ARM 아키텍처 기반의 리눅스 시스템에서 BOF에 대한 대응

남택준, 강정민, 장인숙, 이진석
국가보안기술연구소
e-mail : tjnam@etri.re.kr

Opposition to BOF in ARM architecture based Linux system

TaekJun Nam, JungMin Kang, InSook Jang, Jinseok Lee
National Security Research Institute, Daejeon, Korea

요 약

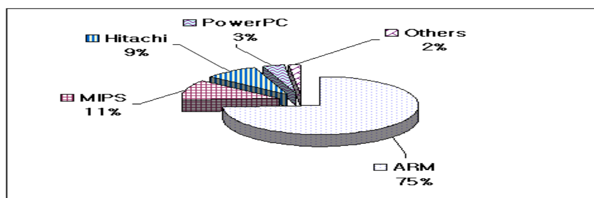
본 논문은 임베디드 장비에 사용되는 코어중 시장의 약 70% 이상을 점유하고 있는 ARM(Advanced RISC Machine) 코어에서의 BOF(Buffer OverFlow)에 대해서 논하고자 한다. 먼저, ARM 아키텍처에서 함수 호출시 스택의 변화에 대해서 기술하고 이 환경에서 시스템 공격 기법 중 가장 빈번한 BOF가 어떻게 이루어지는가에 대해서 설명한다. 그리고 ARM 아키텍처만이 가지는 특징을 이용하여 이에 대처하는 방법을 제안 한다.

1. 서론

현재 임베디드 장비에 들어가는 코어로는 ARM, MIPS, PowerPC등이 있다. 이중 현재(2004년) 70% 이상의 장비에 ARM 코어가 사용되고 있다[그림-1]. 특히, 모바일폰, PDA 등에는 거의 예외 없이 ARM 코어가 사용되고 있으며[그림-2], 모바일폰 같은 경우에는 멀티미디어 데이터를 처리하기 하기 위한 코어(ARM9TDMI), RF와 키패드 입력 등을 처리하기 위한 코어(ARM7TDMI)등 2개 이상의 ARM 코어가 사용되고 있다.

컴퓨팅 시스템에 대한 취약점 공격은 BOF, sniffing, spoofing, race condition등을 통해서 이루어지고 있다. 컴퓨팅 시스템에 대한 공격 중 가장 빈번하게 일어나고 쉽게 구현되는 방식은 BOF 이다. 현재는 이러한 BOF가 x86계열의 윈도우 시스템이나 리눅스 시스템에서 많이 발생하고 있다[1,2]. ARM계열의 리눅스 시스템이나 WIN CE 시스템에서의 BOF 공격에 대한 사례는 많이 보고되지 않고 있다. 하지만, ARM 코어가 사용되는 장비는 더욱 많아질 것이므로 ARM 아키텍처에서의 BOF에 대한 연구는 반드시 필요하다.

본 논문에서는 ARM 아키텍처에서의 BOF에 대해서 논의하고 그에 대한 대응방안을 제안하고자 한다. 본 논문의 구성은 다음과 같다. 2장에서 ARM 아키텍처의 특징에 대해서 다루고[4,5,6] ARM 아키텍처에서 BOF가 어떻게 이루어지는지에 대해서 설



[그림-1] Core별 시장 점유율

[그림-2] ARM Embedded Products

명한다[7,8]. 3장에서는 ARM 아키텍처만의 특징을 이용해서 BOF 공격에 대해서 어떻게 대응할 수 있는지에 대해서 논한다. 4장에서는 본 논문의 결과 및 향후 연구과제에 대해서 설명한다.

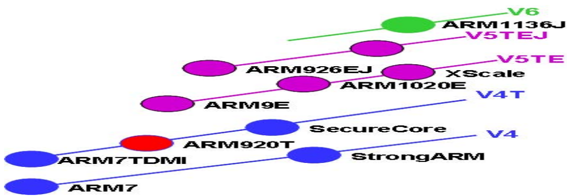
2. 관련연구

이번 장에서는 현재 가장 많이 사용되고 있는 ARM 아키텍처와, 이 ARM 아키텍처에서 발생 가능한 BOF에 대한 배경지식을 제공한다.

2.1. ARM 아키텍처

2.1.1. ARM 아키텍처 진행

현재 ARM 아키텍처는 V6 아키텍처 까지 디자인 되어있다[그림-3]. PDA등에 많이 쓰이던 StrongARM은 V4 아키텍처에 속한다. V4T 아키텍처가 되면서 TDMI 기능들[표-1]이 ARM 코어에 적용 되었다. V4T중 ARM920T가 현재 모바일 폰에서 동영상 데이터의 인코딩, 디코딩에 사용되고 있다. V5TE 아키텍처에는 XScale이 포함된다. XScale은 StrongARM과 같이 인텔에서 ARM 라이센스를 들여와서 만든 코어로서 모듈별 전력관리를 통한 저전력의 특성을 가지고 있기 때문에 현재 많은 PDA에서 사용되고 있다. V5TEJ에는 자바코드를 실행할 때 성능향상을 위한 명령어 셋이 추가되었고, V6 아키텍처는 기존의 아키텍처에 비해서 멀티미디어 데이터 재생 시 최대 4배까지의 성능 향상이 있도록 디자인 되었다[3].



