버퍼 오버플로우 취약점 자동 탐지 시스템

김재환*, 김한결*, 김현정**, 유상현**, 원일용*
*서울호전문대학교 사이버해킹보안과
**건국대학교 상허교양대학 소속 초빙교수
e-mail: bibe0829@gmail.com, alwayskim9305@naver.com, nygirl@konkuk.ac.kr,
yoosh22@gmail.com, clccclcc@hoseo.ac.kr

Buffer Overflow Vulnerability Auto Detection System

Jae-Hwan Kim*, Han-Gyeol Kim*, Hyun-Jung Kim**, Sang-Hyun Yoo**, Il-Yong Won*
*Dept. of Cyber Hacking Security, Seoul Hoseo Technical College
*Dept. of Sang-huh college, Konkuk University

요 약

버퍼 오버플로우(Buffer Overflow) 취약점은 시스템 보안에서 아주 중요하다. 본 논문은 바이너리 파일에서 버퍼 오버플로우 취약점을 자동 감지하는 시스템을 제안하였다. 다양한 버퍼 오버플로우 샘플소스를 이용하여 패턴을 만들고, 이렇게 만들어진 패턴을 이용하여 바이너리 파일에서 버퍼 오버플로우 취약점이 포함되어 있는지를 판단한다. 제안한 시스템의 유용성을 위해 실험을 하였고, 어느 정도 의미 있는 결과를 얻을 수 있었다.

1. 서론

버퍼 오버플로우(Buffer Overflow)는 데이터가 일시적으로 저장되는 공간인 버퍼에 공격자가 함수의 입력을 저장하기 위해 할당된 버퍼 크기보다 더 많은 데이터를 입력하여 메모리 공간을 조작하고 시스템의 제어 권한을 획득 및 이를 이용한 다른 취약성 공격을 시도하는 곳에 여전히 이용되고 있다. 이러한 버퍼 오버플로우 공격을 효과적으로 연구하기 위해 많은 연구가 진행되고 있으나 특별한 경우에 대해서는 버퍼 오버플로우 공격에 대응 할 수 없거나 대응을 하더라도 보완이 필요하다[3,4].

이러한 문제를 해결하기 위해 버퍼 오버플로우 취약점을 발견할 수 있는 기존의 다양한 기술들이 연구가 되고 있다. 정적 분석의 이점을 이용하여 코드가 배포되기 전에 보안 취약점을 제거하는 시스템의 프로토타입을 제안했다. 그러나 이는 규모가 큰 응용 프로그램에서 확장하기 쉽지만, 최적화 되지 않았으므로 처리 시간이 추가로 필요하고, 많은 false negative 가 발생한다는 문제가 있다[7]. 이진 코드 변환을 이용한 버퍼 오버플로우 방지기법은 바이너리 파일을 디스어셈블하여 어셈블리 레벨의 코드를 분석한다. 그러나 취약 함수 중에 strcpy()만을 고려했기 때문에 다른 취약 함수를 사용하는 특수한 경우에는 버퍼 오버플로우를 탐지할 수 없거나 프로그램의 속도가 느려지는 문제가 있다[3]. 또 다른 방법은 기호 실행 기법을 이용하여 세가지 버퍼 오버플로우 패턴을 탐지하는 방법이 있다. 그러나 이 방법은 strcpy()와 동일한 작업을 수행하는 사용자 정의 연산을 고려하지 않고, 많은 false positive 가 발생한다는 문제점이 있다[8]. 바이너리 파일을 디스어셈블 하여 생성된 어셈블리 코드를 제안된 보안 규칙 수준에서 분석하고 strcpy()호출을 어셈블리 명령에서 감지하여 단순 탐지 모듈과 콜그래프, 컨트롤 플로우 그래프와 데이터 플로우 그래프를 이용하여 고급 탐지 모듈을 포함시켜 취약점 보고서를 작성해주는 모델 제안도 연구 되었다다[4]. 이러한 버퍼 오버플로우 취약점을 대응하기 위한 다양한 방법의 연구되고 있지만 특수한 취약점의 경우가 생기면 막을 수 없다는 문제점이 있다.

