기호 실행기 (BUG-FIND), 동적 이진 평가기(DBA), 익스플로잇 생성기
(EXPLOIT -GEN) 및 제약 조건 솔버 의 4가지 주요 구성 요소로 구성됩니다.
(확인하다). 백엔드 심볼릭 실행기로 KLEE [5] 를 선택하고 약 5000줄의 코드를 추가했습니다.
우리의 기술과 휴리스틱을 구현하고
소켓과 같은 다른 입력 소스에 대한 지원 추가
및 기호 환경 변수). 우리의 동적 이진 평가기는 래퍼를 사용하여 Python으로 작성되었습니다.
GNU 디버거 [22]. 제약 조건에 STP를 사용했습니다.
해결 [12].

8 평가

다음 섹션에서는 실험
AEG 챌린지에서 작업합니다. 우리는 먼저 설명합니다
우리가 실험을 수행한 환경.
그런 다음 14개의 실제 응용 프로그램에 대해 AEG가 생성한 16개의 익스플로잇을 제시하여 AEG의 효율성을 보여줍니다. 다음으로 우리는 우리의 중요성을 강조합니다.
검색 휴리스틱 - 조건부 기호 포함
실행 - 악용 가능한 버그를 식별합니다. 게다가,
우리는 익스플로잇을 보여주는 몇 가지 예를 제시합니다
AEG에서 이미 구현된 기술. 마지막으로 생성된 익스플로잇의 신뢰성을 평가합니다. 생성된 각 익스플로잇에 대한 자세한 설명과 더 실험적인 결과를 보려면 독자에게 당사 웹사이트를 참조하십시오 [2].

8.1 실험 설정

기계에서 알고리즘과 AEG를 평가했습니다.
2.4GHz Intel(R) Core 2 Duo CPU 및 4GB
4MB L2 캐시가 있는 RAM. 모든 실험은 Debian Linux 2.6.26-2에서 수행되었습니다. LLVM GCC 2.7을 사용하여 소스 기반에서 실행할 프로그램을 컴파일했습니다.
바이너리 실행 파일을 빌드하기 위한 AEG 및 GCC 4.2.4. 모두
논문에 제시된 프로그램은 사람들이 사용하는 수정되지 않은 오픈 소스 응용 프로그램이며 인터넷에서 다운로드할 수 있습니다. 시간 측정은 Unix 시간 명령으로 수행됩니다. 버그 경로 우선 및 루프 소진 검색 휴리스틱

§ 5.3 은 모든 실험에 대해 기본적으로 켜져 있습니다.

8.2 AEG에 의한 익스플로잇

표 1 은 AEG가 성공적으로 악용한 취약점 목록을 보여줍니다. 우리는 이 14개의 프로그램을 다음에서 찾았습니다.
다양한 인기 권고: CVE(Common Vulnerabilities and Exposures),
오픈 소스 취약성
Database(OSVDB) 및 Exploit-DB(EDB)를 다운로드하고 AEG에서 테스트하기 위해 다운로드했습니다. AEG는 재현했을 뿐만 아니라
CVE에서 제공된 익스플로잇, 2개의 추가 취약점에 대해 작동하는 익스플로잇을 발견하고 생성했습니다.
1은 expect-5.43의 경우 1이고 htget-0.93의 경우 1입니다.
우리는 경로 탐색의 종류에 따라 테이블을 주문
버그를 찾는 데 사용되는 기술, 가장 작은 것부터 순서대로
알고리즘 자체에 주어진 대부분의 정보. 4개의 익스플로잇은 전제 조건이 전혀 필요하지 않으며 경로
경로 우선 순위 지정 기술(§ 5.3)만 사용하여 탐색했습니다. 우리는 비록 우리가 그 위에 구축하지만
KLEE [5], 우리 실험에서 KLEE는
iwconfig 악용 가능한 버그.
익스플로잇 중 6개는 추론한 후에만 생성되었습니다.
다음을 사용하는 기호 입력의 가능한 최대 길이
정적 분석(길이 행). 최대 입력 길이가 없으면 기호 실행이 가능한 모든 것을 고려하기 때문에 AEG가 가장 자주 실패합니다.

