Heap 영역 코드의 정적 스캔

목성균, 엄기진, 조은선
충남대학교 컴퓨터공학과
e-mail:manggoguy@naver.com
e-mail:kjeom2104@gmail.com
e-mail:eschough@cnu.ac.kr

A static scanning method for heap section codes

Seong-Kyun Mok, Ki-Jin Eom, Eun-Sun Cho
Dept of Computer Science and Engineering, Chungnam National University

요 약

힙(heap) 영역은 데이터 또는 코드가 동적으로 할당되는 영역이다. 따라서 heap 영역에 악성코드가 할당되면, 바이러스 백신이 탐지하거나, 분석하기가 어려워진다. Heap 영역의 코드가 실행될 경우, 사용자의 허가를 받지 않은 동작을 수행하여 사용자에게 피해를 끼칠 수 있다. 본 논문에서는 여러 가지 툴을 이용하여 heap 영역의 코드를 정적 스캔하는 방법을 제시한다.

1. 서론

해커들은 악성 프로그램을 만들 때, 또는 프로그램의 취약점을 이용하여 악성코드를 심을 때, 자신의 코드를 숨기고자 한다. 백신의 탐지를 피하기 위해서 또는 자신의 코드가 분석되는 것을 피하기 위해서 자신의 코드를 숨긴다. 이 때, 이용하는 한 가지 방법으로 힙(heap)에 코드를 생성하는 것이다.

힙영역에 코드가 생성될 경우, 프로그램을 실행하지 않고서는 그 여부를 알 수가 없기 때문에 해커들은 이러한 특성을 이용하여 백신의 탐지를 회피한다. 또한, 힙영역은 동적으로 메모리를 할당한다는 특성 때문에 코드를 분석하는 것 또한 어려워 진다. 이러한 프로그램을 역어셈블리해서 힙 영역에 코드를 할당하는 것을 정적 분석을 통해서 예상 할 수 있지만 이 코드가 사용자에게 해를 가하는 코드인지 알 수 없고, 어떤 side-effect가 발생하는지 알 수 없다. 즉, 사용자에게 허가받지 않은 작동을 할 수 있다. 그래서 힙 영역에 생성된 코드에 대해서 스캔할 수 있는 방법이 필요하다.

본 논문에서는 힙 영역에 생성된 코드를 정적 스캔하는 방법을 제시하고자 한다.

2. 힙 영역 코드의 스캔 절차

힙 영역에 소스코드를 할당하는 프로그램에 대해서 정적 분석으로는 그 여부를 판단하기 어렵지만 프로그램이 실행 된 이후에는 소스코드를 힙 영역에 할당하고 그 코드를 수행한다는 것을 이용하여 동적으로 생성된 코드를 스캔할 수 있다.

제시하고자 하는 방법은 다음과 같다.

(그림 1) 힙 영역 코드 스캔 절차 흐름도

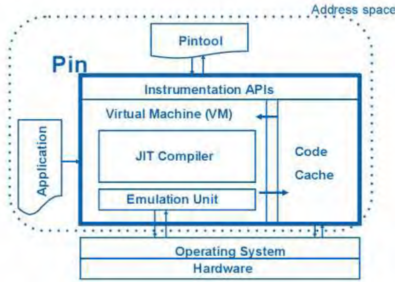
(그림 1)과 같은 과정을 거쳐서 힙 영역의 코드를 스캔할 수 있다.. 일반적인 정적 스캔으로는 프로그램이 힙 영역에 생성하는 코드에 접근할 수 없기 때문에 과정 1과 같은 과정을 거친다. 2와 3의 과정은 정적 스캔을 위해서 힙 영역의 코드를 리버싱을 할 수 있게 가공하는 과정이다. 4은 3의 과정을 통해 나온 결과물을 리버싱 한다. 과정 5는 과정 4를 통해 나온 결과물을 스캔할 수 있는 형태로 만들고 이어서 스캔을 시작한다.

2.1 디버깅 단계

일반적인 정적 분석으로는 힙 영역에 생성된 코드에 접근을 할 수가 없다. 힙 영역에 접근하려면 프로그램을 실행 시켜야만 한다.

