2016년 추계학술발표대회 논문집 제23권 제2호(2016. 11)

보안 정보 보호를 위한 프로그램 데이터 근원 분석

안선우*, 신장섭*, 방인영*, 백윤흥*
*서울대학교 전기정보공학과
e-mail:swahn@sor.snu.ac.kr

Tracking Sensitive Data Source for Secret Information Protection

Sun-Woo Ahn*, Jang-Seop Shin*, In-Young Bang*, Yun-Heung Paek*
*Dept of Electrical and Computer Engineering, Seoul National University

요 약

컴퓨터 시스템은 악성 프로그램 또는 프로그램의 취약점을 통한 해커의 공격 등 위험에 항시 노출되어 있다. 따라서 이러한 컴퓨터 시스템에서 처리되는 보안적으로 중요한 데이터는 언제나 노출될 위험이 있다. 이 문제를 해결하기 위해 보안적으로 중요한 데이터의 처리 과정을 다른 프로그램 부분과 분리하여 다른 프로그램 부분의 취약점을 통해 해당 데이터까지 위험에 노출되는 것을 막으려는 연구들이 있다. 이러한 연구들에서 필요한 것은 보안적으로 중요한 데이터가 프로그램 상에서 처리되는 시작점을 찾고, 해당 데이터가 처리되는 부분을 분석하는 것이다. 본 연구에서는 그 중 첫 번째 문제를 자동적으로 풀기 위한 컴파일러 기반 분석 도구를 개발하였다.

1. 서론

컴퓨터 시스템은 악성 프로그램이나 프로그램의 취약점을 통한 해커의 공격에 노출되어 있다. 사용자의 부주의로 인한 악성 프로그램 설치, 또는 개발자의 부주의로 인한 프로그램 취약점 생성을 원천적으로 막는 것은 거의 불가능하다. 또한 컴퓨터 시스템 상 소프트웨어는 많은 부분이 메모리 오류에 취약한 C/C++언어로 작성되어 이를 통한 공격의 위험에 노출되어 있다.

한편, 컴퓨터 시스템은 종종 보안적으로 중요한 정보를 처리한다. 예를 들면 개인정보, 계좌정보, 비밀번호, 저작권 컨텐츠, 회사 보안 서류, 암호화 키 등이 있다. 정상적인 프로그램이 이러한 정보들을 외부로 유출시키지 않더라도, 프로그램 상에 내재한 취약점을 통해 공격자가 해당 정보를 빼내갈 수 있다. 최근 이슈가 된 Heartbleed 취약점[1]이 그 위험성을 잘 드러낸다.

이러한 문제를 해결하기 위해 보안 정보를 처리하는 부분을 나머지 부분과 분리하여 나머지 부분의 취약점을 통해 보안 정보까지 탈취하는 일을 방지하려는 연구들이 있다. 보통 보안 정보를 처리하는 부분은 전체 프로그램에 비하면 극히 일부분에 지나지 않고, 프로그램 취약성 존재 가능성은 프로그램 크기에 비례하는 경향이 있기 때문에, 취약점이 공격받아도 적어도 지키고자 하는 데이터를 지키는 데에는 효과적이다. 이런 방법에서 공통적으로 요구되는 것은 지키고자 하는 데이터를 정의하는 것과 프로그램 상에서 해당 데이터를 처리하는 부분을 식별하는 것이다.

보안 데이터를 처리하는 부분을 찾아내기 위해서는 우선 해당 데이터가 입력되는 시점을 찾고 데이터의 흐름을 분석하여야 한다. 개발자가 이를 수작업으로 행하는 것은 시간과 정확성 측면에서 생산적이지 못하다. 본 연구에서는 그 중 첫 번째로 개발자가 간단한 프로그램 상의 표시를 통해 보안적으로 중요한 데이터가 입력되는 시작점을 찾아주는 컴파일러 기반 도구를 개발하였다. 이를 위해 Pointer Alias 정보를 제공해 주는 Data Structure Analysis [2]를 사용하였다.