최대 버퍼 크기까지의 입력 길이,
일반적으로 매우 큽니다(예: 512바이트). 길이가
자동으로 유추된 이 6개는 이전 4개와 결합되어 총 10개의 익스플로잇이 추가 사용자 정보 없이 자동으로 생성되었음을 의미합니다.

사용자가 접두사를 지정해야 하는 5가지 익스플로잇
탐색할 입력 공간. 예를 들어 xmail의 취약한 프로그램 경로는 유효한 이메일로만 트리거됩니다.
주소. 따라서 AEG에 다음을 지정해야 했습니다.
취약성을 유발하는 "@" 기호가 입력에 포함됨
길.
Corehttp는 공동 실행이 필요한 유일한 취약점입니다. 우리가 제공한 입력은 "A"x입니다.
(880번 반복) + \r\n\r\n. 없이
완전한 GET 요청을 지정하면 공백을 배치할 위치를 탐색하는 데 기호 실행이 중단되었습니다.
및 EOL(행 끝) 문자.
생성 시간. 표 1 의 열 5는 작동하는 익스플로잇을 생성하는 총 시간을 보여줍니다. 우리가 가장 빨리
생성된 익스플로잇은 iwconfig에 대해 0.5초였습니다(길이
전제 조건), 단일 경로 탐색이 필요했습니다.
가장 긴 것은 1276초(21분 조금 넘게)의 xmail이었고 가장 많은 경로를 탐색해야 했습니다. 평균적으로
익스플로잇 생성은 테스트 스위트에 대해 114.6을 사용했습니다. 따라서,
AEG가 작동하면 매우 빠른 경향이 있습니다.
다양한 환경 모델링. 에서 회상
§ 5.4, AEG는 파일, 네트워크 패킷 등을 포함한 다양한 입력 소스를 처리합니다.
환경 모델링에서 AEG의 효과, 우리는
익스플로잇 유형별로 예제 그룹화(표 1 열
4) 이는 로컬 스택(로컬 스택의 경우

	프로그램	보다.	익스플로잇 유형	취약한 입력 src	세대 시간 (비서.)	실행 파일 코드 라인	자문 아이디.
없음	영겁	0.2a	로컬 스택	A V. 어디에.	3.8	3392	CVE-2005-1019
	iwconfig	V.26	로컬 스택	인수	1.5	11314	CVE-2003-0947
	glftpd	1.24	로컬 스택	인수	2.3	6893	OSVDB-ID # 16373
	압축	4.2.4	로컬 스택	인수	12.3	3198	CVE-2001-1413
길이	htget(프로세스URL) 0.93		로컬 스택	인수	57.2	3832	CVE-2004-0852
	htget(홈)	0.93	로컬 스택	A V. 어디에	1.2	3832	제로데이
	예상(DOTDIR) 5.43		로컬 스택	A V. 어디에	187.6	458404	제로데이
	기대하다(홈)	5.43	로컬 스택	A V. 어디에	186.7	458404	OSVDB-ID # 60979
	충격을 받은	1.4	로컬 형식 인수		3.2	35799	CVE-2004-1484
	탑xd	1.1.1	로컬 형식 인수		1.5	7244	OSVDB-ID # 12346
접두사	주문	0.50.5	로컬 스택	로컬 파일	15.2	550	CVE-2004-0548
	엑심	4.41	로컬 스택	인수	33.8	241856	EDB-ID # 796
	엑스서버	0.1a	원격 스택 소켓		31.9	1077	CVE-2007-3957
	재동기화	2.5.7	로컬 스택	A V. 어디에	19.7	67744	CVE-2004-2093
	메일	1.21	로컬 스택	로컬 파일	1276.0	1766	CVE-2005-2943
복통 코어http		0.5.3	원격 스택 소켓		83.6	4873	CVE-2007-4060
평균 생성 시간 및 실행 가능한 코드 줄					114.6	56784	

표 1: AEG가 성공적으로 악용한 오픈 소스 프로그램 목록. 생성 시간은 다음과 같이 측정되었습니다.
GNU 리눅스 시간 명령. 실행 가능한 코드 라인은 LLVM 명령어를 계산하여 측정되었습니다.