그래서 디버거로 프로그램을 실행한 뒤, 힙 영역의 코드에 도달하는 것이 이 과정에서의 목표이다. 디버거는 API로 활용이 가능한 Pin[1]을 사용한다. 아래의 (그림 2)는 Pin의 구성도이다. Pin은 intel에서 나온 동적 분석을 위한 도구로 여러 API가 제공되기 때문에 힙 영역 코드에

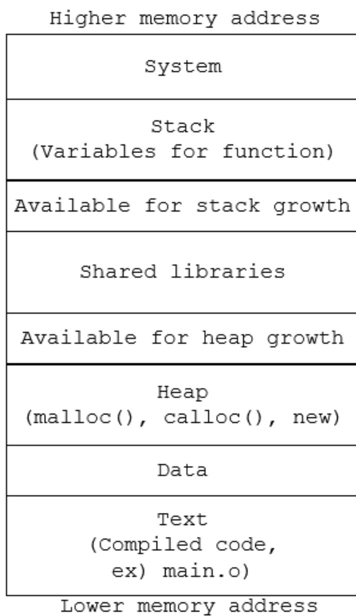
접근하기를 더 원활하게 할 수 있다.



(그림 2) Pin의 구성도[1]

2.2 실행코드가 힙 영역의 코드인지 판단

아키텍처 또는 프로그래밍 언어 마다 프로그램이 메모리에 업로드 되었을 때의 레이아웃이 다르지만 데이터 영역, stack 영역, 힙 영역 그리고 코드 영역으로 나눠진다. [그림3]은 x86아키텍처에서 C로 작성한 프로그램이 메모리에 업로드 되었을 때의 전형적인 모습이다. 즉, 코드 영역과 힙 영역은 다른 곳에 생성이 되기 때문에 이를 통해서 실행되는 코드가 코드 영역에 올라온 정상적인 코드인지, 아니면 힙 영역에 생성된 코드인지 구분할 수 있다. 힙 영역에 코드가 생성될 경우, 실행하는 코드의 주소는 코드 영역이 아닌 힙 영역의 주소를 나타낼 것이다. 이를 이용하여 정상적인 code 영역의 주소인지, 아니면 힙 영역의 주소인지 판단할 것이다.



(그림 3) x86 프로그램 실행 시 레이아웃

본 논문에서는 바이너리에서 힙의 코드를 실행하는 경우를 2가지 경우로 한정해서 고려할 것이다. 첫 번째는 call을 하는 경우이다. call의 오퍼랜드가 함수가 아닌 주소일 경우, 그 주소로 넘어가서 코드를 실행하게 된다. 두번째는 ret 명령어 수행 시 return의 주소가 힙의 주소로 되어있어서, EIP가 힙영역으로 넘어간 경우이다. 이 경우는 대부분의 해커가 이용하는 방법이다. 힙 영역에 코드를 생성 후, return 주소를 바꿔서 자신이 만든 코드로 EIP가 변경 되서 실행한다. 따라서 이 2가지 경우에 대해서만 고려할 것이다.

```
BOOL LEVEL_CORE::INS_IsCall(INS ins)
Returns:
     true if ins is a Call instruction
BOOL LEVEL_CORE::INS_IsRet(INS ins)
Returns:
     true if ins is a Return instruction
```

<표 1> 명령어가 call, ret 인지 체크하는 메소드[1]

Intel에서 제공하는 Pin(Dynamic binary instrumentation framework)을 사용하여 구현할 경우, API를 이용하여 보다 쉽게 구현할 수 있다. 위의 두 함수를 이용하면 call일 경우와, ret일 경우 두 가지 경우에 대해서 구현하면 된다. call일 경우, operand의 주소값을 보고 확인하면 된다. ret의 경우 돌아가는 주소를 확인하면 된다.

```
ADDRINT  LEVEL_PINCLIENT::INS_DirectBranchOrCallT
argetAddress(INS ins)
Returns:
     For direct branches or calls, the target
address
```

<표 2> 명령어가 call, ret 인지 체크하는 메소드[1]

위의 함수를 이용하면, call이 불릴 경우의 주소 또는 ret 명령어가 실행 되서 되돌아가는 주소를 알아낼 수 있다. 이 주소로 EIP가 코드영역으로 가는지 힙 영역으로 가는지 판단할 수 있다.