본 논문의 구성은 다음과 같다. 2장에서는 본 연구의 동기가 되는 프로그램 분리 기반 보안 연구들에 대해 소개한다. 3장에서는 본 연구에서 활용한 Data Structure Analysis에 대해 간단히 소개하겠다. 4장에서는 본 연구에서 개발한 보안 정보 근원 분석 도구를 소개한다. 5장에서는 개발된 도구에 대한 실험 결과를 제시하고 이에 대해 분석하겠다.

2. 배경 - 프로그램 분리 기반 보안 연구

프로그램 분리는 보안 문제 해결에 있어 널리 쓰이는 접근 방법이다. 가장 익숙하게는 운영체제 상에서 프로세스 간의 분리를 예로 들 수 있다. 운영체제 상의 프로세스들은 메모리 공간이 분리되어 서로 간섭이 불가능하다.

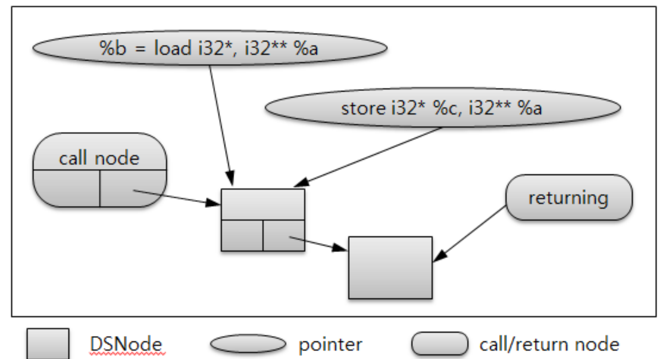
많은 연구들이 이러한 기본 원리를 바탕으로 보안 문제에 대한 해결책을 제시하였다. Code Pointer Integrity [2]

연구에서는 프로그램의 수행 흐름을 변조할 수 있는 코드 재사용 공격 (Code Reuse Attack) 등을 막기 위해 프로그램 상의 Code Pointer 들을 특별한 공간에 관리하고, 이들에 대한 접근에 대해서는 메모리 접근 오류를 점검하는 방식을 사용하였다. 이 방식은 개발자가 원하는 다른 특정 데이터에 대해서도 적용이 가능하다. HDFI [4] 연구에서는 보안적으로 중요한 데이터는 메모리 상에서 메모리 태깅을 통해 표시하고 특수 하드웨어 명령어를 통해서만 이에 접근할 수 있도록 하고, 특수 하드웨어 명령어를 통한 메모리 접근은 메모리 오류를 점검하여 보안적으로 중요한 데이터를 분리시켰다, SeCage [5] 연구에서는 하이퍼바이저를 사용하여 보안적으로 중요한 데이터를 처리하는 부분에 대한 분리를 제공하고, Intel 아키텍처의 특정 기능을 이용하여 보안 영역과 비 보안 영역 간의 Context 변경을 신속하게 할 수 있도록 구현하였다. 산업계에서는 ARM TrustZone 과 Intel SGX 아키텍쳐에서 하드웨어 기반의 분리된 실행 환경을 제공하고 있다.

서론에서 언급했듯이, 이러한 환경을 효과적으로 이용하기 위해서는 이에 맞는 어플리케이션 개발이 필요하고, 이를 위해서는 보안적으로 중요한 데이터가 처리되는 부분을 찾아 별도의 코드로 분리해 내는 것이 중요하다.