흐름), 로컬 형식(로컬 형식 문자열 공격용) 또는
원격 스택(원격 스택 오버플로용) 및 입력
출처(5열), 우리가 어디에 있는 출처를 보여줍니다.
익스플로잇 문자열을 제공합니다. 가능한 사용자 입력 소스
환경 변수, 네트워크 소켓, 파일, 명령줄 인수 및 stdin입니다.

2개의 제로데이 익스플로잇, expect와 htget은 둘 다
환경 변수 악용. 대부분의 공격 시나리오는 다음과 같은 환경 변수 취
약성에 대한 것입니다.
이것들은 그다지 흥미롭지 않습니다. 요점은 AEG가
새로운 취약점을 발견하고 자동으로 악용했습니다.

8.3 전제된 기호 실행 및 경로 우선 순위 지정 휴리스틱

8.3.1 전제된 기호 실행

우리는 또한 사전 조건이 지정된 기호 실행이 특정

다른 전제 조건이 사용될 때 취약점.

그림 9 는 결과를 보여줍니다. 최대 분석 시간을 10,000초로 설정한
후 종료합니다.
프로그램. 실패한 사전 조건화 기법
제한 시간 내에 악용 가능한 버그를 감지하면 표시됩니다.
그림 9 에서 최대 길이의 막대로 .
우리의 실험은
심볼릭 실행자에 제공되는 정보
전제 조건은 버그 감지 시간을 크게 향상시킵니다.
따라서 AEG의 효과. 예를 들어, 길이 전제 조건을 제공함으로써 AEG
가

시간 제한. 그러나 제공된 정보의 양
익스플로잇의 속도는 크게 변하지 않았습니다.
전혀 성공할 때 생성됩니다.

8.3.2 Buggy-Path-First: 연속적인 버그 감지

§ 5.3 에서 경로 우선 순위 지정 휴리스틱으로 회상
먼저 버그 경로를 확인하십시오. tipxd 및 htget은 이 우선순위 휴리
스틱이 지불하는 예제 응용 프로그램입니다.
끄다. 두 경우 모두 악용할 수 없는 버그가 있습니다.