2.3 메모리 파일화

2.2의 과정을 거친 후, 힙 영역의 코드에 해당하는 메모리를 파일로 만든다. 이 과정은, 역어셈블러가 바이너리를 읽고 어셈블리어로 나오게 하기 위한 과정이다.

2.4 역어셈블리

Capstone[2]은 역어셈블리 프레임워크로 여러 아키텍처에 대해서 제공한다. Windows는 물론 유닉스 계열(Mac OSX, iOS, Android, Linux 등)을 지원한다. 가장 큰 장점은 여러 언어를 지원한다는 점이다. 후에 설명할 binNavi[3] 플러그인은 자바로 작성할 수 있다. Capstone도 자바를 지원한다. jar 파일의 형태로 capstone 프레임워크를 이용할 수 있다.

```
1 // Test.java
2 import capstone.Capstone;
3
4 public class Test {
5
6   public static byte [] CODE = { 0x55, 0x48, (byte) 0x8b, 0x05, (byte) 0xb8,
7     0x13, 0x00, 0x00 };
8
9   public static void main(String argv[]) {
10     Capstone cs = new Capstone(Capstone.CS_ARCH_X86, Capstone.CS_MODE_64);
11     Capstone.CsInsn[] allInsn = cs.disasm(CODE, 0x1000);
12     for (int i=0; i<allInsn.length; i++)
13       System.out.printf("0x%x:\t%s\t%s\n", allInsn[i].address,
14           allInsn[i].mnemonic, allInsn[i].opStr);
15   }
16 }
```

(그림 4) Capstone을 활용한 역어셈블리 예제[2]

Capstone을 이용해서 2.3의 과정을 통해 나온 파일을 역어셈블리 할 수 있다. 위의 예제 코드처럼 역어셈블리를 하면 명령어의 주소, 명령어의 mnemonic, 명령어의 operand를 결과로 얻을 수 있다. 이 과정을 통해 명령어의 mnemonic과 명령어의 operand를 결과로 얻을 수 있다.

2.5 코드 스캔

binNavi[3]는 REIL(리버스 엔지니어링을 위한 중간코드) 코드를 지원하는 리버싱 툴이다. REIL코드의 명령어는 총 17개로 다른 아키텍처의 명령어보다 적기 때문에 분석을 더 용이하게 할 수 있다.

```
static Instruction.creat(final Module module,
                         final Address address,
                         final String Mnemonic,
                         final List<Opernad> opernads
                         final byte[] data,
                         final String architecture)
Create a new instruction which is part of this
module. The new instruction is immediately
stored in the database.
```

<표 3> binNavi에 명령어를 삽입하는 메소드[3]

위의 capstone으로 얻은 결과를 binNavi의 API를 이용하여 모듈에 삽입해서 코드가 있는 것처럼 분석할 수 있다. 이를 바탕으로 코드가 파일 입출력을 하는지, 아니면 네트워크 연결 같은 동작을 하는지 또는 System API를 사용하는지 스캔한다. 이를 바탕으로 프로그램이 허가받지 않은 동작을 수행하는지 알 수 있다.

3. 결론

힙은 메모리를 동적으로 할당한다는 특징 때문에 해커들이 코드를 숨기거나 분석을 어렵게 하기 위해서 악용해왔다. return 주소를 덮어씌워서 힙 영역의 코드로 EIP를 옮기거나 혹은 힙 영역의 코드를 call 하는 방식으로 코드를 실행했고, 앞으로도 이에 관한 방법은 더욱 다양해질 것이다. 따라서 힙 영역의 코드에 대해서 스캔할 수 있는 방법이 필요하다. 본 논문에서는 힙 영역의 코드에 대해서 스캔할 수 있는 방법을 제시하였다.

향후 연구 계획은 본문에서 제안한 힙 영역 코드를 스캔한 후, 이 결과를 정적 분석하는 방법을 연구하고자 한다. 원래의 코드에서 크래시가 날 경우, 이 path가 힙 영역의 코드까지 이어져 exploit 할 수 있는지에 대해서 연구할 예정이다.

참고문헌

[1] Intel “Pin - A Dynamic Binary Instrumentation Tool” https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[2] Nguyen Anh Quynh “Capstone” http://www.capstone-engine.org/index.html
[3] zynmic “zynamics BinNavi” http://www.zynamics.com/binnavi.html