M1522.000800 System Programming

Fall 2018

System Programming

Buflab Report

Kim Bochang
2014-16757

1. <lab> Buffer Lab

버퍼랩의 목적은 버퍼-오버플로우 취약점을 이용해 프로그램의 스택을 조작,
스택의 리턴어드레스를 우리가 삽입한 스트링으로 덮어씌워 공격자가 원하는 인스트럭션으로 이동, 공격자가 삽입한 코드를 실행하여 프로그램의 동작을 바꾸는 것이다.

2. Implementation

getbuf 함수에서 사용자가 입력한 스트링을 길이를 파악하지 않고 그대로 스택에 덮어쓴다는것을 이용했다. objdump로 bufbomb 파일을 disassemble하여 bufbomb의 동작을 파악한 다음, getbuf와 다른 함수들의 stack frame을 파악하고 stack을 우리가 원하는 코드로 덮어씌워 목표한 코드가 실행되게 한다.
(2014-16757 buflab repository에 buflab-stack.ods에 buflab을 진행하면서 파악한 스택구조를 적어놓았다.)
The buffer lab consists of five phases: Smoke, Sparkler, Firecracker, Dynamite and Nitroglycerin.
…

2.1 Smoke

0	candle	getbuf's stack			Ebp-relative
			>	argument	8
			>	ret adr	4
		now ebp	>	old ebp	0
	Ebp-relativ			old ebx	-4
%eax	-48				
%ebx	argument				
	*val				
%ebx	1				
(if val = 0statement executed)					
				gets started	-48
get's arguments		now esp	>	Ebp-48	-68

<getbuf's stack frame>

getbuf의 스택 프레임을 보면, gets의 인자로 %ebp-48 (%ebp-0x30)에 해당하는 인자를 넘겨주어 그 주소부터 gets가 스택에 사용자가 입력한 스트링을 덮어씌우게 된다는것을 알 수 있다. 따라서 그로부터 56바이트를 덮어씌워 return address를 smoke function의 주소로 하게되면 getbuf가 리턴할때 smoke의 주소로 이동할 것이기 때문에, 56바이트의 스트링을 인풋으로 주고, 52~56바이트째를 smoke의 주소인 0x08048be8로 설정하면 된다.
Answer - <candle.txt>

2.2 Sparkler

08048c12	fizz	fizz's stack			
				Argument#2	12
				Argument#1	8
%eax	Arg 1 (val 1)			ret adr	4
%edx	Arg 2 (val2)	now ebp	>	old ebp	0
%ecx	~val1 << 8				
ox804d128	cookie				
		now esp	>		24
mycookie	0x23518db1				
b1	8d	51	23		
b10110001	b10001101	b01010001	b00100011		
val1	0xffffffff				
11111111	11111111	11111111	11111111		
~val1 << 8					
0	0	0	0		
val2					
0	0	0	0		

<fizz's stack frame>

candle에서 했던것처럼 injection string의 52~56바이트에 fizz의 주소인 0x08048c12를 넣어
getbuf가 종료되면 fizz로 이동하게 한다.

그리고 fizz 내부의 instruction을 보면 val1과 val2를 인자로 받아 (~val1 << 8) & cookie 의 결과가 val2와 같은지를 비교하는데, 이때 val1 = 0xffffffff이면 ~val<<8 = 0x0이 되어 cookie가 무슨값이든 0이 된다는 점에 착안, val1에 -1, val2에 0이 들어가도록 injection string을 만든다.

Getbuf 실행 직후 fizz가 실행되기 때문에, fizz의 argument로 -1과 0이 인식되게 하려면
injection string의 56~60바이트는 fizz의 return address, 60~64바이트가 val1, 64~68바이트가 val2가 되므로 그와 같이 injection string을 작성하였다.

Answer - <fizz.txt>

2.3 Firecracker

bang함수에서는 global_value의 값을 cookie & 0x0f0f와 비교하기 때문에,
bang함수의 인스트럭션을 실행하기전에 global_value의 값을 바꿔줘야
우리가 원하는 결과를 얻을 수 있게된다.

따라서 먼저 injection string의 52~56바이트를 getbuf에서 gets로 스트링을 받아와서 기록하는 스택의 첫 주소인 0x55683df0 로 지정하여, 우리가 만든 인스트럭션을 실행하게 한다. (injection string의 0바이트부터)

Global value의 메모리 내의 위치는 0x0804d120이고, mycookie = 0x23518db1,
mycookie & 0x00000f0f = 0x0d01이므로 결국 global value에 이 값을 덮어씌워주면 된다.