대다수의 버퍼 오버플로우 공격을 위해 이미 만들어진 프로그램의 소스가 없더라도 그 프로그램의 바이너리 레벨에서 검사할 수만 있다면, 버퍼 오버플로우의 취약점을 사전에 파악하는 것이 가능할 수 있으며, 이러한 방법을 연구하는 것은 매우 어려운 일이나 보안이라는 관점에서 유용 할 수 있다.

본 논문에서는 바이너리 레벨의 프로그램을 리버싱 기법을 이용하여 어셈블리 레벨에서 분석하고, 버퍼 오버플로우를 발생시킬 수 있는 취약점의 패턴을 찾는 버퍼 오버플로우 취약점 자동 점검 시스템을 제안한다. 본 논문에서는 사용된 버퍼 오버플로우 공격 방법은 이미 널리 알려진 다양한 버퍼 오버플로우 샘플에서 리버싱된 소스를 비교 분석해 버퍼 오버플로우 탐지 패턴을 만들어 내는 방식이다.

본 논문의 구성은 다음과 같다. 2 장에서는 정규표현식과 버퍼 오버플로우 공격에 대한 관련 연구를 논하고, 3 장에서는 제안하는 시스템의 설계에 대하여 설명한다. 4 장에서는 제안하는 시스템을 위해 적용한 실험 및 분석에 대해 기술하고, 마지막으로 5 장에서는 결론 및 향후 과제로 맺는다.

2. 관련 연구

2.1 정규표현식

정규 표현식(Regular Expression, Regex)은 프로그래밍에서 텍스트의 특정한 규칙을 갖는 문자열을 검색하고 치환하기 위해 사용하는 강력한 기능이다. 대부분의 텍스트 편집기와 프로그래밍 언어에서 이 기능을 지원하고 있다. 표현이 간결하다는 장점이 있지만 그만큼 가독성이 떨어지므로 표현식의 문법을 정확히 알고 있어야만 하며, 프로그래밍마다 문법이 조금씩 다르다.

정규 표현식은 주로 웹 프로그래밍에서 사용되며, 주로 문자열 관련 프로그래밍이나 패턴 매칭 알고리즘 등에서 사용된다. POSIX(Portable Operating System Interface) BRE, POSIX ERE, 그리고 PCRE(Perl Compatible Regular Expression) 방식이 있으며, PCRE 방식은 확장성이 뛰어나 대부분의 프로그래밍 언어에서 라이브러리를 제공하고 있다. 이중 Python 언어의 정규 표현식 API 인 re 모듈은 아래 Table1 와 같은 방식을 사용하고 각각의 정규식은 패턴 매칭을 위해 빈도 높게 사용된다.

Table 1. Regular Expression Examples

Regular Expression	
\d	숫자인 것을 검색
\w	문자나 숫자인 것을 검색
\s	White Space 인 것을 검색
. (Dot)	\n 을 제외한 모든 것을 검색
Greedy Operator	
*	앞 문자가 0 번 이상 반복인 것을 검색
+	앞 문자가 1 번 이상 반복인 것을 검색
{}	앞 문자의 반복 횟수를 지정
Non-Greedy Operator	
?	문자 앞에 위치할 시, 해당 문자가 있거나 없음을 의미하고, +나 *뒤에 위치할 시, 멈추고 최소 반복만을 수행

2.2 버퍼 오버플로우 공격

버퍼 오버플로우 공격은 크게 스택 오버플로우와 힙 오버플로우가 있다. 스택 오버플로우는 C 와 C++ 프로그래밍 언어에서 주로 발생하는데, 버퍼 배열에 기록되는 데이터가 배열 범위를 넘어가는지 확인하는 기능을 제공하지 않기 때문이다. 반면에 Python 과 Java 프로그래밍 언어의 경우는 자체적으로 메모리 환경을 관리하기 때문에 오버플로우가 발생하지 않는다. 예를 들어, Java 에서는 배열의 크기를 초과하는 영역에 접근하려고 할 때 ArrayOutOfBounds 예외가 발생한다. C# 프로그래밍 언어는 자동으로 배열의 경계 검사를 수행하고, checked 컨텍스트에서 Overflow Exception 예외를 통해 오버플로우를 감지할 수 있기 때문에 오버플로우가 발생하지 않는다[2].