[그림-3] ARM 아키텍처 Progression

종 류	내 용
T	Thumb extension
D	Debug extension on JTAG
M	Hardware multiplier
I	EmbeddedICE extension
E	DSP Enhanced
J	Java extension

[표-1] TDMI 용어 정리

2.1.2. ARM 아키텍처 레지스터 셋

User & System	FIQ	IRQ	Supervisor	Abort	Undef
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und
Program Counter					
PC(R15)	PC(R15)	PC(R15)	PC(R15)	PC(R15)	PC(R15)
Program Status Register					
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und

[그림-4] ARM Register Set

번 호	내 용
r9	sb and SB static base
r10	sl and SL stack limit
r11	frame pointer
r12	intra-procedure-call scratch
r13	stack pointer
r14	link register

[표-2] Predeclared register names

ARM 코어에는 31개의 GPRS(General Purpose Registers)와 6개의 PSRS(Program status Registers)레지스터 셋이 있다. [그림-4]에서 색이 칠해진 부분은 각 모드 에서만 독립적으로 사용하는 레지스터가 한 셋씩 더 있는 것을 의미한다. R0~R14는 범용의 목적으로 사용되고, R15와 CPSR, SPSR은 특정한 목적으로 사용된다. R9부터 R14는 [표-2]와 같은 특정한 이름으로 불린다.

2.1.3. ARM 아키텍처에서의 함수 호출

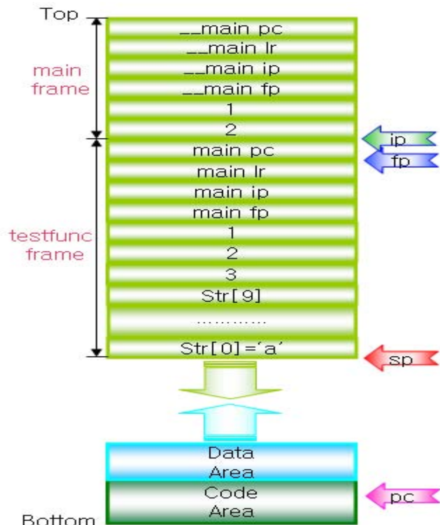
ARM 아키텍처에서의 스택에 대한 연산은 x86 계열과는 다른 방식으로 이루어진다. 우선 명시적인 pop이나 push연산을 하지 않는다. sp(stack pointer)를 조정할 때는 sp에 대한 add나 sub연산을 통해서 직접 한다. [그림-5]는 함수 호출과정의 예를 들기 위한 C코드이고 [그림-6]은 이 C코드에 대한 어셈블리 코드 이다. 그리고 [그림-7]은 testfunc에서 main으로 돌아가기 위한 ldmdb 명령(0x83b4번지)을 수행하기 전의 스택 모양이다.

```
#include <stdio.h>
int testfunc(int a, int b) {
        int     sum;
        char    str[10];
        str[0]='a';
        sum=a+b;
        return sum;
}
void main() {
        int a=1, b=2;
        testfunc(a, b);
        return
}
```

[그림-5] 함수 호출 C코드 예

```
0000837c <testfunc>:
    837c:    e1a0c00d    mov ip, sp
    8380:    e92dd800    stmdb   sp!, {fp, ip, lr, pc}
    8384:    e24cb004    sub fp, ip, #4  ; 0x4
    8388:    e24dd018    sub sp, sp, #24 ; 0x18
    838c:    e50b0010    str r0, [fp, -#16]
    8390:    e50b1014    str r1, [fp, -#20]
    8394:    e3a03061    mov r3, #97 ; 0x61
    8398:    e54b3024    strb    r3, [fp, -#36]
    839c:    e51b2010    ldr r2, [fp, -#16]
    83a0:    e51b3014    ldr r3, [fp, -#20]
    83a4:    e0823003    add r3, r2, r3
    83a8:    e50b3018    str r3, [fp, -#24]
    83ac:    e51b3018    ldr r3, [fp, -#24]
    83b0:    e1a00003    mov r0, r3
    83b4:    e91ba800    ldmdb   fp, {fp, sp, pc}
000083b8 <main>:
    83b8:    e1a0c00d    mov ip, sp
    83bc:    e92dd800    stmdb   sp!, {fp, ip, lr, pc}
    83c0:    e24cb004    sub fp, ip, #4  ; 0x4
    83c4:    e24dd008    sub sp, sp, #8  ; 0x8
    83c8:    e3a03001    mov r3, #1  ; 0x1
    83cc:    e50b3010    str r3, [fp, -#16]
    83d0:    e3a03002    mov r3, #2  ; 0x2
    83d4:    e50b3014    str r3, [fp, -#20]
    83d8:    e51b0010    ldr r0, [fp, -#16]
    83dc:    e51b1014    ldr r1, [fp, -#20]
    83e0:    ebffffe5    bl  837c <testfunc>
    83e4:    e91ba800    ldmdb   fp, {fp, sp, pc}
```

