소프트웨어에 적용된 암호화 모듈의 역공학 분석에 관한 고찰[1)]

*김권엽 *최재민 *이상진 *임종인

고려대학교 정보경영공학전문대학원

*kkyoup@korea.ac.kr

A Study on Reverse-Engineering Analysis of Encryption Module applied to Software

*Kim, Kwonyoup **Choi, Jaemin *Lee, Sangjin *Lim, Jongin

*Graduate School of Information Management and Security, Korea University

요약

최근 각종 정보통신 서비스를 제공하기 위한 소프트웨어는 정보보호 기술의 핵심요소인 암호화 모듈을 탑재하고 있다. 여기서 암호화 모듈은 사용자 인증, 콘텐츠 보호, 프라이버스 보호 등의 여러 가지 정보보호 기능을 구현하기 위한 핵심 모듈이다. 그러나 악의적인 공격자에게 암호화 모듈을 사용하는 소프트웨어는 기밀한 정보를 다룬다는 점에서 공격의 대상이 될 수 있다. 또한 사람이 제작하는 소프트웨어는 크기가 커질수록 복잡해질수록 위협 요소는 증가하기 마련이다.[1]

이에 본 논문에서는 암호화 모듈이 탑재된 소프트웨어가 역공학 분석측면에서 악의적인 공격자에게 어떠한 위협 요소를 노출할 수 있는지를 살펴보고 이러한 역공학 분석 공격으로부터 보호하기 위해 소프트웨어 제작에서 고려해야 할 사항을 제시한다.

1. 서론

최근 IT 기술의 발전과 더불어 다양한 소프트웨어가 제작되고 있으며 그러한 소프트웨어 기반의 콘텐츠 서비스가 활발히 진행되고 있다. 그러나 이러한 소프트웨어들은 해킹 과 불법 복제로 인한 지적재산권 침해의 위협에 직면해 있는 실정이다. 실제로 국내에서 개발되고 판매되는 소프트웨어의 경우에도 그 피해가 심각하다. 이러한 상황 속에서 소프트웨어 제작 업체와 소프트웨어 기반의 콘텐츠 서비스 업체는 불법 복제, 재사용 및 유포 등을 방지하기 위하여 수학적으로 안전성이 검증된 표준 암호화 모듈을 적용하고 있다.

일반적인 소프트웨어와 콘텐츠 서비스를 위한 소프트웨어는 블록 암호와 스트림 암호가 주로 사용된다. 스트림 암호는 짧은 길이의 난수열을 이용하여 긴 길이의 의사-난수열인 키 스트림을 생성하는 대칭키 암호이다. 스트림 암호는 평문과 키 스트림의 연관성 유무에 따라 동기식(Synchronous)과 비동기식(Asynchronous)으로 구분된다. DES와 같은 블록 암호는 개발 당시 하드웨어 구현이 용이하도록 설계되었으나, 1990년대 이후 암호 기술이 일반화되면서 AES 등과 같이 소프트웨어적으로 구현이 용이한 알고리즘이 주로 설계되고 있다. 스트림 암호의 경우에도 과거에는 주로 하드웨어 구현이 용이한 LFSR(Linear Feedback Shift Register)에 기반을 둔 알고리즘이 주로 설계되어 왔으나, 근래에는 다양한 응용 환경의 개발과 인터넷 서비스에서 스트리밍 기법이 다수 개발되면서 블록 암호보다 고속 동작이 가능한 소프트웨어 구현에 적합한 스트림 암호 개발이 증가하고 있다. 일반적으로 스트림 암호는 블록 암호보다 5~10배 정도 빠르게 구현되는 장점을 가지고 있다. 블록 암호화 방법은 라운드(round)라고 알려진 몇몇의 약한 논리 연산의 중복으로 구성된다. 이런 몇몇의 일련 된 약한 라운드가 하나의 강력한 블록 암호화 방법을 만든다. 이러한 구조는 설계하기 쉽고 구현이 편리하다는 장점이 있다.