버퍼 배열에 대한 경계 검사를 수행하는 코드를 사용하지 않았을 때, 공격자는 버퍼 배열의 크기보다 큰 입력 데이터 값을 이용하여 덮어 씌어 스택 프레임 리턴 주소가 공격 코드를 가리키도록 변경한다. 이때 함수는 호출된 곳으로 이동이 아닌 공격자가 의도한 공격 코드를 실행하게 된다[5]. 이러한 문제를 해결하기 위해 취약점이 존재하는 C 표준 라이브러리 함수가 있고, C 언어에서 strcpy(), strcat(), gets() 등과 같은 취약 함수들의 사용을 피할 것을 권고했다[6].

3. 버퍼 오버플로우 취약점 자동 탐지 시스템

본 논문에서 제안하는 시스템은 여러 개의 버퍼 오버플로우 취약점 샘플들 중에서 패턴을 찾아낸 후 이 패턴을 이용하여 버퍼 오버플로우 취약점 유·무를 판단하는 시스템이다. 제안하는 시스템은 크게 두 개의 모듈로 구성되어 있다. 첫 번째 모듈은 패턴을 추출하는 모듈이며, 두 번째 모듈은 패턴을 이용하여 취약점을 찾는 탐지 모듈이다.

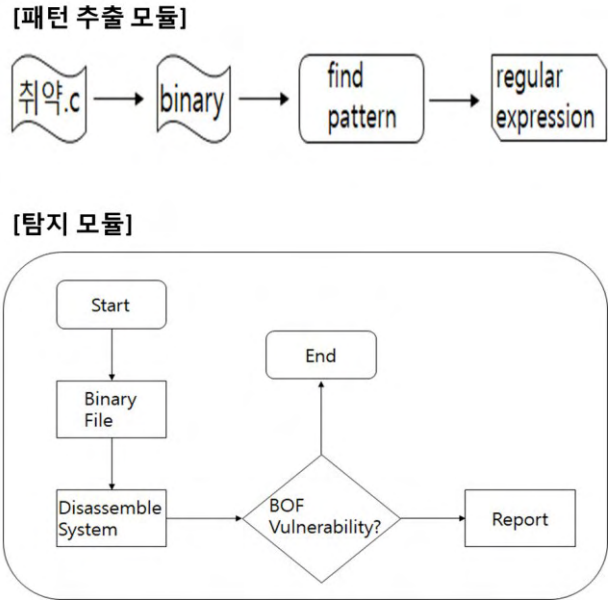

Figure 1. System Flowchart

Figure 1 의 시스템 순서도에서 패턴 추출 모듈은 알려진 버퍼 오버플로우에 취약한 C 언어 함수의 샘플들을 바이너리 파일로 바꾼 뒤 그 바이너리 파일을 디스어셈블을 한 후 파일들을 비교하여 취약한 부분의 패턴을 찾는다. 패턴을 찾은 뒤 찾은 패턴을 이용하여 정규식으로 표현했다. Figure1 의 시스템 순서도에서 탐지 모듈에 따라 검사하는 바이너리 파일을 본 논문에서 제안하는 디스어셈블을 이용한 내용과 패턴 추출 모듈에서 표현한 정규식과의 매칭을 통해 해당 파일이 취약 함수를 사용하였는지 또는 취약 함수를 보안할 수 있는 버퍼 길이 확인 함수인 strlen()를 사용하였는지 여부를 판단 한다.