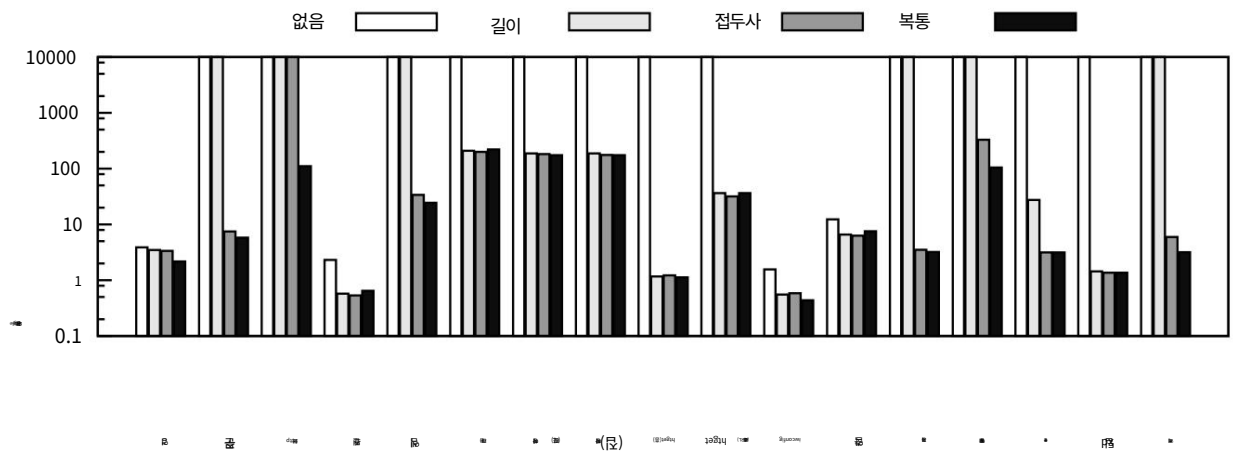

그림 9: 사전 조건이 지정된 기호 실행 기술의 비교.

```
1 if (!(sysinfo 2 · configfilename = m 모든 oc ( strlen ( optarg ) ) ) {
fprintf(stderrexit(1)) ; , "파일 이름 저장소에 대한 메모리를 할당할 수 없습니다.\n") ;
삼
4 }
5 strcpy ( ( char *) sysinfo 6 tipxdlog ( 로그 · 구성 파일 이름optarg ) ; ,
인 정보 7 , " C on fi gfileis %s \n "sysinfo , · 구성 파일 이름) ;
...
8 v oid tipxdlog( int 우선순위, ...) { char * for rm at ,
vsnprintf ( logentry , LOG ENTRY SIZE 1, 형식 9 , ap ) ;
10 syslog(우선순위, logentry) ;
```

그림 10: tipxd의 코드 스니펫.

같은 경로의 악용 가능한 버그에 의해 낮아졌습니다. 그림 10 은 tipxd의 스니펫을 보여줍니다.
1행의 악용할 수 없는 초기 버그(
NULL의 경우 "malloc(strlen(optarg) + 1)"
바이트). AEG는 버그를 악용할 수 없음을 인식합니다.
지속적인 탐색을 위해 해당 경로의 우선 순위를 더 높게 지정합니다.

나중에 경로에서 AEG는 10행에서 형식 문자열 취약점을 감지합니다.
구성 파일 이름은
5행의 명령줄 인수 optarg에서 설정,
임의의 형식 문자열을 syslog에 전달할 수 있습니다.
변수 로그 항목을 통해 10행의 함수. AEG
형식 문자열 취약점을 인식하고
적절한 명령을 만들어 형식 문자열 공격
라인 인수.

8.4 혼합 바이너리 및 소스 분석

§ 1에서 우리는 소스 코드 분석만으로는
스택 레이아웃과 같은 낮은 수준의 런타임 세부 정보가 중요하기 때문에 익스플로잇 생성에 충분하지 않습니다. 그만큼

```
1 int ProcessURL ( char *TheURL , 문자 *
호스트 이름 , char * 파일명 , 문자 *
Act u al Fil en am e , 부호 없는 * P ort ) {
2 자 BufferURL [MAXLEN] 3 자 NormalURL ;
[MAXLEN] strcpy ( BufferURL ;
4 , URL) ;
5 ...
6 strncpy ( 호스트 이름 , 일반URL , 나 ) ;
```

그림 11: htget의 코드 스니펫

aspell, htget, corehttp, xserver는 이 공리의 예입니다.

예를 들어, 그림 11 은 다음의 코드 스니펫을 보여줍니다.
앗. 이 함수를 호출할 때 스택 프레임에는 스택 맨 위에 함수 인수가 있습니다.
그런 다음 반환 주소와 ebp를 저장한 다음
로컬 버퍼 BufferURL 및 NormalURL. 그만큼

4행의 strcpy는 TheURL이
BufferURL보다 훨씬 길다. 그러나 우리는
반환 주소까지만 덮어쓰는 익스플로잇에 주의하십시오. 예를 들어 반환 주소를 덮어쓰는 경우
및 호스트 이름, 프로그램은 다음과 같은 경우 충돌합니다.
호스트 이름은 6행에서 (돌아오기 전에) 역참조됩니다.