1970년대 유럽을 중심으로 LFSR 기반의 스트림 암호에 대한 개발 및 연구가 본격화되었다. LFSR 자체는 선형이기 때문에 비선형적 논리와 결합하여 스트림 암호를 설계한다. 사용된 비선형적 논리에 따라 LFSR 기반의 스트림 암호는 비선형 필터 생성기, 비선형 결합 생성기, 시각 제어 생성기로 분류된다.

또한 암호화 원리에서 해쉬 함수(Hash Function)는 다양한 역할을 하며 실제로 인증, 디지털 서명 등 여러 가지 기능으로 소프트웨어 적용된다. 해쉬 함수는 비트나 혹은 바이트로 된 임의의 긴 문자열을 입력으로 받아 고정 크기의 결과물을 만들어 낸다. 즉, 인증에 필요한 데이터의 크기가 다양한 경우 해쉬 함수를 이용하여 이 값을 고정된 크기의 값으로 변환할 수 있다. 또한 해쉬 함수는 비밀 정보를 생성하기 위한 난수 생성기(Pseudo-random Generator)로서도 사용된다.

이와 같이 수학적으로 검증된 암호 알고리즘을 적용한 소프트웨어라 할지라도 안전한 것은 아니다. 암호화 모듈을 어떻게 적용하였느냐에 따라 소프트웨어 안전성은 크게 달라질 수 있다.[1] 따라서 본 논문에서는 소프트웨어에서 일반적인 암호화 모듈을 사용할 경우 악의적인 목적을 가진 공격자에게 어떠한 위협요소가 노출되는지 살펴보고, 이러한 공격으로부터 보호하기 위한 고려해야 할 사항을 확인하고자 한다.

1) 본 연구는 과학재단 디지털 정보 획득 기반기술 연구(M10640010005-06N4001-00500)의 지원으로 수행되었습니다.

2. 역공학 (Reverse Engineering) 분석

역공학이라는 용어의 기원은 완성된 제품으로부터 설계 정보를 해석해냈던 하드웨어의 분석에서부터 유래되었는데, 소프트웨어에서는 일반적으로 생명 주기의 초기 단계 생성물인 기능적 묘사, 설계 문서 등을 생성하는 과정을 의미한다. 소프트웨어 생명 주기의 주요 단계인 분석, 설계, 구현 단계들과 관련되는 작업에는 역공학이외에 역공학과 대조되는 개념은 순공학(forward engineering)이 있고, 역공학과 순공학을 포함하는 재공학(reengineering), 재구성(restructuring)등이 있다.[2]

역공학 분석의 위협은 공격자가 프로그램 구조를 수정할 수 있어 그 논리적 흐름에 직접 영향을 줄 수 있다는 점이다. 새로운 코드를 기존 코드에 심어놓을 수 있는데 이러한 기술을 패치(Patch)라는 용어로 사용한다. 즉, 특정 함수가 하는 일에 공격적인 명령을 더하거나 그 방식을 바꿀 수 있다. 이로써 은밀한 기능을 추가하거나 함수를 제거 혹은 무력화할 수 있고 소스 코드 없이 보안 코드를 수정할 수 있다.

일반적으로 역공학 분석 기법의 형태에 있어서 정적 분석(Static Analysis)과 동적 분석(Dynamic Analysis)으로 나뉜다.

