#Phase 1~3의 목표

프로그램을 실행하면 test 함수가 호출이 되고 그 함수 안에서 getbuf가 호출이 되는데 이떄 getbuf 함수는 get함수를 호출한다. 우리의 목표는 get함수에서 버퍼 오버플로우 통해서 원하는 함수들을 호출해 정상적으로 그 함수를 종료 시키는 것이다.

|  |
| --- |
| Test 함수의 어셈블리어 표현 |
|  |
| Getbuf함수의 어셈블리어 표현 |
| 확인해보면 스택 포인터를 40만큼 감소시키고 그것을 인자로 해서 gets함수를 호출한다. 즉 스택을 버퍼로 사용하였고 그 것의 크기는 40이다. |

#Phase1

Phase1은 getbuf 호출이후 touch1 함수를 호출하면 된다.

Getbuf 함수의 어셈블리어 표현을 확인해보면 gets함수의 버퍼는 40을 가지고 있다.

즉 gets함수의 호출 후 스택 메모리의 상태는 아래와 같다.

|  |  |
| --- | --- |
| Stack frame  For test |  |
| Return Address | 00 00 00 00 00 00 21 0f |
| gets함수 버퍼 | %Rsp-0x28 |

여기서 우리는 gets 함수에 40바이트 이상의 값을 입력해주면 return address에 접근을 해서 조작을 할 수 있게 된다. 우리가 원하는 return address는 touch1의 호출 주소이므로 touch 1의 주소를 확인해보자

|  |
| --- |
| Touch1의 어셈블리어 표현 |
|  |

확인해 보면 touch1의 시작 주소는 00 00 55 55 55 55 5f 2e이다. 또한 호출만 하면 일련의 과정을 거치고 정상적으로 끝이 난다. 그러므로 우리는 리턴 주소만 확인을 하면 된다. X86-64의 스택 메모리에서 데이터 저장은 리틀 에디안 방법을 사용해 데이터를 저장하므로 40 바이트의 임의의 값을 넣어주고 그후 2e 5f 55 55 55 55 00 00까지 입력해 주면 return address는 00 00 55 55 55 55 5f 2e로 변경될 것 이다.

우리가 입력할 값을 answer1.txt로 새롭게 저장하고 hex2raw 프로그램으로 변환 후 정답을 입력해 보자.

|  |
| --- |
| Answer1.txt |
|  |
| Haw2raw로 변환 |
|  |
| 정답 입력 후 결과 확인 |
|  |

#Phase 2

Phase2는 getbuf 호출 후 touch2함수를 호출 해야 한다.

위에서 확인한 것과 같이 gets 함수의 버퍼는 40바이트를 가지고 있다. 우선 touch2의 함수의 주소와 내용을 확인하기 위해 어셈블리어 표현을 확인해보자.

|  |
| --- |
| Touch2 어셈블리어 표현 |
|  |

edi값과 cookie값이 같으면 정상적으로 phase2가 완료가 된다. 즉 우리가 수행 해야 할 것은 touch2 함수를 호출하고 edi값을 조작하는 것이다.

그러기 위해서는 우리가 gets함수를 통해 우리 원하는 동작을 수행하는 코드를 삽입 해야 한다. 먼저 gets 함수 호출 후 스택 메모리에 상태는 아래와 같다.

|  |  |
| --- | --- |
| Stack frame  For test |  |
| Return Address | 00 00 00 00 00 00 21 0f |
| gets함수 버퍼 | %Rsp-0x28 |

우리는 여기서 gets함수가 끝나고 우리가 삽입한 코드를 수행해야 하므로 return address는 %rsp-0x28의 주소로 설정 해야 한다.