우리 기술은 동적 분석을 수행하기 때문에
정확한 스택과 같은 런타임 세부 정보에 대해 추론할 수 있습니다.
레이아웃, 컴파일러가 할당한 정확히 몇 바이트
버퍼 등으로 매우 정확하게. 위 프로그램의 경우
이 정밀도는 예를 들어 htget 술어에서 필수적입니다.
반환 주소까지 덮어쓰지만
더 나아가. 페이로드를 놓을 공간이 충분하지 않은 경우
반환 주소 이전에 AEG는 스택 복원(§ 6.1 참조) 을 적용하여 여전히 악용을 생성할 수 있습니다.
지역 변수와 함수 인수가 있는 곳
덮어썼지만 우리는 그들의 값이
변경되지 않은 상태로 유지되어야 합니다. 그렇게 하기 위해 AEG는 다시
런타임을 검색하기 위한 동적 분석 구성 요소
지역 변수 및 인수의 값.

8.5 변종 악용

악용 가능한 버그가 발견될 때마다 AEG는 악용 공식
(Πbug ∧Πexploit) 을 생성하고
만족스러운 답변을 찾아 활용합니다. 그러나 이
하나의 만족스러운 답변이 있다는 의미는 아닙니다
(악용하다). 사실, 우리는 엄청난 숫자가 있을 것으로 예상했습니다.
공식을 만족하는 입력의 우리의 기대치를 확인하기 위해 추가 실험을 수행했습니다.
익스플로잇 변종을 생성하도록 구성된 AEG - 다른
동일한 익스플로잇 공식에 의해 생성된 익스플로잇. 표 2
AEG에 의해 생성된 익스플로잇 변종의 수를 보여줍니다.
5개의 샘플 프로그램에 대해 1시간 이내에

프로그램	익스플로잇 수
iwconfig	3265
압축	576
영겁	612
htget	939
glftpd	2201

표 2: 생성된 익스플로잇 변종 수
AEG는 1시간 이내입니다.

8.6 추가 성공

AEG는 또한 일화적인 성공을 거두었습니다. 우리의 연구
그룹은 smpCTF 2010 [27]에 참가했습니다.

보안 문제를 해결하여 기타. 도전 과제 중 하나는 주어진 바이너리를 악용하는 것이었습니다. 우리 팀이 달렸다
소스를 생성하기 위한 16진법 디컴파일러
AEG에 공급(일부 잘못된 부분을 수정하기 위한 약간의 조정 포함)
Hex-ray 도구에서 디컴파일). AEG 반환
60초 이내에 익스플로잇.

9 논의 및 향후 과제

고급 익스플로잇. 우리의 실험에서 우리는 집중했습니다.
스택 버퍼 오버플로 및 형식 문자열 취약성. 힙 기반 오버플로를 처리하도록
AEG를 확장하려면 힙 관리 구조도 고려하도록 제어 흐름 추론을 확장해야 합니다.

바로 확장입니다. 정수 오버플로는
그러나 일반적으로 흐름에 대한 정수 자체는 문제가 되지 않기 때문에 더 복잡합니다. 보안에 중요한 문제는 일반적으로 오버플로된 정수가 사용될 때 나타납니다.
메모리를 인덱싱하거나 할당합니다. 우리는 추가 지원을 떠납니다
향후 작업으로 이러한 유형의 취약성에 대해.

기타 악용 클래스. 우리의 정의에는 오늘날 악용되는 가장 인기 있는 버그가 포함되지만, 예를 들어 입력
정보 공개, 버퍼와 같은 유효성 검사 버그
오버플로, 힙 오버플로 등, 캡처하지 않습니다.
모든 보안에 중요한 취약점. 예를 들어, 우리의
공식은 범위를 벗어난 타이밍 공격을 남깁니다.
안전하다고 쉽게 특징지어지지 않는 암호화폐
문제. 우리는 AEG를 이러한 유형으로 확장합니다.
향후 작업으로 취약점.

기호 입력 크기. 우리의 현재 접근 방식은 단순한 정적 분석을 형성하고 기호 입력 변수의 크기가 다음보다 10% 커야 한다고 결정합니다.

가장 큰 정적으로 할당된 버퍼. 이 동안
KLEE보다 개선되었으며(KLEE는 사용자가 크기를 지정해야 함) 우리의 예에서는 충분했습니다.
다소 단순하다. 보다 정교한 분석은
예를 들어 스택 레이아웃을 고려하여 악용될 수 있는 항목에 대해 더 높은 정밀도를 제공하고
버퍼가 동적으로 할당되는 힙 오버 플로우와 같은 고급 익스플로잇에 필요합니다.