가. 정적 분석 (Static Analysis)

역공학 분석에서 그 분류에 따라 화이트-박스(White Box) 분석과 블랙-박스(Black Box) 분석, 그레이-박스(Gray Box) 분석으로 나뉜다.[3] 화이트박스 분석은 소스 코드를 분석하고 이해하는 일이다. 일반적으로는 실행 코드형태의 바이너리 코드를 가지고 있는데 이는 역컴파일과 역어셈블 도구를 이용하여 소스 코드 또는 원시 코드 형태로 얻을 수 있는데 이 역시 화이트 박스 코드 분석으로 간주한다. 블랙박스 분석은 여러 가지 입력을 주어 실행중인 소프트웨어를 검사하여 분석하는 방법으로 악의적인 입력을 소프트웨어에 주입하여 오동작하게 할 수 있다. 그레이 박스는 화이트박스 기술과 블랙박스 입력 검사를 혼합한 형태이다. 이와 같이 정적 분석은 소스 코드 또는 원시 코드 자체를 분석하여 소프트웨어의 구조를 판단하는 방법이다.

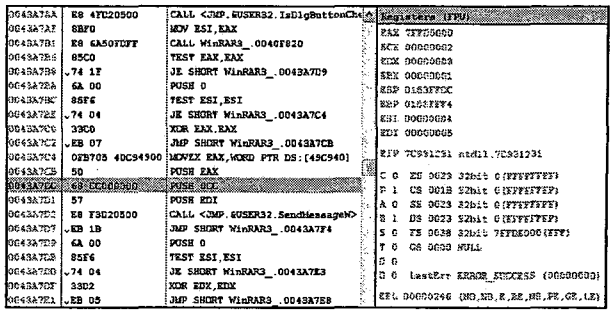
```
.text:0043A7BE                  jz      short loc_43A7C4
.text:0043A7C0                  xor     eax, eax
.text:0043A7C2                  jmp     short loc_43A7CB
.text:0043A7C4 ; 컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴컴
.text:0043A7C4
.text:0043A7C4 loc_43A7C4:
.text:0043A7C4                  movzx   eax, word_49C940
.text:0043A7CB
.text:0043A7CB loc_43A7CB:
.text:0043A7CB                  push    eax
.text:0043A7CC                  push    0CCh
.text:0043A7D1                  push    edi
```

[그림 1] 정적 분석을 이용한 역공학 분석

소스 코드를 보유하고 있지 않은 악의적인 공격자는 [그림 1]과 같이 원시 코드 형태의 분석에 익숙해져 있지만 모든 원시 코드를 분석하는 것은 상당히 어려운 일이다. 따라서 공격자는 분석 대상의 위치를 찾는 것이 역공학 분석에 상당한 부분을 차지하게 된다.

나. 동적 분석 (Dynamic Analysis)

소프트웨어가 실행될 때에 같은 모듈의 같은 함수에서도 각 상황에 따라서 인자 값과 리턴 값이 다른 경우가 발생한다. 이는 입력 값과 프로그램 실행환경 등 여러 가지 변수에서 기인하며, 프로그램의 상태에 따라서 달라진다. 이와 같이 역공학 분석에서는 정적 분석만으로 해결하기에 한계가 있는 부분을 극복하기 위해 동적 분석에 관한 연구가 활발히 진행되었다.[4]



[그림 2] 동적 분석을 이용한 역공학 분석

[그림 2]과 같이 동적 분석 도구를 이용하여 소프트웨어 동작에 따른 각 원시 코드의 레지스터 값, 상태와 메모리 정보 등의 내부적인 동작을 상세히 분석할 수 있다.

3. 관련 연구

최근에는 디지털 콘텐츠 보호에 관한 연구가 진행됨에 따라 소프트웨어의 지적 재산권을 보호하는 기술이 나타나게 되었는데 Encryption, Server-Side execution, Trusted native code, Tamper Proof, Tamper Resistant, Obfuscation 등이 있다. 여기서 Server-Side Execution은 모든 컴퓨터 연산은 서버에서 동작하도록 하여 클라이언트에서는 역공학 분석 공격을 수행할 수 없도록 사전에 보호하는 효과가 있다. 그리고 Trusted native code 기술은 소프트웨어를 각각의 사용자의 시스템에 맞추어 배포하는 것으로 역공학 분석 자체를 어렵게 하는데 효과가 있다. 본 논문에서는 암호화 모듈의 특성상 소스 코드의 구조를 변화시켜 역공학 분석을 효과적으로 보호할 수 있는 기술은 Obfuscation 이다. Obfuscating 변환을 통하여 프로그램은 동일하게 작동하지만, 외부 공격자에게 훨씬 프로그램을 이해하기 어렵게 만드는 기술이다.[5]