3. Data Structure Analysis

본 장에서는 본 연구에서 개발한 보안 정보 근원 탐색 도구의 바탕이 되는 Data Structure Analysis에 대해 소개한다. Data Structure Analysis는 프로그램 상의 포인터가 어떤 메모리 객체를 가리키는지를 분석하는 Points-to Analysis이다. Data Structure Analysis는 프로그램의 각 함수마다 DSGraph를 생성한다. DSGraph는 프로그램 상의 포인터와 그 포인터가 가리키는 메모리 객체에 대한 정보를 담고 있다. (그림 1)은 DSGraph의 예를 나타낸다. DSNode는 해당 함수에서 어떤 포인터가 가리킬 수 있는 메모리 객체를 나타낸다. 예제의 load 명령은 %a라는 주소에서 32-bit integer pointer형 값을 읽어오고, store 명령은 %a 주소에 %c라는 32-bit integer pointer 형 변수를 저장한다. %b 와 %c는 같은 DSNode를 가리키게 하는 것을 통해 %b와 %c가 같은 메모리 객체를 가리킬 수 있음을 나타낸다. DSNode가 나타내는 대상이 또 다른 포인터를 포함하고 있다면, 이는 또 따른 DSNode를 생성하여 이를 가리키게 한다. 예제에서의 DSNode가 struct 이고, 해당 struct의 두 번째 요소가 포인터라면, 그 포인터가 가리키는 메모리 객체를 또 다른 DSNode로 나타낸다. 또한 DSGraph에는 call node와 return node가 있다. call node는 해당 함수에서 다른 함수 호출이 있을 때, 그 매개 변수 중 포인터가 있다면 그 포인터가 가리키는 메모리 객체를 나타낸다. return node는 해당 함수가 포인터 형 변수를 반환할 때, 그 포인터가 가리키는 메모리 객체를 나타낼 때 쓰인다. call node와 return node는 interprocedural 분석을 할 때 사용된다.



(그림 1) DSGraph 예

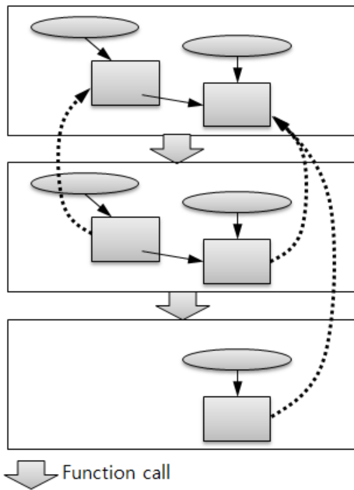
Data Structure Analysis는 먼저 각각 함수에 대해 명령어들을 하나하나 분석하여 DSGraph를 생성한다. 그 후 call graph를 Bottom-Up으로 방문하며 Callee 정보를 Caller graph에 통합하고, 마지막으로 Top-Down으로 call graph를 방문하며 Caller graph의 정보를 Callee graph에 통합한다. 그 과정에서 포인터가 가리키는 별개의 DSNode가 합쳐질 수 있고, 이는 해당 포인터가 런타임에 같은 메모리 객체를 가리킬 수 있음(May-Alias)을 나타내게 된다.

4. 보안 데이터 근원 분석 도구

앞서 언급했듯이, 보안적으로 중요한 데이터를 분리된 환경에서 보호하기 위해서는 해당 데이터가 처리되는 부분을 알아야 하고, 이를 위해서는 우선적으로 해당 데이터가 프로그램 상에 유입되는 지점을 찾아야 한다. 보안적으로 중요한 데이터는 주로 파일이나 네트워크를 통해 프로그램에 유입된다. 예를 들어, OpenSSL 에서는 유저가 비밀키가 담긴 파일 경로를 프로그램 입력으로 주면, 해당 경로에서 파일을 읽어 비밀키를 얻게 된다. OpenSSL을 분석해본 결과, 해당 경로를 나타내는 char* 형 변수는 찾기가 매우 쉬운 반면, 해당 경로에서 비밀키를 읽어오는 fopen 시스템 콜의 위치는 바로 찾기가 쉽지 않았다. 따라서, 본 연구에서는 이런 작업을 자동적으로 처리할 수 있는 보안 데이터 근원 분석 도구를 개발하였다. 도구를 사용하기 위해서는 개발자가 프로그램 소스 코드 상에서 보안적으로 중요한 데이터 파일 경로를 나타내는 char* 변수에 __attribute__ ((annotate(“sensitive”))) 와 같은 표시를 한다. 그 후 프로그램 소스 코드를 도구에 입력시키면 해당 경로에서 파일을 읽어오는 fopen 시스템 콜의 위치를 출력해 준다.

이 기능을 구현하기 위해 Data Structure Analysis의 결과를 이용하였다. 개발자가 표시한 char* 형 변수가 가리키는 메모리 객체와 같은 곳을 fopen 시스템 콜의 첫 번째 매개 변수가 가리킬 수 있다면 해당 fopen 시스템 콜은 개발자가 표시한 보안 데이터를 읽어오는 가능성이 있는 시스템 콜임을 의미하게 된다. 기본적인 Data Structure Analysis의 결과는 같은 함수 내에서의 포인터 및 메모리 객체 정보만을 포함하므로, 프로그램 전체에서 이와 같은

분석이 가능케 하려면 Data Structure Analysis의 결과에 대한 추가적인 분석이 필요하다.