휴대용 익스플로잇. 우리의 접근 방식에서 AEG는
주어진 환경, 즉 OS, 컴파일러,
등. 예를 들어 AEG가 GNU에 대한 익스플로잇을 생성하는 경우
컴파일된 바이너리, 동일한 익스플로잇이 작동하지 않을 수 있음
인텔 컴파일러로 컴파일된 바이너리. 이는 익스플로잇이 컴파일러에서 컴파일러로 변경될 수 있는 런타임 레이아웃에 의존하기 때문에 예상됩니다. 그러나 A
로 컴파일할 때 작동하는 익스플로잇을 고려할 때,

컴파일러에서 생성된 바이너리에서 AEG를 실행할 수 있습니다.
B 새로운 익스플로잇을 만들 수 있는지 확인합니다. 또한 현재 프로토타입은 Linux 호환 익스플로잇만 처리합니다.
플랫폼 독립적이고 이식 가능한 익스플로잇을 만드는 것은
다른 저작 [7] 에서 다루었으며 범위를 벗어남
이 논문의.

10 관련 작업

자동 익스플로잇 생성. Brumley et al. [4] 는 자동 패치 기반 익스플로잇 생성(APEG) 챌린지를 도입했습니다. 그들은 또한 익스플로잇이 프로그램 상태 공간에 대한 술어로 설명될 수 있다는 개념을 도입했으며, 이는 우리가 이 작업에서 사용하고 개선합니다. AEG와 APEG에는 두 가지 중요한 차이점이 있습니다.

첫째, APEG는 버그가 있는 프로그램과 패치에 대한 액세스가 필요하지만 AEG는 잠재적인 버그가 있는 프로그램에만 액세스해야 합니다. 둘째, APEG는 패치에 의해 도입된 새로운 안전 검사를 위반하는 입력으로 익스플로잇을 정의합니다(예: 그림 4에서 안전하지 않은 입력만 생성).

반면 Brumley et al. 루트 쉘을 생성하는 것이 가능할 수 있다고 추측하지만, 그들은 그것을 보여주지 않습니다. 우리는 특정 작업을 포함하도록 "공격"의 개념을 확장하고 셸 시작과 같은 특정 작업을 생성할 수 있음을 보여줍니다. Heelan et al.의 MS 논문. [13], Brumley et al.의 기술과 대략 유사한 기술을 사용하여 익스플로잇 생성을 탐색합니다. [4].

버그 찾기 기술. 블랙박스 퍼징에서는 프로그램이 실패하거나 충돌할 때까지 임의의 입력을 프로그램에 제공합니다 [19]. 블랙박스 퍼징은 사용하기 쉽고 저렴하지만 복잡한 프로그램에서는 사용하기 어렵다. 기호 실행은 취약성 발견 및 테스트 케이스 생성 [5, 6], 입력 필터 생성 [3, 8] 등을 포함하여 여러 애플리케이션 도메인에서 광범위하게 사용되었습니다. 기호 실행은 단순성 때문에 매우 인기가 있습니다. 일반 실행처럼 작동하지만 데이터(일반적으로 입력)가 기호가 될 수도 있습니다.

구체적인 값 대신 기호 데이터에 대한 계산을 수행함으로써 기호 실행을 통해 단일 실행으로 여러 입력에 대해 추론할 수 있습니다.

Taint 분석은 신뢰할 수 없는 사용자 입력이 신뢰할 수 있는 싱크로 유입될 수 있는지 여부를 결정하기 위한 일종의 정보 흐름 분석입니다. 정적 [15, 18, 26] 및 동적 [20, 28] 오염 분석 도구가 있습니다. 상징적 실행과 taint 분석에 대한 보다 광범위한 설명은 최근 설문 조사 [23]를 참조하십시오.