[그림-6] 어셈블리 코드



[그림-7] 함수 호출시 스택의 모양

0x83b4번지의 ldmdb 명령을 수행하면 [그림-7]에서 fp가 가리키고 있는 곳까지 main fp, main ip, main lr 값이 fp, sp, pc 레지스터로 순서대로 들어간다. 그렇게 되면 fp는 __main fp를, sp는 main frame의 2를, pc는 0x83e4부분을 가리키게 되고 그 결과 제어가 main 함수로 돌아가게 된다.

2.2. BOF의 개념과 방식들

[그림-5]의 C코드에서 str 변수에 10자 이상의 스트링을 입력하면 10자를 넘어가는 스트링들은 [그림-7]에 보이는 지역변수 sum(3), b(2), a(1), main fp, main ip, main lr, main pc 값들을 차례로 덮어쓰게 될 것이다.

2.2.1. 리턴주소 덮어쓰기

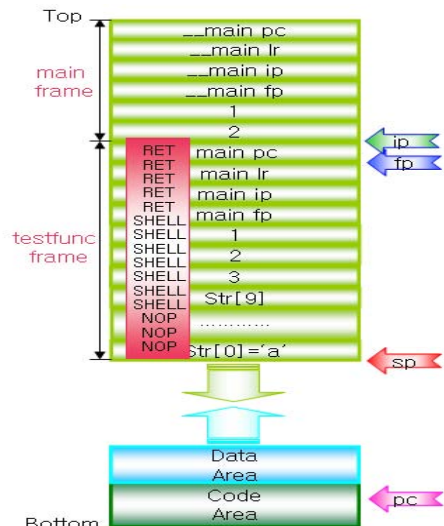
특히 str 변수에 exploit코드를 입력하고 str변수의 주소로 main lr값을 덮어쓴다면 제어가 main으로 넘어가지 않고 exploit코드가 실행된다. 이러한 조작방식이 가장 흔하게 시도되는 BOF 방식이다.

2.2.2. egg shell

리턴주소 덮어쓰기와 프레임 포인터 덮어쓰기를 수행하기 위해서는 str변수의 길이와 스택상의 위치를 예측해야만 가능하다. egg shell은 이러한 예측에 들어가는 비용을 줄이기 위해서 [그림-8]과 같은 방식으로 BOF를 시도하는 것을 말한다. egg shell을 env 환경변수에 저장해 놓았다가 str변수에 대입한다(이러한 일을 해주는 프로그램이 필요). 정확한 str의 주소를 알지 못하더라도 그 근처의 주소(RET들)로 main lr를 덮어쓰면 NOP중 한곳으로 제어가 넘어간 후 exploit이 실행된다.

2.2.3. 시스템 콜 주소와 힙 BOF

현재 BOF에 대처하는 기술들은 PC가 스택영역을 가리키게 되면 이를 BOF로 인식하고 실행을 막는 방식을 사용하고 있다. 악의적인 사용자들은 이러한 대처 기술에 대응하기 위해서 리턴주소를 시스템 콜이 있는 곳의 주소(코드영역)로 덮어쓰거나 힙 공간(데이터영역)에 exploit코드를 삽입하고 이곳으로 제어를 넘기는 방식을 사용한다.