(그림 2) 보안 데이터 근원 분석을 위한 추가 정보 생성

(그림 2)는 본 연구에서 구현한 추가적인 분석을 나타낸다. 추가적인 분석에서는 개발자가 표시한 char* 형 변수와 같은 곳을 가리키는 fopen 시스템 콜을 찾기 위해, 각 DSGraph의 DSNode를 call graph 상에서 caller 함수의 DSGraph의 Aliasing되는 DSNode에 연결시켰다. 이는 call graph를 post-order로 방문하며, call site에서 caller-callee 사이의 DSNode의 Aliasing 관계 정보를 축적하여 생성하였다. 이를 통해, 임의의 함수의 DSNode에 대하여, 자신을 호출한 함수 history의 최상위 함수에 있는 May-alias DSNode를 알 수 있었다. 이를 이용하여, 개발자가 표시한 char* 형 변수 값(보안적으로 중요한 데이터 파일 경로)을 parameter로 받을 수 있는 fopen 시스템 콜을 찾을 수 있다.

5. 결과

```
==== test.c ====
#include "my_fopen.h"

char *sensitive __attribute__ ((annotate("sensitive")));

int main(int argc, char **argv) {
  FILE *f1, *f2;
  sensitive = argv[1];
  f1 = fopen(sensitive, "r");
  f2 = my_fopen_r(sensitive);
  return 0;
}
==== my_fopen.c ====
#include "my_fopen.h"

FILE *my_fopen_r(char *path) {
  return fopen(path, "r");
}
```

```
==== output ====
test.c:8

my_fopen.c:4
 inlined @ test.c:9
```

(그림 3) 보안 정보 근원 분석 결과

(그림 3) 은 본 연구에서 개발한 보안 정보 근원 분석 도구의 수행 결과를 나타낸다. 개발자가 “sensitive”로 표시한 변수에 대해서 그 값을 사용하는 fopen 시스템 콜의 위치를 출력해 준다. Indirect function call을 자주 사용하는 복잡한 프로그램에서는 indirect function call site에서 호출 가능한 함수를 파악하는데 한계가 있어 분석 결과에 정확성에 문제가 발생하였다. 추후 연구에서 이 문제를 보정할 계획이다.

6. 결론

최근 보안 솔루션으로 제시되고 있는 분리(isolation) 기반의 기법들을 효율적으로 이용하기 위해서는 프로그램에서 보안적으로 중요한 부분을 분리해 내는 것이 필요하다. 본 연구에서는 이를 자동으로 수행하기 위한 첫 단계로, 보안 정보 근원을 분석하는 도구를 개발하였다. 추후 이 결과를 바탕으로 보안 데이터 처리 부분을 추출하는 연구를 할 계획이다.

감사의 글

이 논문은 2016년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원(No. R0190-16-2010, 시스템 침입 탐지를 위한 프로세서 모니터링 기술 및 주요 HW/SW 모듈 개발)과 2016년도 정부(미래창조과학부)의 재원으로 정보통신기술진흥센터의 지원 (No. R-20160222-002755, 맞춤형 보안서비스 제공을 위한 클라우드 기반 지능형 보안기술 개발), 미래창조과학부 및 정보통신기술진흥센터의 대학ICT연구센터육성 지원사업(IITP-2016- R0992-15-1006), 2016년도 두뇌한국21플러스 사업 및 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임.(No.2014R1A2A1A10051792)

참고문헌

[1] Codenomicon and N. Mehta, “The Heartbleed Bug,” http://heartbleed.com/, 2014

[2] Lattner, Chris, and Vikram Adve. Data structure analysis: An efficient context-sensitive heap analysis. Tech. Report UIUCDCSR-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2003.

[3] Kuznetsov, Volodymyr, et al. "Code-pointer integrity." 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). 2014.

[4] Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Paek, Y. (2016). HDFI: Hardware-Assisted Data-flow Isolation. Chicago

[5] Liu, Yutao, et al. "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.