4. 암호화 모듈 역공학 분석

본 절에서는 실질적으로 암호화 모듈이 적용된 소프트웨어에서 해당 코드를 분석하는 방법을 소개한다. 본 논문에서 분석 대상은 스트림/블록 암호, 해쉬 알고리즘을 사용한 소프트웨어를 대상으로 하였다. 즉, 스트림/블록 암호, 해쉬 알고리즘은 각각의 고유한 특성이 있으며 이는 각각의 알고리즘을 구현한 소스 코드에도 나타날 뿐만 아니라 원시 코드로 배포되는 소프트웨어에도 나타난다는 것이다.

가. 사회 공학적 특성

일반적으로 소프트웨어 업체의 개발자는 수학적 기반의 암호화 알고리즘을 이해하고 소스코드로 구현하기가 쉽지 않다. 또한 표준화된 암호화 알고리즘은 재사용이 가능하므로 굳이 새롭게 작성하지 않을 뿐만 아니라 재사용되는 암호화 모듈에 어떠한 의심도 가지지 않게 된다. 즉 대부분의 소프트웨어 제작 업체는 암호화 모듈에 대한 안전성을 분석하지 않는 것이 통상적이며 공통적인 모듈을 사용하기 마련이다.

이는 악의적인 공격자에게 상당히 유용한 정보가 될 수 있다. 일반

적으로 원시 코드의 형태의 컴파일러의 특성에 의존하며 이는 이진 데이터로 표현된다. 역공학 분석에서는 이와 같은 특징을 이용하여 기본적으로 제공되는 라이브러리(kernel32.dll, win32.dll ...)를 구분하는 시그네처 기법을 제공한다. 즉, 공통적으로 사용되는 암호화 모듈의 원시 코드에 대한 시그네처 정보를 이용하여 소프트웨어에 적용된 암호화 모듈의 위치를 쉽게 파악할 수 있다.[4]

나. 알고리즘의 특성

위와 같은 방법은 악의적인 분석자가 해당 암호화 모듈을 소유하였을 경우 사용할 수 있는 분석 방법이었다. 본 소절에서는 소프트웨어에 적용된 암호화 모듈을 보유하고 있지 않을 경우 스트림/블록 암호, 해쉬 알고리즘의 특성에 기반을 둔 역공학 분석에 대하여 소개하고 있다.

스트림/블록 암호, 해쉬 알고리즘은 설계 특성상 고유한 정보를 가지고 있다. 알고리즘의 특성상 고유한 정보는 여러 가지가 있지만 모두 다른 개발자가 구현한다는 가정 하에 원시 코드 상에서 그 특성을 구분할 수 있는 것은 고정된 상수 값이다. 알고리즘의 표준 논리는 같지만 구현할 때마다 코드가 달라질 수 있지만 스트림/블록 암호, 해쉬 알고리즘에서 사용하는 초기 상수 값 설정은 달라지지 않는다는 것이다. 예를 들어 SHA-1 해쉬 알고리즘은 [표 1] 과 같이 5개의 초기 상수 값을 가진다.[6]

1	2	3	4	5
0x67452301	0xEFCDAB89	0x98BADCFE	0x10325476	0xC3D2E1F0

[표 1] SHA-1 해쉬 알고리즘의 초기 상수 값

일반적으로 소프트웨어 개발자는 SHA-1 해쉬 알고리즘을 구현할 때 [그림 3] 와 같이 특정 변수에 초기 상수 값을 설정하도록 한다.

```
/* SHA1 initialization constants */
context->state[0] = 0x67452301;
context->state[1] = 0xEFCDAB89;
context->state[2] = 0x98BADCFE;
context->state[3] = 0x10325476;
context->state[4] = 0xC3D2E1F0;
```