4. 실험 및 분석

4.1 패턴 생성

제안한 방법을 검증하기 위해 버퍼 오버플로우 취약점의 다양한 소스를 수집하여 실행 파일로 변경하고 이를 이용한 바이너리를 리버싱해서 어셈블리어 수준에서 분석을 진행했다. 수집한 소스는 스택 기반 버퍼 오버플로우 취약점 소스 20 개 이며, 이 소스에 버퍼 오버플로우 취약점을 바이너리 레벨에서 소거한 샘플 20 개를 생성했다. 실험은 Linux 환경에서 C 언어를 사용하여 테스트를 했고, 바이너리를 디스어셈블하기 위해 라이브러리나 컴파일된 오브젝트 모듈, 공유 오브젝트 파일, 독립 실행 파일 등의 바이너리 파일 정보를 보여주는 GNU 바이너리 유틸리티인 objdump 툴을 이용했다. 또한 본 실험을 위해 사용한 샘플 중 하나는 버퍼 오버플로우에 취약한 함수인 strcpy 를 사용했다. 그리고 두 개의 매개변수는 사용자가 입력하는 값이다. Figure 2 는 취약한 소스를 디스어셈블한 코드이다.

```
00000000004006d6 <main>:
  4006d6:       55                      push   %rbp
  4006d7:       48 89 e5                mov    %rsp,%rbp
  4006da:       48 83 ec 60             sub    $0x60,%rsp
  4006de:       89 7d ac                mov    %edi,-0x54(%rbp)
  4006e1:       48 89 75 a0             mov    %rsi,-0x60(%rbp)
  4006e5:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4006ec:       00 00
  4006ee:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4006f2:       31 c0                   xor    %eax,%eax
  4006f4:       48 8d 3d 09 01 00 00    lea    0x109(%rip),%rdi        # 400804 <_IO_stdin_used+0x4>
  4006fb:       e8 70 fe ff ff          callq  400570 <puts@plt>
  400700:       48 8d 45 b0             lea    -0x50(%rbp),%rax
  400704:       48 89 c6                mov    %rax,%rsi
  400707:       48 8d 3d fc 00 00 00    lea    0xfc(%rip),%rdi        # 40080a <_IO_stdin_used+0xa>
  40070e:       b8 00 00 00 00          mov    $0x0,%eax
  400713:       e8 a8 fe ff ff          callq  4005c0 <__isoc99_scanf@plt>
  400718:       48 8d 45 d0             lea    -0x30(%rbp),%rax
  40071c:       48 89 c7                mov    %rax,%rdi
  40071f:       b8 00 00 00 00          mov    $0x0,%eax
  400724:       e8 87 fe ff ff          callq  4005b0 <gets@plt>
  400729:       48 8d 55 b0             lea    -0x50(%rbp),%rdx
  400737:       e8 24 fe ff ff          callq  400560 <strcpy@plt>
  400734:       48 89 c7                mov    %rax,%rdi
  400737:       e8 24 fe ff ff          callq  400560 <strcpy@plt>
  40073c:       48 8d 55 d0             lea    -0x30(%rbp),%rdx
  400740:       48 8d 45 b0             lea    -0x50(%rbp),%rax
  400744:       48 89 c6                mov    %rax,%rsi
  400747:       48 8d 3d bf 00 00 00    lea    0xbf(%rip),%rdi        # 40080d <_IO_stdin_used+0xd>
  40074e:       b8 00 00 00 00          mov    $0x0,%eax
  400753:       e8 38 fe ff ff          callq  400590 <printf@plt>
  400758:       b8 00 00 00 00          mov    $0x0,%eax
  40075d:       48 8b 4d f8             mov    -0x8(%rbp),%rcx
  400761:       64 48 33 0c 25 28 00    xor    %fs:0x28,%rcx
  400768:       00 00
```

Figure 2. BOF Vulnerabilities Source Assemblies

Figure 2 와 같은 디스어셈블한 코드에서 취약한 곳을 찾아 만든 패턴의 정규식 일부를 Figure 3 과 같이 표현했다.

```
".+callq.+<strcpy@.+\n"
```

Figure 3. Vulnerability Pattern Regular Expression

Figure 3 은 strcpy 를 사용하여 디스어셈블 했을 때 나오는 식을 정규식으로 표현한 것이다.

4.2 BOF 탐지

4.1 단계에서 만들어진 바이너리에 대한 탐지 결과는 Table 2 와 같다.