기호 실행 AEG 설정에 적용할 수 있는 기호 실행 및 형식적 방법에는 다양한 작업이 있습니다. 예를 들어, Engler et al. [11] 은 구체화를 위해 기호 데이터에 등식 제약이 부과되는 정확히 제약된 기호 실행의 아이디어를 언급했으며 Jager et al. 잠재적으로 더 효율적으로 익스플로잇 생성에 필요한 공식을 생성할 수 있는 방향 없는 가장 약한 전제 조건을 도입합니다 [14]. 우리의 문제 정의를 통해 모든 형식의 검증을 사용할 수 있으므로 AEG를 개선할 때 공식 검증 작업을 시작하는 것이 좋습니다.

11 결론

이 백서에서 우리는 익스플로잇 생성을 위한 최초의 완전 자동 종단 간 접근 방식을 도입했습니다. 우리는 AEG에서 접근 방식을 구현하고 14개의 오픈 소스 프로젝트를 분석했습니다. 우리는 16개의 제어 흐름 하이재킹 익스플로잇을 성공적으로 생성했으며 그 중 2개는 이전에 알려지지 않은 취약점에 대한 것이었습니다. AEG를 실용적으로 만들기 위해 우리는 악용 가능한 버그를 찾고 식별하기 위한 새로운 사전 조건 지정 기호 실행 기술과 경로 우선 순위 지정 알고리즘을 개발했습니다.

12 감사의 말

AEG 프로젝트에서 일한 모든 사람들, 특히 이종협, David Kohlbrenner 및 Lokesh Agarwal에게 감사드립니다. 유익한 논평과 제안을 해주신 익명의 심사자에게도 감사드립니다. 이 자료는 그랜트 번호 0953751에 따라 National Science Foundation에서 지원하는 작업을 기반으로 합니다. 여기에 표현된 모든 의견, 결과, 결론 또는 권장 사항은 저자의 것이며 반드시 National Science Foundation의 견해를 반영하는 것은 아닙니다. 이 작업은 또한 사이버 보안 연구 컨소시엄의 일부인 Northrop Grum man, Lockheed Martin 및 DARPA Grant No.

N10AP20021.

참고문헌

[1] AEG. 자동 익스플로잇 생성 데모. http://www.youtube.com/watch?v=M_nuEDT-xaw , 2010년 8월.

[2] D. 브룸리. 카네기 멜론 대학 보안 그룹. http://security.ece.cmu.edu.

[3] D. Brumley, J. Newsome, D. Song, H. Wang, S. Jha. 취약점 기반 서명의 자동 생성을 위한 이론 및 기술. 신뢰할 수 있고 안전한 컴퓨팅에 대한 IEEE 트랜잭션, 5(4):224–241, 2008년 10월.

[4] D. Brumley, P. Poosankam, D. Song 및 J. Zheng. 자동 패치 기반 익스플로잇 생성이 가능합니다. 기술 및 의미. 2008년 5월 보안 및 개인 정보 보호에 관한 IEEE 심포지엄 회보에서.

[5] C. Cadar, D. Dunbar 및 D. Engler. Klee: 복잡한 시스템 프로그램에 대한 높은 범위의 테스트를 자동으로 생성합니다. 2008년 운영 체제 설계 및 구현에 관한 USENIX 심포지엄 회보에서.

[6] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill 및 D. En gler. EXE: 상징적 실행을 사용하여 죽음의 입력을 자동으로 생성하는 시스템. 컴퓨터 및 통신 보안에 관한 ACM 회의 절차, 2006년 10월.

[7] SK Cha, B. Pak, D. Brumley, RJ Lipton.

플랫폼 독립적인 프로그램. 절차에서
컴퓨터 및 통신 보안에 관한 ACM 회의, 2010.

[8] M. Costa, M. Castro, L. Zhou, L. Zhang,
M. 페이나도. Bouncer: 차단하여 소프트웨어 보안
잘못된 입력. ACM 심포지엄 진행 중
운영 체제 원칙, 2007년 10월.

[9] M. Costa, J. Crowcroft, M. Castro, A. Rowstron,
L. Zhou, L. Zhang, P. Barham. 자경단: 인터넷 웜의 종단 간 격리.
의 절차에서
운영 체제 원칙에 대한 ACM 심포지엄,
2005.