그 후 touch2의 주소값을 넣어 주기 위해 삽입할 코드에 push 명령을 통해 스택 메모리에 touch2의 주소값을 넣어줘야 한다.

gets함수 리턴후 삽입 코드에서 push 명령어 수행 후 스택 메모리는 아래와 같은 형태가 된다.

|  |  |
| --- | --- |
| Stack frame  For test |  |
| Return address ( touch2 ) | 삽입 코드가 끝나면 push명령을 통해 저장된 touch2의 주소 값으로 점프 수행 |
| 삽입 코드 내용 + 패딩  ( 합쳐서 총 40바이트 ) |  |

그러기 위해서는 우리는 getbuf 에서 break 명령어 수행후 rsp값을 확인해 그때의 rsp-0x28값을 확인해야 한다. 또한 touch2의 주소를 확인해야 하고 우리가 원하는 코드를 어셈블리어로 표현하고 그것을 기계어 코드로 표현 해야 한다.

|  |  |
| --- | --- |
| Rsp-0x28 값 확인 | |
|  | Getbuf 호출시 rsp값은 0x5561abf0이다. 여기서 우리가 필요한 값은 rsp-0x28 이므로 계산을 해보면 그 값은  0x5561abc8 이다.  그러므로 우리는 이 값을 첫번 째 return address로 사용한다. |

Touch2 의 주소는 위의 어셈블리어 표현에서 00 00 55 55 55 55 5f 62 임을 확인할 수 있다. 즉 이것을 삽입 코드에서 pushq 명령어로 스택 메모리에 저장한다.

|  |  |
| --- | --- |
| Cookie값 확인 | |
|  | 쿠키 값이 0x70206b77 이므로 이 값을 edi 레지스터로 mov 해주는 삽입 코드를 작성하면 된다. |
| 삽입 코드 작성 ( vi를 통해 phase2.s 작성 ) | |
|  | |
| Gcc 통해 기계어 코드 변환 후 변환 코드 확인 | |
|  | |

즉 우리는 위의 기계어 코드를 순서대로 삽입 후 나머지 패딩에 아무 값을 입력 후 마지막에 return address를 0x5561abc8 로 변경하는 값을 입력해주면 된다.

|  |
| --- |
| 입력 값 작성 |
| Phase1과 같이 리턴 주소는 리틀 에디안 방법을 따라 입력해줘야한다. |
| Hax2raw로 변환 |
|  |
| 결과 확인 |
|  |

#Phase 3

Phase3는 getbuf 호출 후 touch3함수를 호출 해야 한다.

위에서 확인한 것과 같이 gets 함수의 버퍼는 40바이트를 가지고 있다. 우선 touch3의 함수의 주소와 내용을 확인하기 위해 어셈블리어 표현을 확인해보자.

|  |
| --- |
| Touch3 어셈블리어 표현 |
|  |

이번 함수는 입력 값을 문자열로 받고 hexmatch 함수를 통해 cookie값과 같은지 확인을 한다.

Phase2의 풀이와 같이 삽입 코드에서 rdi값을 수정을 하면 된다.

이때는 문자열을 스택 메모리에 저장을 하고 그 주소를 rdi에 저장을 한다.

|  |  |
| --- | --- |
| Cookie값 아스키 코드값으로 표현 | |
|  | 주어진 cookie 값은 70206b77이므로 이 값 각각 아스키 코드값으로 변환하면 된다 |
| |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 0 | 2 | 0 | 6 | b | 7 | 7 | | 0x37 | 0x30 | 0x32 | 0x30 | 0x36 | 0x62 | 0x37 | 0x37 |   이후 마지막에 입력값으로 string의 끝을 알리는 null값 0x00을 넣어준다. | |

이 값들을 저장하기 위해서 입력 값 마지막에 추가를 한다.

즉 우리가 입력해야 할 리턴 주소 그 다음에 이 문자열을 추가해서 rsp+0x8 의 주소인 스택 메모리에 이 값들을 저장한다. 그리고 rdi 값은 rsp+0x8로 지정을 하면 rdi는 문자열의 시작 주소가 된다.

Rsp+0x8 의 값은 위에서 구한 값을 사용하면 0x5561abf0 +0x8 = 0x5561abf8 이다.

|  |
| --- |
| 삽입 코드 작성 ( vi를 통해 phase3.s 작성 ) |
|  |
| Gcc 통해 기계어 코드 변환 후 변환 코드 확인 |
|  |

|  |
| --- |
| 입력 값 작성 |
| Phase1과 같이 리턴 주소는 리틀 에디안 방법을 따라 입력해줘야한다. (phase 2와 리턴 주소 동일)  또한 문자열을 스택 메모리에 저장하기 위해 리턴 주소 그다음에 순서대로 문자들의 아스키 코드를 작성한다. |
| Hax2raw로 변환 |
|  |
| 결과 확인 |
|  |