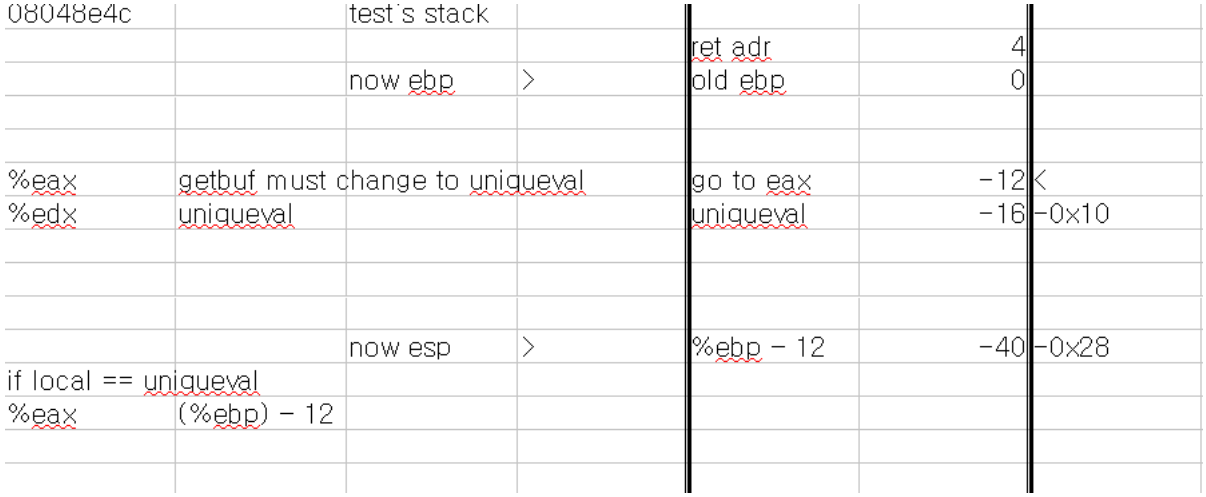
따라서 이러한 작업을 하는 코드를 injection string에 집어넣어주면 우리가 원하는 결과가 나오게 된다.

이 코드가 끝나고 ret 문을 실행할때, 현재 esp에 있는 부분을 리턴어드레스로 간주하게 되기 때문에 injection string의 56~60바이트를 bang함수의 주소인 0x08048c81로 주었다.

Answer - <fire.txt>

이때 앞의 코드를 90으로 채워넣은것은 nop를 이용해 조금 더 편하게 injection string을 다루기 위함이다.