Table 2. BOF Detection Result

	BOF	NOT BOF
BOF	100	0
NOT BOF	0	100

Table 2 에서 첫 번째 열은 실제 결과, 첫 번째 행은 본 논문의 실험을 위한 시스템이 찾은 결과로 전체적인 탐지율은 100% 이다. 이러한 실험 결과가 나오는 이유는 실험 환경이 실제 환경과는 달리 C 언어에서 공격에 취약한 함수만을 이용하여 본 논문을 위한 프로그램에 최적화되어 있어 객관성이 떨어진다, 또한 strlen() 함수가 사용되었는지 아닌지 판단할 수 없기 때문에 실험에서 실제 사용자가 만든 함수나 공격 코드가 탐지가 된 것인지를 판별하지 못한다는 문제점도 있다. 이러한 이유로 False positive 도 0% 관찰되었다.

실험의 객관성을 높이기 위하여 추가적인 실험으로 Linux 기본 폴더에 있는 실행파일들을 가지고 다시 실험 했으며 그 결과는 Table3 과 같다. Table 3 은 BOF 알람을 받은 바이너리를 나타낸 것이다.

Table 3. BOF Alert Binary Sample

	BOF 알람을 받은 바이너리 예
/bin	bash, bzip2recover, cpio, fuser…
/usr/bin	bsd-write, cpp-5, csplit, editer…
/usr/sbin	filefrag, grub-probe, pptp, zic…

5. 결론 및 향후과제

버퍼 오버플로우 취약점은 여전히 동작하는 취약점으로 시스템 보안의 관점에서 매우 위험하다. 소스를 알 수 없는 바이너리에서 이러한 취약점을 자동으로 찾는 연구가 꾸준히 진행되고 있다. 본 논문에서는 보안 시스템에서 흔히 사용하는 프레임워크를 이용하여 바이너리에서 버퍼 오버플로우 취약점 포함 여부를 감지할 수 있는 시스템을 제안하였다.

시스템은 샘플로부터 패턴을 만드는 부분과 이를 이용하여 만들어진 패턴을 이용하여 바이너리에서 버퍼 오버플로우 취약점을 탐지하는 탐지 모듈로 나눴고 제한한 시스템을 실험한 결과는 어느 정도 성능을 보여 주었다. 그러나 실험에서 몇 가지 문제점 중에 strlen() 함수가 사용되었는지 아닌지 판단할 수 없기 때문에 실험에서 실제 사용자가 만든 함수나 공격

코드가 탐지가 된 것인지를 판별하지 못하는 한계점이 있었다.

향후 과제는 strlen()이 취약 함수를 보안하기 위해 사용되었는지 판단하는 것과 사용자가 만든 공격코드까지 탐지와 Sample 에서 자동으로 패턴을 추출하는 연구 및 마지막으로 취약점이 있는 바이너리를 자동으로 패치를 할 수 있는 연구를 해볼 수 있을 것이다.

참고문헌

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi, "A Survey of Symbolic Execution Techniques", 2016

[2] MSDN, "Array Bounds Check Elimination in the CLR", https://blogs.msdn.microsoft.com/clrcodegeneration/2009/08/13/array-bounds-check-elimination-in-the-clr/, 2009

[3] 김윤삼, 조은선, "이진 코드 변환을 이용한 효과적인 버퍼 오버플로우 방지기법", 2005

[4] Shehab Gamal El-Dien, Reda Salama, and Ahmed Eshak, "Buffer Overflow Vulnerability Detection in the Binary Code", 2008

[5] Nathan P. Smith, "Stack Smashing Vulnerabilities in the Unix Operation System", 1997

[6] Arash Baratloo, Navjot Singh, and Timothy Tsai, "TRANSPARENT RUN-TIME DEFENSE AGAINST STACK SMASHING ATTACKS", 2000

[7] David Wagner, Jeffrey S.Foster, Eric A.Brewer, Alexander Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities", 2000

[8] Sun Ding, Hee Beng Kuan Ten, Kaiping Liu, Mahinthan Chandramohan, Hongyu Zhang, "Detection of Buffer Overflow Vulnerabilities in C/C++ with Pattern Based Limited Symbolic Evaluation", 2012