#Phase 4~5의 목표

위의 phase2,3과는 달리 스택의 주소가 실행때 마다 달라져서 버퍼의 주소를 특정지을수 없어 삽입 코드의 실행이 불가능하다. 그러므로 rop attack을 사용해 해결 해야 한다. 이때 우리는 start\_farm와 end\_farm 함수 사이에서 필요한 명령을 찾아 사용 해야 한다. 이때 우리는 ret (= c3)으로 끝나는 명령들을 모아야 한다. 그래야 필요 명령 수행 하고 ret을 해 그 다음 찾은 명령 수행 해 순차적으로 진행이 가능하다.

각각 필요한 명령들의 주소를 순서대로 gadget 1 … n 이라하면

그 후 리턴 주소에다가 1부터 차례대로 쌓아 스택 메모리에 올리면 gadget1 수행 후 리턴해서 gadget2 수행을 하고 다시 반복해서 n까지 수행 하게 된다. 그 후 마지막에 우리가 호출하고자 하는 함수의 주소 값을 쌓아 올린다.

#Phase 4

Phase4에서는 rop attack 을 통해 touch 2 함수를 호출을 하면 된다.

우리는 touch2 의 인자 rdi에 쿠키 값을 넣어 줘야 한다. 사용 할 수 있는 명령어는 ppt를 확인 하면 mov, pop 정도이다. Mov를 사용해 직접 값을 넣기에는 적당한 함수가 없으므로 pop을 사용해 레지스터에 값을 넣고 mov연산을 통해 rdi에 복사 하는 방법으로 수행을 하자.

아래는 우리가 수행 해야 할 명령어들을 기계어 코드로 표현한 것이다. (ppt참고)

|  |  |
| --- | --- |
| Pop rax 후 ret | 58 c3 |
| Mov %rax, %rdi 후 ret | 48 89 c7 c3 |

이제 start\_farm 부터 end\_farm 사이에 이 기계어 코드 구성이 있는지 찾으면 된다.

이때 90은 아무 수행 안하는 코드임을 인지하고 찾아보자

|  |  |  |
| --- | --- | --- |
| Pop rax 후 ret |  | 58이 시작하는 부분의 주소는  Addval\_201 시작 주소 +8이 된다. |
| Mov %rax, %rdi 후 ret |  | 48이 시작하는 부분의 주소는 setval\_299 시작 주소 +6 이 된다. |

Gdb실행 후 함수의 주소값이 달라지므로 실행후 각 필요 함수의 주소 값을 찾아보자

|  |  |  |
| --- | --- | --- |
| Pop rax 후 ret |  | 즉 우리가 입력 해야 할 gadget1 의 주소는 555555556148이 된다. |
| Mov %rax, %rdi 후 ret |  | 즉 우리가 입력 해야 할 gadget2 의 주소는 555555556166이 된다. |

이제 우리가 입력 해야 할 값들을 생각해보면

버퍼크기만큼 패딩 -> pop명령어 주소 -> pop할 값 (cookie 값) -> mov 명령어 주소 -> touch2의 주소 이다. 이것들 다 적어보면 아래와 같다.

|  |  |
| --- | --- |
| 입력 값 작성 | |
|  | 앞에서 와 같이 모든 값들은 리틀 에디안 저장 방법을 따라 입력한다. |
| Hax2raw로 변환 | | |
|  | | |
| 결과 확인 | | |
|  | | |

#Phase 5

Phas54에서는 rop attack 을 통해 touch 3 함수를 호출을 하면 된다.

이때 우리는 phase3에서 한 것과 같이 문자열을 스택 메모리에 저장 후 rdi 값에 그 스택 메모리의 주소 값을 전달 해야 한다. 버퍼의 주소를 특정 지을 수 없어 우리가 사용할 수 있는 주소 값은 rsp뿐이다.

이때 start\_farm 과 end\_farm 사이에 add\_xy 함수 를 확인해보자

|  |
| --- |
| Add\_xy 어셈블리어 표현 |
|  |

확인해보면 rax에 인자로 받은 두 값을 더해 저장을 수행한다.