[그림 3] SHA-1의 상수 값 설정 소스 코드

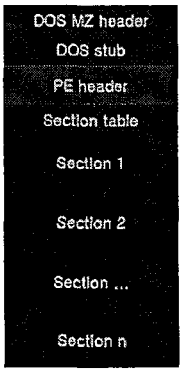
해당 코드를 컴파일한 후 실행 파일을 역어셈블 도구를 이용하여 원시 코드로 변환하면 [그림 4] 와 같은 결과를 확인할 수 있다.

```
mov     dword ptr [eax], 67452301h
mov     ecx, [ebp+arg_0]
mov     dword ptr [ecx+4], 0EFCDAB89h
mov     edx, [ebp+arg_0]
mov     dword ptr [edx+8], 98BADCFEh
mov     eax, [ebp+arg_0]
mov     dword ptr [eax+0Ch], 10325476h
mov     ecx, [ebp+arg_0]
mov     dword ptr [ecx+10h], 0C3D2E1F0h
```

[그림 4] SHA-1의 상수 값 설정 원시 코드

이와 같이 특정 소프트웨어의 원시 코드에서 SHA-1 해쉬 알고리즘의 고유한 상수 값을 확인하게 되면 원시 코드로부터의 복구 정보를 통해서 함수들 간의 호출 관계를 확인할 수 있다.[7]

일반적으로 Win32 환경에서 실행되는 파일은 [그림 5]와 같이 PE(Portable Executable) 파일 포맷 구조를 가진다.



[그림 5] PE 파일 포맷 구조

Win32 환경에서 구현한 소스 코드를 컴파일하여 PE 파일 포맷 구조를 가진 실행 파일은 실제 실행되는 코드를 기계어 코드로 변환하여 Section table의 .text Section에 저장되며 전역 변수, 정적 변수의 경우는 .data Section과 .rdata Section에 해당 정보가 저장된다. 따라서 어떠한 변수를 전역 변수, 정적 변수로 초기화 하게 되면 그 정보는 .data Section, .rdata Section의 Offset 정보로 .text Section에 기계어 코드로 변환되어 저장되고, 지역 변수로 초기 상수 값을 설정한 경우에는 그 상수 값이 기계어 코드로 변환되어 .text Section에 저장된다. 이는 Unix/Linux 기반의 운영체제 사용되는 ELF 실행 파일 포맷과 기본적으로 유사하다.

따라서 Windows, Unix/Linux 기반에서 암호화 모듈을 사용한 소프트웨어를 분석하기 위해서 스트림/블록 암호, 해쉬 알고리즘에서 사용하는 고유한 상수 값으로 소프트웨어에서 사용한 알고리즘을 확인할 수 있다. 또한 해당 암호 알고리즘을 참조하는 코드의 위치는 소프트웨어가 중요한 데이터를 처리하는 부분이라고 판단할 수 있다. 따라서 악의적인 공격자는 그 위치를 분석하여 소프트웨어에 구조적인 취약점(Buffer/Stack Overflow ...)뿐만 아니라 보안상의 취약점까지 쉽게 노출할 수 있다는 것이다. 이와 같이 암호화 알고리즘에서 사용되는 고정된 상수 값 사용을 고려하지 않고 사용한다면 암호화 모듈을 적용한 소프트웨어의 분석하기 위한 악의적인 공격자에게 상당히 가치 있는 정보를 전달하는 행위가 되는 것이다.