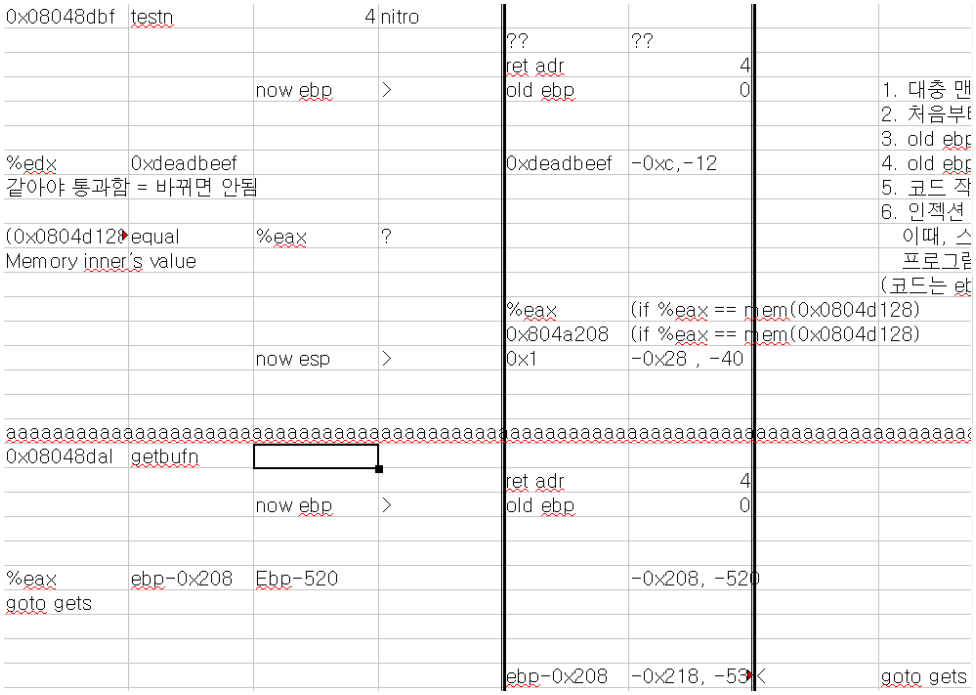
2.4 Dynamite



<test's stack frame>

test 함수에서는 먼저 스택이 오염되었는지를 판단하고, val이 cookie의 값과 같게되면 validate를 하게된다.
따라서 우리는 val의 값을 cookie로 바꿈과 동시에, injection string의 길이가 56바이트를 넘지 않게해서 test의 stack frame을 망가트리지 않고, test의 ebp와 같은 레지스터 값들을 모두 보전시켜 줘야한다.
이를 위해 우리가 삽입한 코드에서 직접 리턴 어드레스를 스택에 푸시하고, getbuf에서 test의 %ebx, %ebp가 손상되지 않도록 injection code의 44~48, 48~52바이트에 각각 test의 %ebx, %ebp의 값을 넣어줘야 할것이다.
(getbuf의 assemble 코드에서 gets를 호출한 이후 스택에서 test의 ebx와 ebp를 복구하기 때문이다.)
%ebx는 0 (설정 안해줘도 결과에는 영향을 안미침), %ebp (0x55683e50)을 설정해준다.
또한 val은 ebp – 0xc에 저장되어 있으므로, 코드를 통해 val에 cookie를 넣어준다.
test가 getbuf를 실행한뒤 실행되어야 하는 다음 인스트럭션의 주소가 0x8048e65이므로
그 주소를 stack에 push해주고, val에 cookie를 집어넣은뒤 리턴을 하게되면 우리가 원하는 결과가 나온다.
Answer - <dynamite.txt>

2.5 Nitroglycerin



<testn & getbufn ‘s stack frame>

우리가 해야하는 작업은 getbufn의 리턴값이 메모리의 (0x0804d128) 주소에 저장된 값이랑 같아지게 해야하는것이므로, dynamite때와 같이 getbufn함수에서 testn 함수의 스택프레임을 건드리지 않으면서, testn의 %ebp를 getbufn이 정상실행 됐을때처럼 복구하고, getbufn의 리턴값인 %eax를 0x0804d128에 저장된 값으로 바꿔버리는것이다.

프로그램이 실행될때마다 스택의 시작 어드레스가 바뀌므로, 앞에서 썼던 특정 함수의 %ebp의 값 자체를 저장해놨다 사용하는것은 불가능하므로, 다음과 같은 사실을 이용한다.

getbufn이 리턴할때의 %esp의 값은, testn이 getbufn을 실행할때의 %esp의 값과 같고,

testn의 %ebp = %esp + 0x28이라는것을 이용한다. 따라서 실행해야 하는 인스트럭션은 다음과 같다.

먼저 %ebp의 값을 복구한뒤, testn에서 다음에 실행될 인스트럭션의 주소(0x08048dd1)를 stack에 push 하고, %eax에 메모리의 0x0804d128에 저장된 값을 넣어주고, 리턴하는것이다.

이에 맞춰 injection string을 작성할건데, injection string의 길이는 512 + 16 = 528 바이트가 되어야 하고, 리턴주소 부분인 524~528바이트 부분에 들어가야 할 값을 정해야한다.

이때 getbufn의 %ebp의 값은 0x55683e20에서 +-240정도 변하게 되므로,

injection string은 ebp-0x208부터 들어가니까 0x55683c18 +-240 → 0x55683b28~0x55683d08

의 시작범위를 가지게 된다. 따라서 injection 스트링의 앞의 480바이트를 모두 nop를 채워넣어

중간부터 실행되어도 우리가 원하는 인스트럭션으로 slip 되게 한다음,

480바이트~520바이트 부분에 우리가 원하는 인스트럭션을 작성하고, 524~528바이트에

0x55683d08을 넣으면 우리의 코드는 스택의 시작위치에 관계없이 실행될 것이다.

Answer - <nitro.txt>

3. Conclusion

<explain what you have learnt in this lab. What was difficult, what was surprising, and so on>

buflab에서 배운점은, gets와 같이 string의 길이를 파악하지 않고 스택에 바로 스트링을 집어넣는 함수를 실행할 경우에는 프로그램에 버퍼 오버플로우 취약점이 생기고, 이를 이용해 공격자들이 자신들이 원하는 코드를 실행시켜 마음대로 프로그램의 결과를 조작하거나, 허용되지 않은 코드를 실행할 수 있게 되니

이런 함수는 되도록이면 사용하지 않아야겠다는 것을 느꼈다.

랩을 진행하면서 어려웠던 점중 하나는, 프로그램을 실행한 뒤 5초가 지나면 프로그램이 종료되게 하여 gdb를 자유자재로 사용할 수 없어 어디서 문제가 발생했는지 정확하게 파악하기 힘들었다는 점이다.

Segmetation fault가 계속 일어나는데 왜 일어나는지를 몰라서 한참 헤매다가 5초가 되기전에 빠르게 gdb를 실행하는것으로 겨우 이유를 알아내어 고친적이 있었다.

또한 어셈블리 코드를 다루는 것이다 보니, 숫자 하나를 잘못눌러서 만든 코드가 제대로 실행이 안되거나 하는 문제가 발생했을때도 어디서 문제가 발생했는지 찾기가 힘들어 디버깅이 힘들었다.