즉 우리는 주소값과 원하는 숫자값을 rdi, rsi 에 저장을 해두고 add\_xy를 수행해 rax값에 저장 후 mov rax rdi 로 touch3의 인자로 보내야한다.

하지만 주어진 범위 내 함수에서 rsi로 mov하는 인스트럭션 코드는 없고 rdi로 mov하는 인스트럭션은 있으므로 ( phase4에서 확인 ) 주소 값을 rdi에 저장하고 esi로 mov하는 인스트럭션을 확인해보자 ( 주소값은 48비트이므로 esi로 모두 옮기기 불가능 )

Start\_farm 과 end\_farm 사이에서 찾을수 있는 esi로 mov하는 인스트럭션은 오직 한 개 뿐이었다.

|  |
| --- |
| Mov ecx, esi |
|  |

이제 ecx 로 mov하는 인스트럭션들을 모두 찾으면 아래와 같다. 여기서도 오직 한 개 뿐이었다..

|  |
| --- |
| Mov edx ecx |
|  |

이제 edx로 mov하는 인스트럭션들을 찾아보자 여기서도 오직 하나만 찾을 수 있다.,

|  |
| --- |
| mov eax, edx |
|  |

위에 세가지의 함수를 통해 최종적으로 mov eax, esi 를 수행할수 있다. 이 과 정 전에 pop 명령어로 rax에 원하는 숫자값을 넣으면 rsi에는 우리가 원하는 숫자값을 저장 할 수 있다.

이제 주소값을 불러오기 위해 mov rsp rax함수를 찾아보자 ( rsp에서 rdi로 바로 mov하는 인스트럭션은 없고 rax에서 rdi로 오는 함수는 이미 알고 있으므로 rsp에서 rax를 찾음 )

|  |
| --- |
| Mov rsp, rax |
|  |

위에 내용을 조합해서 우리가 입력해야할 코드의 순서는 아래와 같다.

|  |  |
| --- | --- |
| 버퍼 크기만큼 패딩 | 40 바이트 |
| Pop rax 후 ret |  |
| 우리가 원하는 숫자값 | Rax에 원하는 숫자값 pop 명령어로 저장하기 위함  이때 숫자값은 rsp의 값을 rax로 mov하는 시점부터 cookie의 아스키 코드 변환 값들이 있는 곳까지의 차이다. |
| mov eax, edx | 원하는 숫자 값 rsi에 저장 |
| Mov edx ecx |
| Mov ecx, esi |
| Mov rsp, rax | 주소값 rax에 저장 |
| Mov %rax, %rdi 후 ret | 주소값 rdi에 저장 |
| Add\_xy의 주소값 | Rdi와 rsi의 값 합친후 rax에 저장 |
| Mov %rax, %rdi 후 ret | Rdi에 계산 결과 저장 |
| Touch 3의 주소 | 이제 touch3을 호출한다. |
| Cookie 아스키코드 변환 값들 | 이곳의 주소 값을 rdi에 저장해야함 |

즉 우리가 pop해야할 숫자 값은 0x20이 된다.

이제 위에서 구한 함수들의 모든 주소값을 구하면 아래와 같다.

gdb 실행 후 주소값이 바뀌므로 getbuf에 break를 걸어준 후 disas 을 해서 주소값을 찾는다.

|  |  |  |
| --- | --- | --- |
| Mov ecx, esi |  | 우리가 필요한 값은 주소값 + 5이므로 입력 값은 00005555555561cd가 된다. |
| Mov edx ecx |  | 우리가 필요한 값은 주소값 + 6 이므로 입력 값은 000055555555625a가 된다. |
| mov eax, edx |  | 우리가 필요한 값은 주소값 + 7 이므로 입력값은 0000555555556287 이 된다. |
| Mov rsp, rax |  | 우리가 필요한 값은 주소값 +6 이므로 입력값은 0000555555556224 이 된다. |
| Add\_xy |  | Add\_xy는 모든 함수 내용 다 필요하므로 입력값은 0000555555556195가 된다. |

지금 까지의 내용을 조합해서 입력값을 적으면 아래와 같다.

|  |
| --- |
| 입력 값 작성 |
|  |
| Hex2raw 로 변환 |
|  |
| 결과 확인 |
|  |