본 논문은 관련 연구에서 살펴본바와 같이 소프트웨어에 대한 역공학 분석으로부터 보호하기 위한 Obfuscation이 적용된 소프트웨어서도 확인하였다. 대부분의 Obfuscation 도구는 데이터 처리 과정, 구조체, 변수 이름 등을 변경하는 소스 코드의 복잡성을 증가시켜 소프트웨어의 전체 구조를 이해하기 어렵도록 하는데 그쳤다. 즉, 소스 코드에서 고유한 상수 값을 그대로 사용한다면 Obfuscation 기술을 적용하여도 악의적인 공격자는 충분히 암호화 모듈의 위치를 판단할 수 있다.

다. 암호화 모듈을 적용한 소프트웨어 제작 시 고려 사항

현재 암호화 모듈을 사용하는 콘텐츠 서비스 기반의 소프트웨어뿐만 아니라 사용자 인증, Privacy 보호, 데이터 보호 등의 정보보호 서비스 기반의 소프트웨어 역시 암호화 알고리즘 구현상의 안정성을 간과하고 있는 것이 일반적이다. 본 논문에서 제시한 방법을 이용한다면 대부분의 소프트웨어들이 암호화 알고리즘을 사용하는 위치를 충분히 파악할 수 있음을 확인하였다.

따라서 소프트웨어 제작자는 암호화 알고리즘을 사용하는데 있어서 분석 위치의 노출을 고려해야 한다. 즉, 상수 값을 소프트웨어 제작

자 고유의 방식으로 수정하여 사용하거나 상수 값 자체를 이해하기 어렵도록 변환할 수 있는 Obfuscation 형태의 코드로 변환하여 분석이 용이하지 않도록 해야 한다.

5. 결론

본 논문에서는 소프트웨어에 구현된 암호화 모듈이 악의적인 공격자에게 어떠한 위협 요소를 노출할 수 있는지 살펴보았다. 즉, 일반적으로 사용되는 암호화 모듈을 사용하는 측면에서 나타날 수 있는 역공학 분석 방법과 스트림/블록, 해쉬 알고리즘의 특성에 따른 역공학 분석 방법을 확인하였고 이러한 역공학 분석 기법을 Obfuscation 기술이 적용된 소프트웨어서도 사용할 수 있음을 확인하였다. 또한 이러한 역공학 분석 방법으로부터 보호하기 위해서는 초기 상수 값을 소프트웨어 제작자에 따라 고유하게 변형된 형태로 사용하거나 변수에 할당된 초기 상수 값을 판단하기 어렵도록 하는 Obfuscation이 적용된 암호화 모듈을 사용해야 한다는 것을 제시하였다.

그러나 사용자의 편의성을 고려해야하는 소프트웨어 제작자에게는 암호화 모듈의 안전성을 확인하고 구현하는 것은 쉽지 않는 선택이다. 따라서 향후 연구에서는 일반적인 소프트웨어 적용할 수 있는 안전한 암호화 모듈 제작 방법에 관한 연구를 진행하고자 한다.

6. 참고문헌

[1] Schneier, B. "Secrets and Lies : Digital Security in a Networked World", Wiley, 2000.

[2] E.J.Chikofsky, J.H.Cross, "Reverse Egineering and Design Recovery: A Taxonomy" IEEE Software, Jan 1990.

[3] Gilles Bornot, "Software testing based on formal specifications : a theory and a tool", Software Engineering Journal, v.6 n.6, p. 387-405, Nov. 1991.

[4] H. Ritsch, "Reverse Engineering Programs via Dynamic Analysis", Proc. of IEEE Workshop on Rev. Eng., IEEE Computer Society Press,. Baltimore, Md., p. 192, Nov. 1993.

[5] C. S. Collberg and C. Thomberson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection", U. of Arizona, TR, Mar 2000.

[6] NIST, http://www.itl.nist.gov/fipspubs/fip180-1.htm

[7] 이명재, 우치수, "원시 코드로부터의 정보 복구에 관한 연구", 한국정보과학회 학술발표논문집 p. 457- 460, 1992.