[10] S. 디자이너. "libc로 반환" 공격. Bugtraq, 1997년 8월.

[11] D. Engler 및 D. Dunbar. 제약이 적은 실행:
자동 코드 파괴를 쉽고 확장 가능하게 만듭니다. ~ 안에
소프트웨어 테스팅 및 분석에 관한 국제 심포지엄, 2007년 1-4페이지.

[12] V. Ganesh 및 DL 딜. 비트 벡터 및 배열에 대한 결정 절차입니다. 회의에
대한 절차에서
Computer Aided Verification, 강의 4590권
Computer Science의 Notes, 524–536페이지, 2007년 7월.

[13] S. 힐런. 제어 흐름 자동 생성 하이재킹 소프트웨어 취약점에 대한 익스플
로잇. 전문인
보고서 석사 논문, Oxford University, 2002.

[14] I. Jager 및 D. Brumley. 효율적인 방향성 가장 약한
전제 조건. 기술 보고서 CMU-CyLab-10-002,
Carnegie Mellon University, CyLab, 2010년 2월.

[15] R. 존슨과 D. 와그너. 사용자/커널 포인터 찾기
유형 추론이 있는 버그. USENIX의 절차에서
보안 심포지엄, 2004.

[16] J. 킹. 상징적 실행 및 프로그램 테스트. ACM 통신, 19:386–394, 1976.

[17] C. 라트너. LLVM: 평생 프로그램 분석 및 변환을 위한 컴파일 프레임워
크. 2004년 코드 생성 및 최적화에 관한 심포지엄 회보에서.

[18] VB Livshits 및 MS Lam. 정적 분석을 사용하여 Java 애플리케이션에서
보안 취약점을 찾습니다. 2005년 USENIX 보안 심포지엄 회의록에서.

[19] B. Miller, L. Fredriksen 및 B. So. 실증적 연구
UNIX 유틸리티의 신뢰성. 통신
컴퓨터 기계 협회, 33(12):32–
1990년 4월 44일.

[20] J. Newsome과 D. Song. 자동 탐지, 분석 및 서명 생성을 위한 동적 오
염 분석
상용 소프트웨어에 대한 익스플로잇. 절차에서
네트워크 및 분산 시스템 보안 심포지엄,
2005년 2월.

[21] A. 하나. 재미와 이익을 위해 스택 스매싱. 프랙,
7(49), 1996. 파일 14/16.

[22] PyGDB. gdb용 파이썬 래퍼. http://코드.
google.com/p/pygdb/.

[23] EJ Schwartz, T. Avgerinos 및 D. Brumley. 당신들 모두
동적 오염 분석 및
순방향 상징적 실행(그러나 두려웠을 수도 있음)
묻다). Se에 대한 IEEE 심포지엄 진행 중
curity and Privacy, 2010년 5월.

[24] K. Sen, D. Marinov 및 G. Agha. CUTE: 복통
C. In Proceedings of the Joint에 대한 단위 테스트 엔진
유럽 소프트웨어 엔지니어링 회의 및 기초에 대한 ACM 심포지엄
소프트웨어 공학, 2005.

[25] H. Shacham, M. Page, B. Pfaff, E.-J. 고,
N. Modadugu 및 D. Boneh. 주소 공간 무작위화의 효율성. 의 절차
에서
컴퓨터 및 통신에 관한 ACM 회의
보안, 298–307페이지, 2004.

[26] U. Shankar, K. Talwar, J. Foster 및 D. Wagner. 유형 한정자를
사용하여 형식 문자열 취약성을 감지합니다. ~ 안에
USENIX 보안 심포지엄의 절차, 2001.

smpCTF. smpctf 2010. http://ctf2010.
smpctf.com/.

[28] GE Suh, J. Lee, S. Devadas. 동적 정보 흐름 추적을 통한 안전한 프로
그램 실행. 프로그래밍 언어 및 운영 체제에 대한 아키텍처 지원에 관한
국제 회의의 회보, 2004.