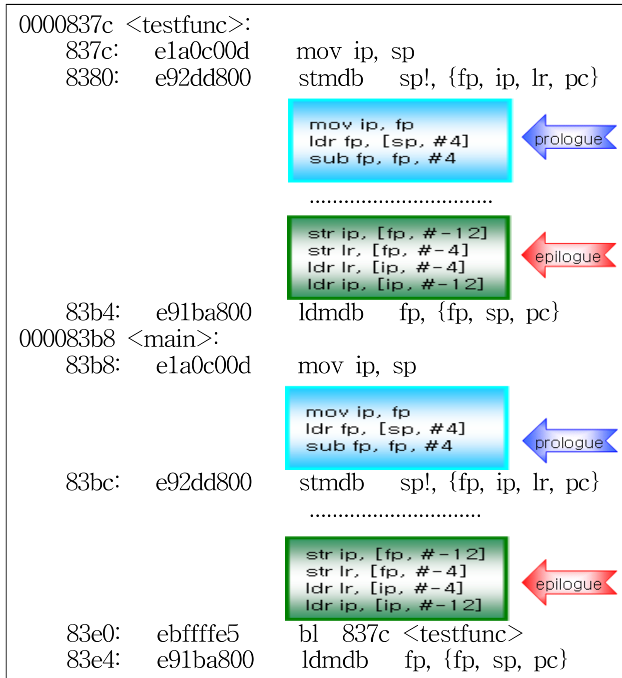


[그림-8] egg shell 방식

3. ARM 아키텍처에서의 BOF에 대한 대응

[그림-6]코드를 보면 함수 내부의 stmdb와 ldmdb

명령어 사이에서는 lr, ip 레지스터는 거의 사용이 되지 않는다. 컴파일러가 함수의 시작과 끝부분에 [그림-9]에 있는 것과 같이 lr, ip 레지스터를 이용하는 프롤로그와 에필로그 코드를 삽입하면 리턴주소와 프레임 포인터 값이 변경되어도 저장되어 있는 lr과 ip를 다시 덮어쓰므로 exploit코드의 실행을 막을 수 있다. testfunc함수내의 프롤로그와 에필로그의 분석은 [표-3], [표-4]와 같다.



[그림-9] 프롤로그와 에필로그가 추가된 코드

코 드	내 용
mov ip, fp	fp(__main pc가 있는 곳)를 ip에 저장
ldr fp, [sp, #4]	ip-4값을 fp에 저장. 두 번째 세 번째 명령어는 sub fp, ip, #4와 같음
sub fp, fp, #4	

[표-3] 프롤로그 코드 분석

코 드	내 용
str ip, [fp, #-12]	ip(__main pc가 저장된 곳의 주소)를 main fp가 저장된 곳에 덮어씀
str lr, [fp, #-4]	main으로 돌아갈 리턴주소(lr)를 main lr이 저장된 곳에 덮어씀
ldr lr, [ip, #-4]	lr에 ip값에서 -4만큼 떨어진 곳에 위치한 __main lr값을 넣어줌
ldr ip, [ip, #-12]	ip값에서 -12만큼 떨어진 곳에 위치한 __main fp 값을 ip에 넣어줌

[표-4] 에필로그 코드 분석

프롤로그와 에필로그 코드를 생성하기 위해서는 컴파일러, 툴 체인, glibc를 수정해야 된다.

4. 결론 및 향후 연구과제

본 논문에서는 ARM 아키텍처에 대한 특징과 BOF에 대한 개념 및 방식들에 대해서 논의 하였다. 그리고 함수 내부(stmdb와 ldmdb사이)에서 유저레벨의 C코드를 위해서는 사용되지 않는 lr, ip 레지스터를 이용해서 리턴주소와 프레임 포인터를 악의적인 사용자로부터 보호하기 위한 프롤로그, 에필로그 코드를 제안하였다.

프롤로그와 에필로그에 사용된 명령어의 수는 7개이다. 이것은 기존의 방식에서는 없는 코드로 실행시간과 프로그램의 크기를 늘리는 결과를 가져온다. 많은 실험을 통해서 이 오버헤드에 대한 정량적 수치를 얻는 것이 필요하다. 이 방식은 ARM gcc 크로스컴파일러 3.2.1 버전을 기준으로 이루어졌다. 컴파일러에 따라서 함수내부에서 lr, ip레지스터에 대한 조작이 달라질 수 있으므로 이러한 부분에 대한 연구가 추후에도 계속 이루어져야 할 것이다.

참고문헌

[1] Lawrence Ricci(eMVP) and Larry McGinnes(CPL), "Embedded System Security (Designing Secure Systems with Windows CE)", Applieddata.net, 2003.

[2] "Software Considered Harmful : Why Software is Insecure",http://www.ee.oulu.fi/research/ouspg.

[3] "The ARM Instruction Set Architecture", http://www.arm.com/products/CPUs/architecture .html.

[4] David Seal, "ARM Architecture Reference Manual", 2nd Ed. Addison-Wesley, December 27, 2000.

[5] A. van Someren and C. Atack, "The ARM RISC Chip, A Programmer's Guide", Addison-Wesley, January 1994.

[6] Steve B. Furber, "ARM System-on-Chip Architecture", 2nd Ed. Addison-Wesley, August 25, 2000.

[7] Aleph one, "Smashing The stack For Fun And Profit", Phrack Magazine 49.

[8] CoreSecurity Team, "Vulnerablilities in your code-Advanced Buffer Overflows(abo)", 2003.